فصل اول

مفاهیم مقدماتی

۱-۱- مقدمه

در این فصل خواهیم دید که چرا هوش مصنوعی موضوعی با ارزش برای مطالعه است، همچنین در مورد تعاریف و کاربردهای آن صحبت خواهیم نمود.

ما انسان‌ها لقب علمی هوموساپینز- انسان هوشمند- را به خود داده‌ایم چرا که توانایی‌های فکری برایمان بسیار حائز اهمیت است. هزاران سال تلاش کرده‌ایم که بفهمیم چگونه فکر می‌کنیم، یا در واقع چگونه مشتی ماده می‌تواند دریافت کند، بفهمد، پیش بینی کند و در جهان اطراف که بسیار پیچیده‌تر و بزرگتر از خود انسان است تغییر ایجاد کند. شاخه هوش مصنوعی از این هم فراتر رفته و تلاش می‌کند نه تنها موجودیت‌های هوشمند را درک کند، بلکه موجودیت هوشمند بسازد.

هوش مصنوعی یکی از علوم جدید محسوب می‌شود. تحقیق در زمینه هوش مصنوعی پس از جنگ جهانی دوم آغاز شد و نامگذاری آن در سال ۱۹۵۶ انجام گرفت. شاید یک دانشجوی رشته فیزیک احساس کند که همه ایده‌های جدید و خوب توسط گالیله و نیوتن و انیشتن و دیگر دانشمندان قبلا مطرح شده است. درحالی که هوش مصنوعی علمی است که هنوز درآن مسائل حل‌نشده بسیار زیادی وجود دارد.

درحال حاضر هوش مصنوعی شامل زیر شاخه‌های بسیاری از علوم مختلف می‌شود. از زمینه‌های همه‌منظوره مثل ادراک و استدلال منطقی گرفته تا زمینه‌های تخصصی‌تر مثل بازی شطرنج، اثبات قضایای ریاضی، شعر و تشخیص بیماری‌ها. هوش مصنوعی فعالیت‌های هوشمندانه را سیستماتیک و اتوماتیک می‌کند و بنابراین به نحوی با هرچیزی که در دامنه فعالیت‌های هوشمندانه قرار دارد مرتبط می‌شود. با توجه به این مفهوم هوش مصنوعی یک زمینه همگانی است.

۱-۲- هوش مصنوعی[۱] چیست؟

تاکنون گفتیم که هوش مصنوعی موضوع جالب توجهی است اما هنوز نگفته‌ایم که هوش مصنوعی چیست؟ تعاریف مختلفی از هوش‌مصنوعی وجود دارد. شکل ۱-۱ تعاریف مختلف هوش‌مصنوعی (که در هشت کتاب دانشگاهی آمده است) را نشان می‌دهد. این تعاریف در دو بعد اصلی تغییر می‌کنند. آن‌هایی که در خانه‌های بالا قرار دارند مربوط به فرآیندهای فکری و استدلال هستند در حالی که آنهایی که در خانه‌های پایین قرار دارند تمرکز بیش‌تری روی رفتار دارند. همچنین، تعاریف سمت چپ برای سنجش موفقیت از لحاظ رسیدن به کارایی

۱- artificial intelligence (AI)

انسان است، در مقابل تعاریف سمت راست یک مفهوم ایده آل از هوش است که ما به آن عقلانیت می‌گوییم. یک سیستم عقلایی (دارای عقلانیت) است اگر با توجه به دانشی که دارد کار درست را انجام دهد.

سیستمی که عقلایی فکر می‌کند	سیستمی که مشابه انسان فکر می‌کند
"مطالعه‌ی استعدادهای ذهنی از طریق استفاده از مدل‌های محاسباتی" (کارنیاک و مک درموت،۱۹۸۵)	"تلاشی نو و مهیج برای این که کامپیوترها را قادر به فکر کردن کنیم... ماشین‌های با فکر و با حس تشخیص واقعی" (هاگلند ۱۹۸۵)
"مطالعه محاسباتی که درک،استدلال و عمل کردن را ممکن می‌سازند" (وینستون ۱۹۹۲)	"خودکارسازی فعالیت‌هایی که ما به تفکر انسانی نسبت می‌دهیم. فعالیت‌هایی مثل تصمیم گیری، حل مسئله، یادگیری و ..." (بلمن ۱۹۷۸)
سیستمی که عقلایی رفتار می‌کند (عمل می‌کند)	سیستمی که مشابه انسان رفتار می‌کند (عمل می‌کند)
"هوش محاسباتی، مطالعه طراحی عامل‌های هوشمند است." (پول ات ال ۱۹۹۸)	"هنر ایجاد ماشین‌هایی که وظایفی را انجام می‌دهند که انجام آن‌ها توسط انسان‌ها نیاز به هوش دارد" (کورزویل ۱۹۹۰)
"هوش مصنوعی درارتباط با رفتار هوشمندانه ساخته‌های مصنوعی است " (نیلسون ۱۹۹۸)	"مطالعه این که چگونه کامپیوترها را قادر به انجام اعمالی کنیم که در حال حاضر انسان، آن اعمال را بهتر انجام می‌دهد" (ریچ و نایت ۱۹۹۱)

شکل ۱-۱ چند تعریف هوش مصنوعی از که در چهار دسته طبقه بندی شده‌اند.

در تاریخ هوش‌مصنوعی هر چهار رویکرد فوق‌الذکر مورد استفاده قرار گرفته‌اند. همانگونه که انتظار می‌رود نوعی تنش بین رویکردهایی که حول انسان متمرکز شده‌اند با رویکردهایی که حول عقلانیت متمرکزند وجود دارد.[1] یک رویکرد انسان محور باید یک علم تجربی باشد که شامل فرضیات و تایید تجربی است درحالی که یک رویکرد عقل‌گرا شامل ترکیبی از ریاضیات و مهندسی است. در ادامه به بررسی هریک از این چهار رویکرد می‌پردازیم.

۱-۲-۱- سیستمی که مشابه انسان رفتار می‌کند (عمل‌می‌کند): رویکرد تست تورینگ

تست تورینگ (که توسط آلن تورینگ در سال ۱۹۵۰ پیشنهاد شد) به‌منظور ایجاد تعریفی برای (رضایت از) هوش عملکردی (یک سیستم) طراحی شد. تورینگ به جای ارائه لیستی طولانی (و شاید بحث برانگیز) از ویژگی‌هایی لازم برای هوشمندی (یک سیستم) تستی را بر پایه ناتوانی در تمییز دادن از موجودی که

[1]- باید به این مطلب اشاره کنیم که منظور از تمایز قائل شدن بین رفتار "انسانی" و "عقلانی"، این نیست که انسانها لزوماً غیر عقلانی رفتار می‌کنند یا این که دچار "بی ثباتی روحی" و یا "جنون" هستند. بلکه کافی است دقت کنیم که ما اغلب دچار اشتباه می‌شویم. مثلاً اگرچه همه ما قوانین شطرنج را می‌دانیم، ولی همه ما استاد برجسته شطرنج نیستیم، و اینکه متاسفانه همه ما نمره الف در امتحان نمی‌گیریم.

هوشمندی‌اش غیرقابل انکار است(انسان) پیشنهاد کرد. کامپیوتر در صورتی این تست را پشت سر می‌گذارد که پرسشگر (که خود یک انسان است) پس از ارائه سؤالاتی (برای کامپیوتر) نتواند تشخیص دهد که پاسخ‌های داده شده از طرف یک انسان است یا یک کامپیوتر. طراحی چنین سیستمی نیازمند زمان بسیار زیادی خواهد بود. برای این منظور کامپیوتر مورد نظر باید دارای قابلیتهای زیر باشد:

- پردازش زبان طبیعی برای این که قادر باشد با زبان انگلیسی (یا زبانهای انسانی دیگر) ارتباط برقرار کند.
- نمایش دانش برای ذخیره اطلاعاتی که می‌داند یا بدست می‌آورد.
- استدلال خودکار (اتوماتیک) برای استفاده از اطلاعات ذخیره شده جهت پاسخ‌گویی به سؤالات و بدست آوردن نتایج جدیدتر
- یادگیری ماشین برای تطبیق با شرایط جدید و برون‌یابی و شناسایی الگوها

تست تورینگ تعامل فیزیکی بین کامپیوتر و پرسشگر را مدنظر قرار نداده است (که البته این کار به عمد انجام شده است) چرا که شبیه سازی فیزیکی یک شخص امری غیرضروری برای آزمایش هوشمندی است. هرچند که تست دیگری به نام تست کامل تورینگ وجود دارد که درآن پرسشگر با استفاده از یک سیگنال ویدیوئی توانایی‌های ادراکی (سیستم) را مورد آزمایش قرار می‌دهد. برای پشت سر نهادن تست کامل تورینگ کامپیوتر نیاز به قابلیت‌های زیر دارد:

- بینایی ماشین برای تشخیص اشیاء.
- رباتیک برای تغییر دادن و جابجایی اشیاء.

۶ شاخه علمی فوق‌الذکر تشکیل دهنده بخش عمده‌ای از هوش مصنوعی هستند و تورینگ برای طراحی تستی که تا ۵۰ سال معتبر باقی مانده شایسته تقدیر است. با این حال محققان تلاش کمی برای گذراندن تست تورینگ انجام داده‌اند زیرا معتقدند که مطالعه اصول پایه‌ای هوش مهم‌تر و باارزش‌تر از تکرار و همانند سازی یک نمونه موجود است. تلاش برای "پرواز مصنوعی" وقتی منجر به موفقیت شد که برادران رایت و دیگران تقلید از پرندگان را کنار گذاشته و به ائرودینامیک پرداختند. در حقیقت هیچ‌گاه کتابهای مهندسی ائرودینامیک هدف رشته خود را ساختن "ماشینهایی که به قدری شبیه به کبوتران پرواز می‌کنند که حتی کبوترهای دیگر را به اشتباه می‌اندازند" تعریف نمی‌کنند.

۱-۲-۲- سیستمی که مشابه انسان فکر می‌کند: رویکرد مدل‌سازی شناختی

اگر بخواهیم بگوییم یک برنامه داده شده مانند انسان فکر می‌کند، بایستی راه‌هایی برای تعیین چگونگی فکر کردن انسان پیدا کنیم. باید به بررسی عملکرد واقعی و درونی ذهن انسان بپردازیم. برای انجام این کار دو روش وجود دارد: یکی درون‌گرایی (سعی در شناخت افکار در زمان‌هایی که خطور می‌کنند) و دیگری بوسیله آزمایشات روانشناسی. زمانی که یک مدل دقیق از ذهن بدست آوردیم، می‌توانیم آن را در قالب یک برنامه کامپیوتری پیاده‌سازی کنیم. اگر ورودی و خروجی‌های برنامه و الگوهای زمانی آن مطابق رفتارهای انسانی متناظر با آن باشد، یعنی برخی از مکانیزم‌های برنامه قابلیت‌های عملکردی انسان را نیز دارا خواهند بود. بعنوان

مثال ناول و سیمون[۱] که GPS (حل‌کننده عمومی مسئله) را طراحی کردند، تنها به این که برنامه‌شان توانایی حل مسئله دارد قانع نبودند. آنها بیشتر علاقه‌مند به مقایسه مراحل استدلال برنامه با مراحل استدلال یک انسان بودند. علم شناختی، مدل‌های کامپیوتری را از AI و تکنیکهای تجربی را از روانشناسی کنار هم می‌گذارد تا تئوری‌های دقیق و قابل آزمایش از عملکرد ذهن انسان ارائه کند.

گرچه علم شناختی زمینه جالب توجهی است و به خودی خود می‌تواند در قالب یک دایرةالمعارف بگنجد، اما ما در این کتاب به تفصیل درباره آن صحبت نخواهیم کرد. فقط گاهی توضیحاتی درباره تشابهات و تفاوتهای تکنیکهای AI و شناخت انسان خواهیم داد. زیرا علم شناختی واقعی مبتنی بر تحقیقات تجربی برروی انسان‌ها یا حیوانات است و ما فرض می‌کنیم که خواننده کتاب برای آزمایش‌های خود فقط به کامپیوتر دسترسی دارد (و نه امکاناتی برای آزمایش برروی انسان و حیوان).

در روزهای آغازین AI رویکردها به درستی از یکدیگر تمییز داده نمی‌شدند. ممکن بود نویسنده‌ای ادعا کند که یک الگوریتم خوب اجرا می‌شود و بنابراین مدل خوبی برای عملکرد انسان است یا برعکس. نویسنده‌های عصر مدرن این دو نوع ادعا را از هم تفکیک می‌کنند. این تفکیک هم AI و هم علم شناختی را قادر ساخته که با سرعت بیشتری پیشرفت کنند. هر دو زمینه به غنی‌سازی یکدیگر به خصوص در زمینه‌های بینایی و زبان‌های طبیعی ادامه می‌دهند. در زمینه بینایی به طور خاص اخیراً از طریق یک رویکرد یکپارچه که شواهد روانشناسی اعصاب و مدل‌های محاسباتی را در کنار هم در نظر می‌گیرد پیشرفت بسیاری صورت گرفته است.

۱-۲-۳- سیستمی که عقلایی فکر می‌کند: رویکرد قوانین تفکر

فیلسوف یونانی ارسطو، یکی از اولین کسانی بود که تلاش کرد تا "فکر کردن به صورت صحیح" را کد کند (بدین معنی که فرآیندهای استدلال غیرقابل انکاری را ایجاد کند). قیاس‌های صوری او الگوهایی را برای ساختارهای برهانی به وجود آورد که در صورت در نظرگرفتن مقدمه‌های درست همواره به نتایج درست می‌رسند. بعنوان مثال، "سقراط یک انسان است؛ همه انسان‌ها فانی هستند؛ بنابراین سقراط فانی است." تصور می‌شد که این قوانین از عملکرد ذهن انسان تبعیت می‌کنند و مطالعه آنها سرآغاز رشته منطق شد.

در قرن نوزده منطق‌شناس‌ها نمادسازی دقیقی از گزاره‌های موجود در عالم و روابط بین آنها ایجاد کردند. (این را با نمادهای معمول ریاضی که گزاره‌های مربوط به مساوی یا نامساوی بودن اعداد را می‌سازند، مقایسه کنید.) در سال ۱۹۶۵ برنامه‌هایی وجود داشتند که بااستفاده از توصیف مسئله به صورت نمادگذاری منطقی توانایی حل هر مسئله قابل حلی را داشتند.[۲] نهضت معروف به منطق‌گرایی در هوش مصنوعی، امیدوار است برنامه‌هایی بسازد که بوسیله آنها سیستم‌های هوشمند خلق کند.

دو مانع اصلی بر سر راه رسیدن به این رویکرد وجود دارد. اول اینکه بیان دانش غیررسمی در قالب عبارات رسمی (که برای نمایش منطقی امری ضروری است) کار دشواری است. به خصوص زمانی که این دانش صد در صد قطعی نباشد. دوم اینکه فرق زیادی بین توانایی حل مسئله "به طور اصولی" و انجام این کار درعمل وجود دارد. حتی مسئله‌هایی با تعداد فرض اندک، می‌توانند منابع محاسباتی هر کامپیوتری را کاملا مصرف کنند (مگر

---

[۱]- Newell and Simon ۱۹۶۱

[۲]- اگر جوابی وجود نداشته باشد، برنامه ممکن است هیچگاه جستجو برای یافتن پاسخ را متوقف نکند.

آنکه دستورالعمل‌هایی برای اینکه کدامیک از مراحل استدلال را اول امتحان کنند) داشته باشند.این دو مانع بر سر راه هر تلاشی برای ساختن سیستم‌های استدلال‌کننده کامپیوتری وجود دارند.

۱-۲-۴- سیستمی که عقلایی رفتار می‌کند (عمل می‌کند): رویکرد عامل‌های عقلایی

عامل به چیزی گفته می‌شود که عمل می‌کند. ولی از عامل‌های کامپیوتری انتظار می‌رود که ویژگی‌های دیگری مثل عمل کردن تحت کنترل خودمختار، دریافت و درک محیط اطراف، پایداری در یک بازه زمانی طولانی، سازگاری با تغییر و قابلیت بر عهده گرفتن اهداف دیگران را نیز داشته باشند تا بدین ترتیب از "برنامه صرف" متمایز شوند. یک عامل عقلایی عاملی است که بهترین نتیجه (در شرایطی که عدم قطعیت وجود دارد، بهترین نتیجه مورد انتظار) را تولید کند.

در رویکرد "قوانین تفکر" AI تاکید کلی بر استنباط صحیح بود. استنباط صحیح (اغلب) بخشی از عامل عقلایی است، چرا که یکی از راه‌های رسیدن به رفتار (عمل کردن) عقلایی، استدلال منطقی است (تا به این نتیجه برسیم که یک کنش باعث رسیدن به هدف می‌شود و سپس بر اساس این نتیجه، آن عمل را انجام دهیم). البته، تنها راه رسیدن به عقلانیت (عقلایی بودن)، استنباط صحیح نیست (یعنی استنباط صحیح تنها بخشی از عقلانیت است). چرا که گاهی موقعیتهایی پیش می‌آید که در آنها هیچ کار درست قابل اثباتی (کاری که درست بودن آن اثبات شود) وجود ندارد، اما به هرحال باید کاری انجام دهیم. همچنین، گاهی رفتارهای عقلایی وجود دارند که بدون استنباط بدست می‌آیند. به عنوان مثال کشیدن دست از روی یک اجاق داغ یک عکس العمل غیر ارادی است که بسیار موفقیت آمیزتر از یک عکس العمل کند است که با سنجش و بررسی انجام می‌شود.

تمامی **مهارت‌های** مورد نیاز تست تورینگ، برای اعمال کنش‌های عقلایی مورد نیازند. بنابراین نیاز به توانایی نمایش دانش داشته تا بوسیله آن استدلال کنیم. زیرا این امر ما را قادر می‌سازد در موقعیت‌های گوناگون تصمیمات خوبی بگیریم.

پس مطالعه AI در زمینه طراحی عامل عقلایی دو مزیت دارد: اول اینکه بسیار عمومی‌تر از رویکرد "قوانین تفکر" است. چرا که در این رویکرد، استنباط صحیح تنها یکی از چندین مکانیزم ممکن برای رسیدن به عقلانیت است. دوم اینکه از لحاظ علمی نسبت به رویکردهایی که مبتنی بر رفتار انسانی یا تفکر انسانی هستند، قابلیت توسعه بیش‌تری دارد زیرا استاندارد عقلانیت (عقلایی بودن) به روشنی تعریف شده و کاملاً عمومی است. در مقابل، رفتار انسان تطبیق یافته در یک محیط خاص و تحت شرایط خاص است و خود محصول یک فرایند تکاملی پیچیده و تا حد زیادی ناشناخته است که هنوز تا رسیدن به تکامل فاصله زیادی دارد. بنابراین، این کتاب تاکیدش بر روی اصول عمومی عامل‌های عقلایی و اجزای لازم برای ساخت آنهاست. کمی جلوتر خواهیم دید که علی‌رغم سادگی آشکاری که به وسیله آن می‌توان مسئله را بیان کرد، در زمان حل مسئله ملاحظات بسیار زیادی پیش می‌آیند. فصل ۲ برخی از این ملاحظات را با جزئیات بیشتری بیان می‌کند.

یک نکته مهم که باید به خاطر داشت: خواهیم دید که در محیط‌های پیچیده رسیدن به عقلانیت کامل (همیشه کار درست انجام دادن) ممکن نیست. چرا که نیازهای محاسباتی آن بسیار زیاد است. در بیشتر بخش‌های کتاب تحلیل‌ها را با فرض عقلانیت کامل برای عامل پیش خواهیم برد. این امر مسئله را ساده‌تر می‌کند و تنظیمات مناسبی برای اکثر موضوعات بنیادی در این رشته به وجود می‌آورد.

۱-۳- مبانی هوش مصنوعی

در این بخش به تاریخچه‌ای مختصر از اصول علمی که ایده‌ها، نقطه نظرها و تکنیک‌هایی به AI بخشیده‌اند، می‌پردازیم. مانند هر تاریخچه‌ی دیگری، در این تاریخچه نیز ناچاریم بر روی تعداد اندکی از افراد، وقایع و ایده‌ها تمرکز کنیم و بقیه را با وجود مهم بودن کنار بگذاریم. تاریخچه را پیرامون یک سری سوالات سازمان دهی می‌کنیم. هر چند نمی‌خواهیم این مطلب را القاء کنیم که این سوالات تنها سوالاتی هستند که علوم مورد بحث، مورد توجه قرار می‌دهند و یا اینکه این علوم همگی در راستای AI به عنوان هدف غایی‌شان عمل کرده‌اند.

۱-۳-۱- فلسفه (۴۲۸ قبل از میلاد تاکنون)

- آیا قوانین رسمی می‌توانند برای استخراج نتیجه‌گیری‌های معتبر استفاده شوند؟
- چگونه از مغز فیزیکی، ذهن غیرفیزیکی به وجود می‌آید؟
- دانش از کجا به وجود می‌آید؟
- چگونه دانش به عمل(کنش) منتهی می‌شود؟

ارسطو (۳۴۸ تا ۳۲۲ قبل از میلاد) اولین کسی بود که مجموعه‌ای دقیق از قوانینی که بخش عقلایی ذهن را توصیف می‌کنند، فرمول‌بندی کرد. او یک سیستم غیررسمی قیاس صوری را برای استدلال گسترش داد که اساس آن نتایج بر اساس فرض‌های اولیه به طور ماشینی تولید می‌شدند. چندی بعد رامون لال[۱] عقیده‌ای را مطرح کرد که آن براساس استدلال سودمند توسط محصول (مصنوعی) مکانیکی انجام می‌شد. توماس هابز[۲] (۱۵۸۸ تا ۱۶۷۹) بر این باور بود که استدلال مانند محاسبات عددی چون جمع و تفریق است که ما بی‌صدا در ذهن خود انجامشان می‌دهیم. اتوماتیک سازی محاسبات به خوبی در حال انجام شدن بود که حدود سال ۱۵۰۰ میلادی لئوناردو داوینچی[۳] (۱۴۵۲ تا ۱۵۱۹) ماشینی برای محاسبات طراحی کرد ولی موفق به ساخت آن نشد(مدل‌هایی که اخیراً ساخته‌اند، نشان داده‌اند که طرح وی عملی بوده است). اولین ماشین محاسباتی شناخته شده حدود سال ۱۶۲۳ توسط دانشمند آلمانی ویلهلم شیکارد[۴] (۱۵۹۲ تا ۱۶۳۵) ساخته شد. اگرچه پاسکال که در سال ۱۶۴۲ توسط بلیز پاسکال (۱۶۲۳ تا ۱۶۶۲) ساخته شد معروفتر است. براساس نوشته‌های پاسکال "ماشین عملیات ریاضی آثاری تولید می‌کند که به تفکر نزدیک‌تر از تمام کنش‌های دیگر حیوانات است". گاتفرد ویلهلم لایبنیز[۵] (۱۶۴۶ تا ۱۷۱۶) وسیله‌ای مکانیکی با هدف انجام عملیات روی مفاهیم (به جای اعداد) ساخت، ولی دامنه آن بسیارمحدود بود.

اکنون که ما ایده‌ای از مجموعه قوانینی که می‌توانند بخش عقلایی و مجرد ذهن را توصیف کنند، داریم قدم بعدی این است که ذهن را به صورت یک سیستم فیزیکی در نظر بگیریم. رنه دکارت[۶] (۱۶۵۰- ۱۵۹۶) اولین بحث روشن و صریح را در مورد تفاوت ذهن و ماده و مشکلات ناشی از آن مطرح کرد. یکی از این مشکلات این

۱- Ramon Lull
۲- Thomas Hobbes
۳- Leonardo davinci
۴- Wilhelm Schickard
۵- Gottfried Wilhelm Leibniz
۶- Rene Descartes

است که به نظر می‌رسد نسبت دادن ماهیت مطلقاً فیزیکی به مفهوم ذهن فضای کمی را برای اختیار(اراده آزاد) باقی می‌گذارد. اگر ذهن کاملاً توسط قوانین فیزیکی کنترل شود، ذهن اختیاری بیش از یک سنگ که "تصمیم می‌گیرد" به سوی مرکز زمین حرکت کند، ندارد. گرچه دکارت حامی سرسخت قدرت استدلال بود اما از "دوگانه انگاری"[۱] نیز پشتیبانی می‌کرد. او بخشی از ذهن یا روح یا نفس را خارج از طبیعت می‌دانست که از قید قوانین فیزیکی آزاد هستند. در نقطه مقابل، به نظر وی حیوانات این خصوصیت دوگانه را دارا نیستند و می‌توان با آنها به منزله ماشین رفتار کرد. یک جایگزین برای دوگانه‌انگاری، "ماده گرایی"[۲] است که می‌گوید عملکرد مغز بر پایه قوانین فیزیکی، ذهن را می‌سازد و اراده آزاد راهی است برای دریافت گزینه‌های موجود که در فرایند تصمیم گیری ظاهر می‌شوند.

با فرض وجود یک ذهن فیزیکی که دانش را دستکاری می‌کند، مسئله بعدی یافتن منبع مولد دانش است. جنبش تجربه‌گرایی که با منطق بیکن[۳] توسط فرانسیس بیکن[۴] (۱۶۲۶-۱۵۶۱) شروع می‌شود با این گفته جان لاک[۵] (۱۷۰۴-۱۶۳۲) شناخته می‌شود که می‌گوید: "ما چیزی را درک نمی‌کنیم مگر آنکه اول آن را احساس کنیم." آنچه دیوید هیوم[۶] (۱۷۷۶-۱۷۱۱) در رساله‌ای در باب طبیعت انسانی (۱۷۳۹) عنوان کرد چیزیست که اکنون آن را به عنوان اصل استقراء می‌شناسیم که می‌گوید: قوانین عمومی از طریق نمایش یک رابطه تکرار شونده بین اجزاشان بدست می‌آیند. بر پایه کارهای لدویگ ویتگنشتاین[۷] (۱۸۸۹ تا ۱۹۵۱) و برتراند راسل[۸] (۱۸۷۲ تا ۱۹۷۰) محفل به رهبری رادولف کارنپ[۹] (۱۸۹۱ تا۱۹۷۰) نظریه مثبت‌گرایی منطقی را گسترش دادند. این نظریه مدعیست که همه دانش بوسیله نظریه‌های منطقی‌ای که به جملات شهودی (که متناظر با ورودی حاصل از سنسورهاست) مرتبطاند قابل توصیف هستند[۱۰]. در "نظریه تایید" رادولف کارنپ و کارل همپل (۱۹۰۵ تا ۱۹۹۷) در مورد چگونگی حصول دانش از تجربه تحقیق کردند. کتاب کارنپ "ساختار منطقی جهان" (۱۹۲۸) یک رویه محاسباتی صریح و روشن را برای استخراج دانش از تجارب ابتدایی تعریف می‌کند. این نظریه احتمالاً نخستین نظریه در مورد ذهن در قالب یک فرآیند محاسباتی بود.

عنصر نهایی در تفسیر فلسفی از ذهن ارتباط بین دانش و کنش(عمل) است. این سوال یک سوال اساسی برای AI است. زیرا هوشمند بودن همان‌قدر که مستلزم استدلال است، به کنش نیز نیازمند است. به علاوه تنها با درک این که کنش‌ها چگونه توجیه می‌شوند، می‌توانیم عاملی بسازیم که کنش‌هایش(اعمالش) موجه(یا عقلایی) باشند. ارسطو اعتقاد داشت که کنش‌ها با یک رابطه منطقی بین اهداف و آگاهی از نتیجه کنش توجیه می‌شوند.

۱- Dualism
۲- Materialism
۳- نسخه جدید ارغنون ارسطو یا ابزار فکر
۴- Francis Bacon
۵-John Locke
۶-David Hume
۷-Ludwig Wittgenstein
۸-Bertrond Russell
۹-Rudolf Carnap
۱۰- در این منظر، تمام اظهارات معنی دار می توانند بوسیله تجزیه و تحلیل معنای کلمات یا استفاده از تجربیاتمان تصدیق یا تکذیب شوند. به علت این که بیشتر این قواعد از حوزه متافیزیک خارجند، (درست مثل مفهوم و خیال)، پوزیتیویسم منطقی در بعضی محافل محبوب نیست.

الگوریتم ارسطو ۳۰۰ سال بعد توسط ناول و سیمون در برنامه GPS شان (حل‌کننده عمومی مسائل) پیاده سازی شد که ما اکنون آن را سیستم طرح‌ریزی رگرسیون می‌نامیم.
تحلیل برپایه هدف مفید است، اما این نوع تحلیل در مواردی که چند گزینه برای رسیدن به هدف داریم و یا مواردی که اصلاً گزینه‌ای برای رسیدن به هدف نداریم کاربرد ندارد. آنتونی آرنولد [۱] (۱۶۱۲ تا ۱۶۹۴) به درستی یک فرمول کلی برای تصمیم گیری در چنین مواردی ارائه کرد. جان استوارت میل [۲] (۱۸۷۳-۱۸۰۶) در کتاب کاربرد گرایی (۱۸۶۳) ایده معیار تصمیم‌گیری منطقی در تمام زمینه‌های فعالیت بشر را تقویت کرد. یک نظریه رسمی یا تفصیلی‌تر برای تصمیم‌گیری در بخش بعدی مورد بحث قرار می‌گیرد.

۱-۳-۲- ریاضیات (سال ۸۰۰ میلادی تا کنون)

- قوانین رسمی برای استخراج نتایج معتبر چه هستند؟
- چه چیزهایی قابل محاسبه هستند؟
- چگونه با اطلاعات غیرقطعی استدلال کنیم؟

بسیاری از ایده‌های هوش‌مصنوعی توسط فلاسفه گردآوری شده‌اند اما برای تبدیل این ایده‌ها به یک علم رسمی احتیاج به سطحی از فرمول‌سازی ریاضی در سه زمینه اساسی محاسبات، منطق و احتمال است.
ایده منطق صوری به فلاسفه یونان باستان برمی‌گردد اما توسعه و گسترش ریاضیاتی آن با کارهای جورج بول [۳] (۱۸۱۵ تا ۱۸۶۴) آغاز شد. وی بر روی جزئیات منطق گزاره‌ای یا منطق بولی کار کرد(بول ۱۸۴۷).
در سال ۱۸۷۹ گاتلوب فریج [۴] (۱۹۲۵-۱۸۴۸) با بوجود آوردن منطق مرتبه اول، منطق بولی را عمومیت بخشید (منطقی که امروزه به عنوان اساسی‌ترین سیستم نمایش دانش استفاده می‌شود). آلفرد تارسکی (۱۹۰۲-۱۹۸۳) نظریه مرجع را معرفی کرد و در آن درمورد چگونگی تناظر اشیا درون منطق با اشیا جهان واقعیت بحث کرد. مرحله بعد، تعیین محدوده کارهای قابل انجام با محاسبات و منطق است.
تصور می‌شود که اولین الگوریتم قابل ملاحظه، الگوریتم "Euclid" برای محاسبه بزرگترین مقسوم علیه مشترک باشد. مطالعه الگوریتم‌ها به خوارزمی ریاضیدان ایرانی قرن نهم بر می‌گردد. کسی که با نوشته‌هایش اعداد عربی و جبر را به اروپا معرفی کرد. بول و دیگران الگوریتم‌ها را از دیدگاه استنتاج منطقی مورد بحث قرار دادند و در اواخر قرن نوزدهم تلاش‌هایی برای فرمول‌بندی استدلال ریاضی عمومی به صورت استنتاج منطقی صورت گرفت.
در سال ۱۹۰۰ دیوید هیلبرت [۵] (۱۸۶۲ تا ۱۹۴۳) لیستی از ۲۳ مسئله را ارائه کرد و به درستی پیش بینی نمود که ریاضیدانان در تمام قرن به آن‌ها مشغول خواهند بود. آخرین مساله هیلبرت این بود که آیا الگوریتمی برای تصمیم‌گیری درمورد درستی گزاره‌هایی منطقی که دارای اعداد طبیعی هستند وجود دارد یا خیر (مساله معروف تصمیم‌گیری). در واقع هیلبرت می‌خواست بداند که آیا محدودیتهای بنیادی برای رویه‌های موثر اثباتی وجود

۱-antoine Arnauld
۲-John Stuart Mill
۳-George Boole
۴-Gottlob Frege
۵-David Hilbert

دارد یا خیر؟ در سال ۱۹۳۰ کرت گودل[۱] (۱۹۰۶-۱۹۷۸) نشان داد که برای اثبات هر عبارت صحیح در منطق مرتبه اول فریج و راسل[۲]، رویه‌های موثری وجود دارد اما منطق مرتبه اول (در آن زمان) نمی‌تواند اصل استقرای ریاضی برای اعداد طبیعی را پوشش دهد. او در سال ۱۹۳۱ نشان داد که محدودیتهای بنیادی در این زمینه وجود دارد. قضیه کامل نبودن او نشان داد که در هر زبانی که به حد کافی گویا برای توصیف ویژگیهای اعداد طبیعی باشد گزاره‌های درستی وجود دارند که تصمیم ناپذیرند، یعنی درستی آنها را با هیچ الگوریتمی نمی‌توان تایید کرد.

این نتیجه اساسی همچنین بدین صورت قابل تفسیر است که روی اعداد صحیح توابعی وجود دارند که با هیچ الگوریتمی قابل بیان نیستند – در واقع قابل محاسبه نیستند. این مساله آلن تورینگ[۳] (۱۹۱۲-۱۹۵۴) را بر آن داشت تا دقیقا مشخص کند که چه توابعی محاسبه‌پذیراند. این تصور در واقع اندکی مشکل‌زا و گیج‌کننده است. زیرا مفهوم محاسبه یا رویه موثر را نمی‌توان به طور رسمی تعریف کرد. با این وجود پایان نامه (چرچ) – تورینگ، که می‌گوید ماشین تورینگ (۱۹۳۶) قادر به محاسبه هر تابع قابل محاسبه است، به عنوان یک تعریف کارآمد پذیرفته شد. همچنین تورینگ نشان داد که توابعی وجود دارند که هیچ ماشین تورینگی نمی‌تواند آنها را محاسبه کند. به عنوان مثال هیچ ماشینی وجود ندارد که در حالت کلی بگوید یک برنامه به یک ورودی خاص، خروجی می‌دهد (پاسخ می‌دهد) یا برای همیشه در حلقه نامتناهی می‌افتد.

اگر چه تصمیم‌ناپذیری و غیر قابل محاسبه بودن برای درک محاسباتی ضروری‌اند، ولی مفهوم انجام‌ناپذیری تاثیر بیشتری در محاسبات دارد . (بطور نادقیق) یک مسئله انجام‌ناپذیر است اگر زمانی که برای حل کردن نمونه‌هایی ازآن احتیاج داریم، نسبت به سایز نمونه‌ها (حداقل) بصورت نمایی رشد کند. در اواسط دهه ۷۰ بود که به تفاوت پیچیدگی بین رشد نمایی و چند جمله‌ای اهمیت داده شد (کوبهام[۴] ۱۹۶۴؛ ادموندز[۵] ۱۹۶۵). این موضوع از این جهت حائز اهمیت است که در رشد نمایی حتی نمونه‌هایی با سایز نه‌چندان بزرگ نیز در زمان معقول قابل حل نیستند. بنابراین باید مساله کلی تولید رفتار هوشمند را به یک سری زیر مساله‌های انجام‌پذیر شکست.

چگونه می‌توان برنامه‌های انجام‌ناپذیر را تشخیص داد؟ تئوری NP- کامل که استیون کوک[۶] (۱۹۷۱) و ریچارد کاپ (۱۹۷۲) از پیشگامان آن بودند، راه حلی برای این مسئله است. کوک و کارپ[۷] نشان دادند که کلاس بزرگی از مسائل استدلالی و جستجوهای ترکیبی استاندارد وجود دارند که‌NP-کامل هستند (احتمالا انجام‌ناپذیراند). اگر چه تا به حال اثبات نشده است که مسائل NP-کامل ضرورتا انجام‌ناپذیر هستند ولی (بیشتر) تئوری‌پردازان به این مطلب اعتقاد دارند. با وجود سرعت رو به افزایش کامپیوترها، استفاده دقیق و درست از منابع یکی از معیارهای توصیف سیستم‌های هوشمند است(زیرا جهان مساله‌ای بی نهایت بزرگ است).

علاوه بر منطق و محاسبات سهم سومی که ریاضیات در هوش مصنوعی دارد، تئوری احتمالات است. اولین شخصی که احتمالات را مطرح کرد، گرولامو گاچادرنوی[۸] ایتالیایی(۱۵۰۱-۱۵۷۶) بود که احتمال را در قالب

۱-Kurt Godel
۲-Frega and Russel
۳-Alan Turing
۴- Cobham
۵-Edmonds
۶-Steven Cook
۷-Richard karp
۸-Gerolamo Cardano

نتایج ممکن بازی قمار مطرح کرد. تا قبل از آن به نتایج بازی قمار به عنوان اراده الهه‌ها نگریسته می‌شد و نه به عنوان شانس. احتمالات به سرعت به بخش فوق‌العاده با ارزشی از تمام علوم کمّی تبدیل شدند ودر مواجهه با اندازه گیری‌های غیرقطعی و تئوری‌های ناکامل کمک فراوانی نمودند. پیر فرمت[۱] (۱۶۰۱-۱۶۶۵)، بلیس پاسکال[۲] (۱۶۲۳-۱۶۶۲)، جیمز برنولی[۳] (۱۶۵۴-۱۷۰۵)، پیر لاپلاس[۴] (۱۷۴۹-۱۸۲۷) و دیگران تئوری احتمالات را گسترش دادند و راه‌حل‌های آماری جدیدی را ارائه نمودند. توماس بیز[۵] (۱۷۰۲-۱۷۶۱) قانونی را برای بروزرسانی احتمالات بر اساس اطلاعات و مدارک جدید ارائه نمود. قانون بیز و زمینه‌های منتج از آن که تحلیل بیزی نامیده می‌شود، پایه‌های جدیدترین روش‌ها برای استدلال غیرقطعی در سیستم‌های هوش مصنوعی را تشکیل می‌دهند.

۱-۳-۳- اقتصاد (سال ۱۷۷۶ میلادی تاکنون)

- چه تصمیماتی برای بیشینه ساختن بهره‌وری باید بگیریم؟
- در صورت عدم همراهی دیگران، چگونه باید این کار را انجام دهیم؟
- اگر زمان دستیابی به بهره مورد نظر در آینده نزدیک نباشد، چه باید کرد؟

علم اقتصاد در سال ۱۷۷۶ میلادی آغاز شد. که در آن هنگام فیلسوف معروف اسکاتلندی آدام اسمیت[۶] (۱۷۲۳تا ۱۷۹۰) کتاب" تحقیقی درباره ماهیت و دلایل ثروت ملت‌ها" را منتشر کرد. زمانی که یونانیان باستان و دیگران به تفکر اقتصادی پرداختند، اسمیت اولین کسی بود که با آن به عنوان یک علم برخورد کرد (با استفاده از این ایده که سیستم‌های اقتصادی از عامل‌های منفردی تشکیل شده‌اند که در صدد ماکزیمم کردن سود اقتصادی خود هستند). اکثر مردم تصور می‌کنند که اقتصاد در ارتباط با پول است، اما اقتصاددان‌ها می‌گویند که آنها درباره چگونگی رسیدن مردم به نتیجه دلخواهشان (از طریق انتخاب‌هایشان) تحقیق می‌کنند. رفتار ریاضی" نتایج دلخواه" و یا "بهره‌وری" اولین بار توسط لئون والراس[۷] (۱۸۳۴ تا ۱۹۱۰) فرمول‌بندی و توسط فرانک رمزی[۸] (۱۹۳۱) و در ادامه توسط جان وان نئومان[۹] و اسکار مورگنسترن[۱۰] در کتابشان " نظریه بازی‌ها و رفتار اقتصادی" (۱۹۴۴) اصلاح شد.

نظریه تصمیم‌گیری که نظریه احتمال و نظریه بهره‌وری را ترکیب می‌کند، یک چارچوب کامل و رسمی برای تصمیمات (اقتصادی و غیر اقتصادی) در شرایط عدم قطعیت ارائه می‌کند. اما این نظریه برای سیستم‌های اقتصادی بزرگ (که در آن‌ها عامل‌ها نیازی به توجه کردن به اقدامات عوامل دیگر ندارند) مناسب است. برای سیستم‌های اقتصادی کوچک شرایط بیشتر شبیه یک بازی است. اقدامات یک بازیکن به میزان قابل توجهی

۱-Pierre Fermat
۲-Blaise Passcal
۳-James Bernoulli
۴-Pierre Laplace
۵-Tomas Bayes
۶-Adam Smith
۷-Leon Walras
۸-Frank Ramsey
۹-John von Neumann
۱۰-Oskar Morganstern

بهره‌وری دیگری را (به صورت مثبت و منفی) تحت تاثیر قرار می‌دهد. وان نئومان و مورگنسترن در ادامه تحقیقات خود در زمینه نظریه بازی‌ها به نتیجه غافلگیرکننده‌ای دست یافتند که براساس آن در برخی از بازی‌ها یک عامل عقلایی باید به صورت تصادفی (یا حداقل به شیوه‌ای که به نظر رقیبان تصادفی بیاید) عمل کند.

اکثر اقتصاددان‌ها به سومین سوال مطرح شده در ابتدای این قسمت چندان توجه نکرده‌اند. "چگونه تصمیمات عقلایی بگیریم زمانی که رسیدن به بهره‌وری (حاصل از کنش‌ها) فوری نیست و در اثر یک رشته اقدامات متوالی حاصل می‌شود؟" این مبحث در حیطه پژوهش‌های عملیاتی پیگیری شد و در جنگ جهانی دوم در جریان تلاش‌هایی در کشور انگلستان برای بهینه‌سازی تاسیسات رادار، شناخته شد و بعدها درخواست‌هایی داخلی در زمینه تصمیم‌گیری‌های مدیریتی پیچیده برای آن به وجود آمد. ریچارد بلمن[۱] (۱۹۵۷) دسته‌ای از مسائل متوالی تصمیم‌گیری با نام فرآیندهای تصمیم‌گیری" مارکوف" را ارائه کرد.

اقداماتی که در پژوهش‌های عملیاتی و اقتصادی انجام گرفته‌اند، بیشتر حول مفهوم عامل‌های عقلایی شکل گرفته‌اند، با این حال در سال‌های زیادی پژوهش‌های AI در مسیرهای کاملا جداگانه‌ای توسعه یافته‌اند. که یکی از دلایل آن، پیچیدگی آشکار در اتخاذ تصمیم‌های عقلایی بود. هربرت سیمون[۲] (۱۹۱۶ تا ۲۰۰۱) از پژوهشگران پیشگام در زمینه AI جایزه نوبل اقتصاد را در۱۹۷۸ برای تلاش‌هایش در جهت معرفی مدل‌های مبتنی بر "رضایت بخش بودن" (اتخاذ تصمیم‌هایی که به اندازه کافی خوب هستند به جای محاسبات دشوار و پرزحمت برای یافتن بهترین تصمیم) به عنوان طرح بهتری از رفتار واقعی بشر دریافت کرد. (سیمون، ۱۹۴۷) دردهه‌ی آخر قرن بیستم علاقه مجددی برای استفاده از (فنون تصمیم‌گیری نظری[۳]) در سیستم‌های عامل به وجود آمد. (ولمن[۴]، ۱۹۹۵)

۱-۳-۴- عصب شناسی (سال ۱۸۶۱ میلادی تا کنون)

- مغز چگونه اطلاعات را پردازش می‌کند؟

عصب‌شناسی علم مطالعه سیستم عصبی به ویژه مغز است. این که چگونه مغز قادر به فکر کردن است، یکی از بزرگترین اسرار علم است. هزاران سال است که تصور می‌شود مغز در فکر کردن نقش موثری دارد (به این دلیل که ضربات شدید به سر می‌تواند منجر به ناتوانی‌های ذهنی شود). هم چنین می‌دانیم که مغز بشر به گونه‌ای متفاوت است. در حدود ۳۳۵ سال قبل از میلاد مسیح ارسطو نوشت :"در میان همه حیوانات انسان به نسبت جثه‌اش بزرگترین مغز را داراست."[۵] در اواسط قرن نوزدهم بود که مغز به عنوان مرکز هوشیاری شناخته شد. پیش از آن تصور می‌شد که این مرکز در قلب، طحال، یا غده هیپوفیز قرار دارد.

مطالعات پل بروکا[۶] (۱۸۲۴ تا ۱۸۸۰) درمورد عدم قدرت تکلم در بیماران آسیب دیده مغزی در سال ۱۸۶۱ عصب‌شناسی را احیا کرد.

۱-Richard Bellman

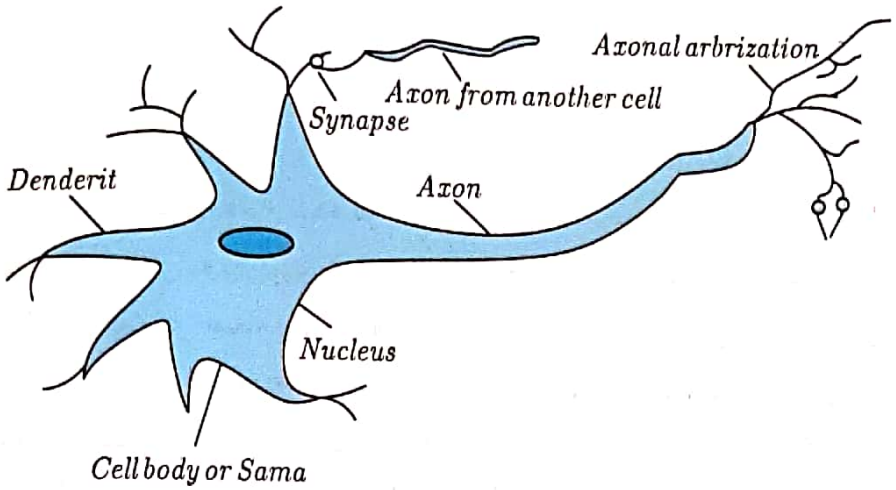
۲-Herbert Simon

۳- Decision- theoretic techniques

۴-Wellman

۵- تاکنون مشخص شده است که برخی از گونه‌های دلفین‌ها و نهنگ‌ها مغزهای بزرگتری دارند. امروزه تصور می‌شود که بزرگی مغز انسان تا حدی به دلیل تکامل‌هایی است که اخیرا در سیستم خنک کننده آن صورت گرفته است.

۶-Paul Broka



شکل ۱-۲ بخشی از سلول عصبی یا نرون. هر نرون شامل بدنه سلول، یا سوما (که شامل یک هسته سلول است) می‌شود. تعدادی فیبر از بدنه سلول خارج می‌شوند که دندرایت نامیده می‌شوند و فیبر با طول بیشتر اکسن نامیده می‌شود. اکسن تا فاصله زیادی گسترش می‌یابد (در دامنه بسیار گسترده‌تری از چیزی که در شکل نشان داده شده‌است). معمولاً طولشان یک سانتیمتر است (۱۰ برابر قطر بدنه سلول) اما می‌توانند تا یک متر گسترش یابند. یک عصب می‌تواند با ۱۰ الی ۱۰۰,۰۰۰ عصب دیگر در محلی که سیناپس نامیده می‌شود، ارتباط برقرار کند. سیگنالها از یک عصب به عصب دیگر به وسیله واکنشهای الکتروشیمیایی گسترش می‌یابند. سیگنالها فعالیت مغز را در یک زمان کوتاه کنترل می‌کنند و نیز تغییر طولانی مدت در موقعیت و اتصال عصب‌ها ایجاد می‌کنند. این مکانیزم‌ها پایه‌ای برای یادگیری در مغز ایجاد می‌کنند. بیشتر بررسی اطلاعات در لایه خارجی مغز شکل می‌گیرد. واحد مدیریت پایه یک ستون پوست با قطر حدود ۰.۵ میلیمتر است که عمق کامل کرتکس را که در انسان‌ها حدود ۴میلیمتر است گسترش می‌دهد. یک ستون شامل حدود ۲۰,۰۰۰ عصب است.

او علم پزشکی را در مورد وجود نواحی متمرکزی در مغز که مسئول اعمال ادراکی مخصوصی هستند، متقاعد کرد. به ویژه او نشان داد که تولید سخن در بخشی از نیمکره چپ مغز که امروزه ناحیه بروکا نامیده می‌شود، صورت می‌گیرد. در آن زمان می‌دانستند که مغز از سلول‌های عصبی یا نرون‌ها تشکیل شده است. اما در سال ۱۸۷۳ بود که کاملو گولگی[۱] (۱۸۴۳ تا ۱۹۲۶) یک تکنیک بافت‌شناسی را توسعه داد که مشاهده نرونهای منفرد در مغز را امکانپذیر می‌ساخت.(شکل ۱-۲ را ببینید.) این تکنیک توسط سانتیاگو رومن کاژال[۲] (۱۸۵۲ تا ۱۹۳۴) در مطالعات پیشگامانه‌اش در مورد ساختار عصبی مغز استفاده شد.

ما نقشه‌هایی عصبی بین بخش‌هایی از مغز و قسمت‌هایی از بدن که توسط این بخش‌ها کنترل می‌شوند یا ورودی‌های حسی را در اختیار آنها قرار می‌دهند، در دست داریم. این نقشه‌ها می‌توانند به طور عمده در طول چند هفته تغییر یابند. همچنین به نظر می‌رسد که برخی از حیوانات نقشه‌های عصبی چندگانه‌ای دارند. به علاوه نمی‌توانیم به طور کامل درک کنیم که چگونه وقتی که یک بخش آسیب می‌بیند، سایر بخش‌ها به خوبی به فعالیت خود ادامه می‌دهند. هیچ تئوری در مورد چگونگی عملکرد یک حافظه منفرد وجود ندارد.

۱-Calillo Golgi
۲-Santiogo Ramon Cajal

سنجش فعالیت مغز سالم با ابداع موج‌نگار مغزی (EEG) درسال ۱۹۲۹ توسط هانس برگر[۱] آغاز شد. توسعه MRI به صورت کاربردی تصاویر دقیق کم‌نظیری از فعالیت مغز در اختیار عصب‌شناسان قرار داد و سنجش‌های متناظر با فرآیندهای ادراکی درحال انجام را ممکن ساخت. این شیوه‌ها با پیشرفت‌هایی در زمینه ثبت فعالیت-های نرون توسط تک سلول تکمیل شد. با وجود این پیشرفت‌ها هنوز فاصله بسیاری با درک چگونگی فرآیندهای ادراکی داریم.

یک برداشت بسیار جالب و شگفت آور این است که: مجموعه‌ای از سلول‌های ساده می‌توانند منجر به تفکر، عمل و هوشیاری شوند. به عبارت دیگر" مغز ذهن را می‌سازد." تنها نظریه پیشنهادی واقعی "عرفان" است که می-گوید یک قلمرو عرفانی که ذهن در آن عمل می‌کند وجود دارد که ورای علم فیزیکی است.

مغز و کامپیوترهای دیجیتال وظایف کاملا متفاوتی را انجام می‌دهند و ویژگی‌های متفاوتی دارند. شکل ۱-۳ نشان می‌دهد که تعداد نرون‌ها در مغز یک انسان معمولی هزار برابر ورودی gateها درCPU یک کامپیوتر معمولی high-end است. قانون مور[۲] پیش بینی می‌کند که شمار ورودی‌های CPU در سال ۲۰۲۰ با شمار نرون‌های مغزی برابری خواهد کرد. البته چنین پیش‌بینی‌هایی چندان قابل اطمینان نیست. به علاوه اختلاف موجود در ظرفیت ذخیره‌سازی (بین انسان و کامپیوتر) بسیار کمتر از اختلاف موجود در سرعت سوئیچینگ و ظرفیت توازی است. تراشه‌های کامپیوتری می‌توانند یک دستورالعمل را در یک نانوثانیه اجرا کنند. اما نرونها میلیون‌ها بار آهسته‌ترند. اما مغز این مسئله را به شکل دیگری جبران می‌کند. در سیستم عصبی تمام نرون‌ها و سیناپس‌ها به طور همزمان فعالند در حالی که اکثر کامپیوترهای کنونی تنها یک و یا تعداد کمی CPU دارند. بنابراین حتی اگر کامپیوتر در سرعت اولیه میلیونها بار سریعتر باشد، باز هم مغز (به دلیل اجرای موازی) نهایتا صد هزار برابر سریعتر عمل می‌کند.

	کامپیوتر	مغز انسان
واحدهای محاسباتی	۱ CPU, 10^{8} gates	10^{11} عصب
واحدهای ذخیره‌سازی	10^{10} بیت RAM 10^{11} بیت دیسک	10^{11} عصب 10^{14} سیناپس
مدت چرخه	10^{-9} ثانیه	10^{-3} ثانیه
پهنای باند	10^{10} بیت بر ثانیه	10^{14} بیت برثانیه
بروزرسانی حافظه در واحد ثانیه	10^{9}	10^{14}

شکل ۱-۳ یک مقایسه تقریبی از منابع محاسباتی خام مغز و کامپیوتر. ارقام مربوط به کامپیوتر از زمان اولین انتشار این کتاب همگی حداقل ۱۰ برابر افزایش یافته‌اند و انتظار می‌رود که این روند در این دهه نیز تکرار شود. ارقام مربوط به مغز طی صدهزار سال گذشته تغییری نکرده است.

---

[۱]-Hans Berger

[۲]- قانون مور می‌گوید که شمار ترانزیستورها در هر اینچ مربع در هر ۱ تا ۱/۵ سال دو برابر می‌شود. در حالیکه ظرفیت مغز انسان به سختی در هر ۲ تا ۴ میلیون سال دو برابر می‌شود.

۱-۳-۵- روانشناسی

- انسان‌ها و حیوانات چگونه فکر و عمل می‌کنند؟

می‌توان گفت که روانشناسی علمی، با تلاشهای فیزیکدان آلمانی هرمن فون هلمولتز[۱] (۱۸۲۱-۱۸۹۴) و شاگردش ویلهلم وونت[۲] (۱۸۳۲-۱۹۲۰) آغاز شد. هلمولتز روشهای علمی را برای مطالعه بینایی انسان بکار برد و کتاب روشنایی فیزیولوژی او هنوز هم به عنوان "تنها رساله مهم" در زمینه فیزیک و فیزیولوژی دید انسان توصیف می‌شود. در سال ۱۸۷۹، وونت اولین آزمایشگاه روانشناسی تجربی را در دانشگاه لیپزیگ تاسیس نمود. وونت تاکید زیادی بر آزمایشات به دقت کنترل شده داشت که در آن کارکنان وی در فرآیندهای فکری خود فرورفته و یک کار گروهی و یا ادراکی انجام می‌دادند. کنترل دقیق راه درازی برای تبدیل روانشناسی به یک علم پیمود. در مقابل بیولوژیست‌ها در مطالعه رفتار حیوانی با کمبود داده‌های درون‌گرایانه مواجه شدند و یک نوع روش‌شناسی بیرونی را گسترش دادند. همان چیزی که توسط ه.س. جنینگ[۳] (۱۹۰۶) در اثر مهم و پر تاثیرش با نام "رفتار ارگانیسم‌های پست تر" توصیف شد.

مکتب رفتارگرایی به رهبری جان واتسون[۴] (۱۸۷۸-۱۹۵۸) این تئوری در مورد انسان را توسعه داده که در آن هر تئوری شامل فرآیندهای ذهنی را که درون گرایی نمی‌توانست شواهد قابل اعتمادی برای آن فراهم کند، را رد می‌کرد. رفتارگرایان تنها به مطالعه معیارهای عینی ادراک (انگیزه یا محرک) که به یک حیوان داده می‌شد و واکنش آن (پاسخ) اصرار می‌ورزیدند. ساختارهای ذهنی نظیر دانش، اعتقادات، اهداف و مراحل استدلال به عنوان" روانشناسی مردمی" و غیر علمی تلقی می‌شدند. رفتارگرایی به کشفیات بسیاری در مورد موشها و پنگوئن‌ها دست یافت اما در مطالعات خود برای درک انسان چندان موفق نبود. با این وجود رفتارگرایی تاثیر زیادی روی روانشناسی در سالهای ۱۹۲۰ تا ۱۹۶۰ داشت. (بخصوص در امریکا)

این دیدگاه در مورد مغز به عنوان یک وسیله پردازش اطلاعات، که یک ویژگی اصلی علم روانشناختی ادراکی است به تلاشهای ویلیام جیمز[۵] (۱۸۴۲-۱۹۱۰) باز می‌گردد. همچنین هلمولتز[۶] اعتقاد داشت که ادراک شامل یک فرم از استنباط منطقی ناخودآگاهانه است. در امریکا دیدگاه ادراکی تحت‌الشعاع رفتارگرایی قرار داشت اما در واحد روانشناختی کاربردی کمبریج که توسط فردریک بارتلت[۷] (۱۸۸۶ تا ۱۹۶۹)اداره می‌شد، مدل ادراکی پیشرفت قابل توجهی کرد. "طبیعت تفسیر" نوشته شاگرد و جانشین بارتلت، کنس کریک[۸] (۱۹۴۳) بر درستی اصطلاحات ذهنی همچون "عقاید" و "اهداف" تاکید کرد. وی اذعان نمود که استفاده از این اصطلاحات به همان اندازه علمی است که از ترم‌های فشار و دما در صحبت کردن در مورد گازها استفاده کنیم، در حالی که گازها از مولکولهایی تشکیل شده‌اند که هیچ یک فشار و دما ندارند. کریک سه مرحله کلیدی عامل "مبتنی بر

۱- Hermann von Helmholtz

۲-Wilhelm Wundt

۳-H.S.jennings

۴-John Watson

۵- ویلیام جیمز برادر رمان نویس مشهور، هنری جیمز می باشد. گفته شده است که هنری بصورتی داستان می‌نوشت که گویی در مورد روانشناسی می‌نویسد و ویلیام طوری مطالب روانشناسی را می‌نوشت که گویی در حال نوشتن داستان است.

۶-Helmholtz

۷-Frederic Bartlett

۸-Kenneth Craic

دانش " را مشخص نمود : (۱) محرک‌ها باید به یک نمایش داخلی تبدیل شوند (۲) توسط فرآیندهای ادراکی این نمایش برای تولید نمایشهای داخلی جدید تغییر یابند (۳) و متعاقبا این نمایشها دوباره به عمل ترجمه - شوند. او بطور روشن تشریح نمود که چرا این طرح خوبی برای یک عامل است:

اگر یک ارگانیسم یک مدل با مقیاس کوچک از واقعیت خارجی و مجموعه اعمال امکانپذیر خود را در سر داشته باشد، قادر خواهد بود گزینه‌های گوناگون را آزمایش کند، نتیجه بگیرد که کدام یک بهترین است و نسبت به وضعیتهای آینده (قبل از این که پیش بیایند) واکنش نشان دهد. از تجربیات اتفاقات گذشته در برخورد با حال و آینده استفاده کند و همچنین در مواجهه با موقعیت‌های خاص و غیرمنتظره از امن‌ترین و کامل‌ترین و مناسب-ترین شیوه‌ها استفاده نماید.(کریک ۱۹۴۳)

پس از مرگ کریک در یک حادثه دوچرخه‌سواری در سال ۱۹۴۵ کارهای وی توسط دونالد برد بنت[۱] که کتاب وی "ادراک و ارتباط" (۱۹۵۸) شامل برخی از اولین مدل‌های "پردازش اطلاعات" برای پدیده‌های روانشناختی است ادامه یافت. همزمان در آمریکا رشد مدلسازی کامپیوتری سبب ایجاد زمینه‌ای از علم ادراکی شد. می‌توان گفت که این زمینه در کارگاهی در سپتامبر ۱۹۵۶ در MIT آغاز شد. (باید دانست که این درست دو ماه پس از کنفرانسی بود که AI در آن متولد شد.) در این کارگاه جرج میلر[۲] "عدد هفت جادویی" و نوام چامسکی[۳]"سه مدل برای زبان" و الن ناول و هربرت سیمون[۴] "ماشین تئوری منطقی" را ارائه کردند. این مقالات قوی و تاثیرگذار نشان دادند که چگونه مدل‌های کامپیوتری می‌توانند به ترتیب برای بحث درباره روانشناختی حافظه، زبان و تفکر منطقی به کار روند. امروزه روانشناسان بر سر این موضوع که "یک تئوری ادراکی باید مانند یک برنامه کامپیوتری باشد."(اندرسون ۱۹۸۰) توافق دارند، یعنی این تئوری باید یک مکانیسم دقیق پردازش اطلاعات را توصیف کند که توسط آن برخی اعمال و توابع ادراکی قابل اجرا و پیاده‌سازی باشند.

۱-۳-۶- مهندسی کامپیوتر (سال ۱۹۴۰ میلادی تا کنون)

- چگونه می‌توانیم یک کامپیوتر کارآمد بسازیم؟

برای موفقیت هوش مصنوعی ما به دو چیز احتیاج داریم: هوش و محصول مصنوعی. کامپیوتر یک گزینه به عنوان محصول مصنوعی است. کامپیوترهای الکترونیک دیجیتال مدرن بطور مستقل و تقریباً همزمان توسط دانشمندان سه کشور درگیر در جنگ جهانی دوم ابداع شدند. اولین کامپیوتر عملیاتی، مدل الکترومکانیکی هس رابینسون[۵] نام داشت که در سال ۱۹۴۰ توسط تیم آلن تورینگ برای یک هدف مشخص یعنی رمزگشایی پیغامهای آلمانی‌ها ساخته شد. در ۱۹۴۳ همان گروه، کلوسوس ماشین همه منظوره قدرتمندی که مبتنی بر لامپ‌های خلا بود را ساخت.[۶] اولین کامپیوتر عملیاتی قابل برنامه ریزی Z-۳ نام داشت که توسط کونراد زوسه[۱]

---

[۱]-Donald Broad Bent

[۲]-George Miller

[۳]-Noam chomsky

[۴]-Allen Newell and Herbert Simon

[۵]-Heath Robinson هس رابینسون کاریکاتوریستی بود که شهرتش به خاطر ابتکارات مضحک و عجیبش برای انجام کارهای روزمره مانند کره مالیدن روی نان بود.

[۶]- در دورا ن پس از جنگ تورینگ می‌خواست از این کامپیوترها (برای تحقیقات AI برای مثال یکی ازاولین برنامه‌های شطرنج(Turing et al., ۱۹۵۳) استفاده کند. ولی تلاش‌های وی با ممانعت دولت بریتانیا مواجه شد.

در سال ۱۹۴۱ و در کشور آلمان ابداع شد . زوسه اعداد ممیز شناور و "پلن کال کیل[۲]"، اولین زبان برنامه نویسی سطح بالا را ابداع نمودند. اولین کامپیوتر الکترونیکی بنام "ای بی سی" توسط جان اتاناسف[۳] و یکی از دانشجویان او بنام کلیفورد بری[۴] بین سالهای ۱۹۴۰ تا ۱۹۴۲ در دانشگاه یووا استیت[۵] ساخته شد .
پژوهش‌های آتاناسف کمتر شناخته و حمایت شدند و این "انیاک[۶]" بود که در دانشگاه پنسیلوانیا[۷] در قالب بخشی از تحقیقات سری ارتش توسط تیمی شامل جان ماکلی[۸] و جان اکرت[۹] ساخته شد و ثابت کرد که پیشروترین در صنعت کامپیوترهای مدرن است.
در طی نیم قرن پس از آن هر نسل جدیدی از سخت افزارهای کامپیوتری به سرعت و ظرفیت بالاتر و قیمت پایین‌تری دست یافته‌اند. قدرت عملکرد تقریبا هر ۱۸ ماه دو برابر می‌شود. با ادامه این روند در یک یا دو دهه آینده به مهندسي مولکولی وتکنولوژی‌های جدیدتر احتیاج خواهیم داشت.
البته قبل از کامپیوتر الکترونیک، وسایل محاسباتی دیگری نیز وجود داشتند. اولین ماشین قابل برنامه‌ریزی یک دستگاه بافندگی بود که در سال ۱۸۰۵ توسط جوزف ماری جاکوارد[۱۰] (۱۷۵۲ تا ۱۸۳۴) ساخته شد که از کارت‌های سوراخ شده برای ذخیره الگوها و طرح‌های بافندگی استفاده می‌کرد. در اواسط قرن ۱۹ چارلز ببیج[۱۱] (۱۷۹۲ تا ۱۸۷۱) دو ماشین طراحی کرد که هیچ یک را کامل نکرد. "ماشین تفاضلی " قرار بود جدول‌های ریاضیاتی برای مهندسی و پروژه‌های علمی را حل کند. این ماشین سرانجام در ۱۹۹۱ در موزه علوم لندن (سوید[۱۲] ۱۹۹۳) ساخته شد و کار کرد. ماشین تحلیلی ببیج بسیار موفق بود. این ماشین شامل حافظه قابل آدرس‌دهی، برنامه‌های ذخیره شده وجهش‌های شرطی بود و اولین محصول مصنوعی قادر به انجام محاسبات جامع بود. همکار ببیج، ایدا لاولیس[۱۳]، دختر لرد بارون[۱۴] شاعر ، اولین برنامه‌نویس جهان بود (زبان برنامه نویسی ایدا به افتخار او نام گذاری شده است.) او برنامه‌هایی برای ماشین تحلیلی ناتمام نوشت و حتی اعتقاد داشت که ماشین می‌تواند شطرنج بازی کند یا موسیقی بسازد.

۱-Konrad ZSusr
۲-Plan kalkül
۳-John Atanasoff
۴-Clifford Berry
۵-Iowa State University
۶-ENIAC
۷- Pennsylvonia
۸-John mauchly
۹-John Eckert
۱۰-Joseph Marie Jacquard
۱۱-Charles Babbage
۱۲-Swade
۱۳-Ada Lovelace
۱۴-Lord Byron

۱-۳-۷- نظریه کنترل و فرمانشناسی (سال ۱۹۴۸ میلادی تا کنون)

- چگونه یک ماشین تحت کنترل خودش عمل می‌کند؟

Ktesibios of Alexandria (۲۵۰ سال پیش از میلاد مسیح) اولین ماشین خود کنترل‌کننده را ساخت. یک ساعت آبی که توسط یک تنظیم‌کننده، سرعت جریان آب را ثابت نگه می‌داشت. این اختراع موجب تغییر انتظارات از یک محصول مصنوعی گردید(سابقا تنها موجودات زنده می‌توانستند رفتار خود در پاسخ به تغییرات محیط تنظیم کنند). نمونه‌های دیگر از چنین ماشین‌هایی شامل ماشین بخار ساخته جیمز وات (۱۷۳۶ تا ۱۸۱۹) و ترموستات ساخته کورنلیس دربل[۱] (۱۵۷۲ تا۱۶۳۳) هستند. تئوری ریاضی سیستم‌های بازخورد پایدار در قرن ۱۹ توسعه یافت.

نوربرت وینر[۲] (۱۸۹۴ تا ۱۹۶۴) نقش اصلی را در بوجود آمدن نظریه کنترل ایفا کرد. وی یک ریاضیدان برجسته بود که با برتراند راسل[۳] کار می‌کرد. و سپس به سیستم‌های کنترل مکانیکی و زیستی و رابطه آنها با ادراک علاقه‌مند شد(همانند کریک که وی نیز از سیستم‌های کنترل به عنوان مدل‌های روانشناختی استفاده کرد.) وینر و همکارانش آرتورو روزنبلوس[۴] و جولیان بیگلو[۵] با رفتار گرایی به تقابل برخاستند. آنها رفتار هدفمند را برخاسته از یک مکانیسم تنظیم می‌دانستند که تلاش می‌کند "خطا" را به حداقل برساند(تفاوت بین حالت کنونی و حالت نهایی). در اواخر دهه پنجاه، وینر به همراه وارن مک کولچ[۶] والتر پیتز[۷] و جان وان نئومان[۸] مجموعه کنفرانس‌هایی را ترتیب داده و مدل‌های جدید محاسباتی و ریاضی ادراک را مورد بررسی قرار دادند. کتاب وینر "فرمانشناسی" (۱۹۴۸) پر تیراژترین کتاب شد و مردم را با امکان ساخت ماشین‌های هوشمند مصنوعی آشنا کرد.

هدف نظریه کنترل مدرن به ویژه شاخه‌ای که با عنوان"کنترل تصادفی بهینه" شناخته شده، طراحی سیستم‌هایی است که کارکرد عینی آن را در طول زمان به حداکثر برساند و این تا حدودی با طراحی سیستم‌های AI که به شکل بهینه رفتار می‌کنند، تطابق دارد. با این وجود چرا AI و نظریه کنترل دو حیطه متفاوت هستند؟ به ویژه با توجه به ارتباط نزدیکی که بین بانیان آنها وجود دارد. پاسخ این سوال در ارتباط نزدیکی است که بین تکنیک‌های ریاضی آشنا برای فعالان این زمینه و مجموعه مسائل متناظری که در هر نظریه جهانی مطرح است، وجود دارد. حساب دیفرانسیل و جبر ماتریسی ابزارهای نظریه کنترل هستند که خود را در اختیار سیستم‌هایی که به وسیله مجموعه‌ای از متغیرهای پیوسته ثابت قابل توصیف‌اند، قرار می‌دهند. از این گذشته تحلیل دقیق معمولا تنها برای سیستم‌های خطی امکان پذیر است. در دهه ۶۰، AI به عنوان راهی برای فرار از محدودیت‌های ریاضی تئوری کنترل بنیان نهاده شد. ابزار استنتاج منطقی و محاسبه، پژوهشگران

۱-Cornelis Drebbel
۲-Norbert Wiener
۳-Bertrand Russel
۴-Arturo Rosenblueth
۵-Julian Bigelow
۶-Warren Mcculloch
۷-Walter Pitts
۸-John Von Neumann

AI را قادر ساخت مسائلی مانند زبان، بینایی و برنا مه‌ریزی را که کاملا خارج از قلمرو نظریه‌پردازان تئوری کنترل است مورد بحث و بررسی قرار دهند.

۱-۳-۸- زبان شناسی

- زبان چگونه با اندیشه مرتبط می‌شود؟

در سال ۱۹۵۷ بی اف اسکینر[۱] کتاب"رفتار زبانی"خود رامنتشر نمود. این کتاب بسیار جامع بود و شامل سیر یادگیری زبان از دیدگاه رفتارگرایی می‌شد که توسط بهترین کارشناس در این زمینه نوشته شده بود. اما نقدی که بر این کتاب نوشته شد به اندازه خود کتاب معروف شد و باعث از بین رفتن علاقه نسبت به رفتارگرایی شد. نویسنده این نقد نوآم چامسکی[۲] بود، (کسی که اخیرا یک کتاب در مورد تئوری خود به نام ساختارهای نحوی چاپ رسانده است). چامسکی نشان داد که تئوری رفتارگرایان سخنی از خلاقیت در زبان به میان نمی‌آورد – این تئوری نشان نمی‌دهد که چگونه یک کودک می‌تواند بفهمد و جملاتی را بسازد که تا بحال نشنیده است. تئوری چامسکی که بر پایه مدل‌های نحوی زبان شناس هندی پانینی (۳۵۰ سال قبل از میلاد مسیح) استوار است، می‌تواند این مطلب را توضیح دهد و برخلاف تئوری‌های قبلی به حد کافی رسمی بود که بتوان آن را بصورت برنامه درآورد.

زبان‌شناسی مدرن و AI تقریباً همزمان متولد شدند و با هم رشد یافتند و در یک زمینه ترکیبی به نام زبان-شناسی محاسباتی یا پردازش زبان طبیعی به هم رسیدند. به زودی مشخص شد که مشکل درک زبان به طور قابل ملاحظه‌ای پیچیده‌تراز آنی است که در ۱۹۵۷ تصور می‌شد. برای فهمیدن زبان تنها فهمیدن ساختار جملات کافی نیست بلکه باید موضوع اصلی و مفاد را نیز فهمید. اگرچه این مسئله ممکن است روشن و واضح به نظر برسد اما تا دهه ۷۰ هنوز به طور گسترده درک نشده بود. بسیاری از فعالیت‌های اولیه در زمینه نمایش دانش (بررسی در مورد اینکه چگونه باید دانش را در قالبی قرار داد که توسط کامپیوتر قابل استفاده باشد) با زبان گره خورده بود و با تحقیقات در زمینه زبان‌شناسی بدست آمده بود، که آن نیز به نوبه خود با چندین دهه کار بر روی تحلیل فلسفی زبان مرتبط بود.

۱-۴- تاریخچه هوش مصنوعی

با استفاده از اطلاعاتی که تا به حال آموخته‌ایم، آماده طرح‌ریزی توسعه هوش مصنوعی واقعی می‌باشیم.

۱-۴-۱- پیدایش هوش مصنوعی (سال ۱۹۴۳میلادی- سال ۱۹۵۶میلادی)

اولین تلاشها در زمینه هوش مصنوعی توسط وارن مک کولچ و والتر پیتز در سال ۱۹۴۳ انجام گرفت. آنها از سه منبع استفاده کردند: دانش فیزیولوژی پایه و عملکرد رشته‌های عصبی در مغز ؛ تحلیل رسمی منطق گزاره‌ای که آن را مرهون Russel و Whitehead بودند و تئوری محاسبات تورینگ. آنها مدلی مصنوعی از رشته‌های عصبی ارائه نمودند که در آن به هر عصب مشخصه خاموش یا روشن بودن داده می‌شد و همچنین هر عصب یک سوئیچ

۱- B.F. Skinner
۲-Noam Chomsky

برای روشن شدن در پاسخ به تحریک‌های عصب‌های مجاور داشت. حالت هر عصب توسط گزاره‌ای که تحریکات آن عصب را نشان می‌داد، مشخص می‌شد. به عنوان مثال آنها نشان دادند که هر تابع قابل محاسبه توسط شبکه‌ای از عصب‌های به هم پیوسته قابل محاسبه است، و تمام رابطه‌های منطقی با ساختار شبکه‌ای ساده قابل پیاده‌سازی هستند. همچنین مک کولچ و پیتز اظهار نمودند که شبکه تعریف شده به شکل مناسبی دارای قدرت یادگیری است. دونالد هب[۱] (۱۹۴۹) یک قانون به روزرسانی ساده برای تغییر قدرت ارتباط بین عصب‌ها ارائه نمود. این قانون که اکنون یادگیری هبین (Hebbian) نامیده می‌شود تا به امروز مدلی تأثیرگذار باقی مانده است.

دو دانشجوی دانشکده ریاضی پرینستون، ماروین مینسکی[۲] و دین ادموندز[۳] اولین کامپیوتر شبکه عصبی را در سال ۱۹۵۱ ساختند. در این شبکه عصبی که اسنارک[۴] نامیده می‌شد، از ۳۰۰۰ لامپ خلا و یک سیستم خلبانی خودکار (باقیمانده از یک بمب افکن بی-۲۴) برای شبیه سازی یک شبکه با ۴۰ عصب استفاده شد. کمیته دکترای مینسکی مردد بود که آیا این نوع کار باید به عنوان ریاضیات درنظر گرفته شود یا خیر، اما نقل شده است که فون نیومن در آن زمان گفت : "اگر امروز این طور نباشد یک روزی اینطور خواهد شد". بعدها مینسکی قضایای تأثیر گذاری را که نشان‌دهنده محدودیت‌های تحقیقات شبکه‌های عصبی بود، اثبات کرد.

نمونه‌هایی از کارهایی پیشین که ویژگیهای هوش مصنوعی را دارا بودند وجود داشتند ولی آلن تورینگ نخستین فردی بود که در سال ۱۹۵۰ چشم انداز کاملی را از هوش مصنوعی در مقاله خود با نام "ماشین‌های محاسباتی و هوشمندی" اظهار کرد. در این مقاله تورینگ به معرفی تست تورینگ،یادگیری ماشین،الگوریتم ژنتیک و تحکیم یادگیری پرداخت.

۱-۴-۲- تولد هوش مصنوعی (سال ۱۹۵۶ میلادی)

پرینستون محل تحصیل یکی دیگر از افراد موثر در هوش مصنوعی یعنی جان مک کارتی بود. مک کارتی بعد از اتمام تحصیل به کالج دارت موث رفت. مکانی که در حال تبدیل شدن به زادگاه رسمی هوش مصنوعی بود. مک کارتی، مینسکی، شنون و راچستر را متقاعد کرد که او را در گرد هم آوردن محققان امریکایی علاقه‌مند به تئوری اتوماتا[۵]، شبکه‌های عصبی و مطالعه در زمینه هوش مصنوعی، کمک کنند. آنها در تابستان ۱۹۵۶ کارگاهی را به مدت دو ماه در دارت موث ایجاد کردند. شرکت کننده‌ها در مجموع حدود ده نفر بودند که در بین آنها افرادی چون تنچارد مور[۶] از پرینستون، آرتور ساموئل[۷] از آی بی ام و ری اسلومونف[۸] و الیور سلفریج[۹] از ام آی تی دیده می‌شدند.

۱-Donald Hebb
۲-Marvin Minsky
۳-Dean Edmonds
۴-SNARC
۵-automata theory
۶-Trenchard More
۷-Arthur Samuel
۸-Ray Solomonoff
۹-Oliver Selfridge

دو محقق از کارنیج تک[۱] بنام‌های آلن ناول و هربرت سیمون بیشتر از همه خود را نشان دادند. اگر چه دیگران ایده‌هایی و در بعضی موارد برنامه‌هایی برای کاربردهای خاص مانند کنترل کننده‌ها داشتند، ولی ناول و سیمون برنامه‌ای بنام نظریه منطق، داشتند که سیمون در مورد آن چنینی ادعایی داشت: "ما برنامه کامپیوتری ایجاد کردیم که قادر است بصورت غیر عددی فکر کند و به وسیله آن می‌توان مساله جسمی- ذهنی را حل کرد." طولی نکشید که بعد از این کارگاه، این برنامه قادر بود بسیاری از قضایای فصل دوم کتاب راسل و وایت‌هد[۲] (ریاضیات اصولی) را اثبات کند. راسل زمانی که سیمون به او نشان داد که این برنامه اثباتی کوتاهتر از اثباتی که در کتاب او آمده است را تولید کرده، بسیار خوشحال شد.

کارگاه دارت موث منجر به پیشرفت جدیدی نشد اما افراد حاضر را با جنبه‌های مختلف هوش مصنوعی آشنا کرد. تا بیست سال بعد هوش مصنوعی زیر نظر این افراد و شاگردانشان وهمکاران آنان در ام آی تی ، سی ام یو، استنفرد و آی بی ام قرار داشت. شاید آخرین چیزی که ماحصل این کارگاه بود توافق به پذیرفتن اسم جدید این زمینه بنام هوش‌مصنوعی بود که توسط مک کارتی پیشنهاد شد. "منطق محاسباتی" می‌توانست اسم بهتری باشد ولی "AI" ماندگار شد.

با توجه به پروپزال مک کارتی برای برگزاری کارگاه دارتموث متوجه ضرورت جدا شدن AI به عنوان یک زمینه جدید میشویم. چرا همه کارهایی که در AI انجام می‌شود را نمی‌توان تحت زمینه‌هایی مثل نظریه کنترل، تحقیق در عملیات یا نظریه تصمیم‌گیری که اهدافی مشابه اهداف AI را دارند انجام داد؟ یا به عبارت دیگر چرا AI شاخه‌ای از ریاضی در نظر گرفته نمی‌شود؟ اولین پاسخ این است که AI در ابتدا دربرگیرنده ایده کپی کردن توانمندی‌های انسان‌ها مانند خلاقیت، بهبود خود و استفاده از زبان بوده است که هیچ یک از زمینه‌های دیگر به این مسائل نپرداخته بودند. پاسخ دوم متدولوژی است. AI تنها زمینه‌ای است که به وضوح شاخه‌ای از علوم کامپیوتر است (اگرچه تحقیق در عملیات نیز روی شبیه‌سازی‌های کامپیوتری تأکید دارد) و تنها زمینه‌ای است که تلاش می‌کند ماشینی بسازد که بتواند به طور مستقل در محیطهای پیچیده و متغیر عمل کند.

۱-۴-۳- شور و شوق گذشته، انتظارات بزرگ (۱۹۵۲-۱۹۶۹)

سالهای آغازین هوش مصنوعی با موفقیتهای زیاد در یک راستای محدود همراه بود. با توجه به کامپیوترهای اولیه و ابزارهای برنامه‌نویسی و این حقیقت که تا چند سال گذشته به کامپیوترها به دید وسیله‌هایی که تنها قادر به انجام محاسبات ریاضی هستند نگاه می‌شد، انجام کارهای هوشمند توسط کامپیوتر بسیار عجیب می‌نمود. بیشتر افرادی که در این زمینه کار می‌کردند ترجیح دادند بپذیرند که "ماشین هرگز نمی‌تواند ایکس را انجام دهد"(ایکس لیست بلندی از مسائل بود که توسط تورینگ جمع‌آوری شده بود). نتیجه تحقیق محققان هوش مصنوعی اثبات یک به یک این ایکسها بود.

موفقیت زودهنگام ناول و سیمون با طراحی برنامه حل‌کننده عمومی مساله یا GPS ادامه یافت. برخلاف برنامه نظریه منطق، این برنامه از ابتدا برای تقلید رفتار انسان در حل مسایل طراحی شد. ثابت شد که ترتیب در نظر گرفتن زیر اهداف و کنش‌های ممکن توسط این برنامه برای کلاس محدودی از مسایل که قادر به حل آنها بود،

---

[۱]- دانشگاه ملون کارنیج امروزی، ام سی یو

[۲]-Russel and Whitehead

مشابه روشی بود که انسان برای حل این مسائل درنظر می‌گیرد. بنابراین احتمالا GPS اولین برنامه‌ای بود که روش فکر کردن مشابه انسان را پیاده می‌کرد.

موفقیت GPS و برنامه‌های پس از آن به عنوان مدل‌های شناختی، سیمون و ناول را به سمت فرموله کردن فرضیه سیستم نمادهای فیزیکی برد. براساس این فرضیه سیستم نمادهای فیزیکی امکانات لازم و کافی برای کنش‌های عمومی هوشمند را داراست. بدین معنی که هر سیستمی (انسان یا ماشین) که اثری از هوشمندی نشان می‌دهد بایستی با دستکاری ساختارهای داده‌ای (که از نمادها تشکیل شده‌اند) کار کند. بعدها می‌بینیم که این فرضیه از بسیاری جهات به چالش کشیده شده‌است.

در آی‌بی‌ام، ناتانیل راچستر[۱] و همکاران او تعدادی از اولین برنامه‌های هوش مصنوعی را ایجاد نمودند. هلبرت گلنتر[۲] (۱۹۵۹) اثبات کننده قضایای هندسی را ساخت که بسیاری از مسائلی را که دانشجویان ریاضی در حل آن با دشواری روبرو بودند، اثبات کرد. در اوایل سال ۱۹۵۲ آرتور ساموئل[۳] یک سری برنامه برای بازی چکرز[۴] نوشت و سرانجام این برنامه یاد گرفت که چگونه چکرز را در سطح قهرمانی بازی کند. او همچنین این ایده را که کامپیوترها تنها قادرند کارهایی را انجام دهند که ما به آنها می‌گوییم، رد کرد. همانند برنامه او که یاد گرفت که چگونه بهتر از خالق خود بازی کند. این برنامه در فوریه سال ۱۹۵۶ در تلویزیون به نمایش در آمد و تاثیر زیادی نیز گذاشت. در فصل ۶ با بازی‌ها آشنا خواهیم شد و فصل ۱۱ تکنیک‌های یادگیری استفاده شده توسط ساموئل را تشریح خواهد کرد.

جان مک کارتی از دارت موث به ام‌آی‌تی رفت ودر سال تاریخی ۱۹۵۸ در سه مورد به موفقیت‌های بزرگی دست یافت. در آزمایشگاه هوش مصنوعی ام‌آی‌تی او زبان سطح بالای LISP را تعریف کرد که در حال تبدیل شدن به بزرگترین زبان برنامه‌نویسی هوش مصنوعی بود. LISP از نظر قدمت بین زبانهایی که در حال حاضر استفاده می‌شوند رتبه دوم را دارد و یک سال از فرترن جوانتر است.[۵] با استفاده از LISP مک کارتی به ابزاری که احتیاج داشت دست یافت اما دستیابی به منابع محاسباتی گران و کمیاب همچنان یک مشکل اساسی بود. بنابراین او و دیگران اشتراک زمانی را در ام‌آی‌تی ابداع نمودند. همچنین در سال ۱۹۵۸ مک کارتی مقاله‌ای با عنوان "برنامه‌هایی با عقل و درایت" منتشر کرد که در آن Advice Taker (یک برنامه فرضی که در واقع اولین سیستم کامل هوش مصنوعی به حساب می‌آید) را توضیح داد. همانند نظریه منطق و اثبات کننده قضایای هندسی ، برنامه مک کارتی به منظور بکارگیری دانش برای یافتن راه‌حل مسایل طراحی شد. اما بر خلاف بقیه ، به منظور در برگرفتن دانش جامع از جهان طراحی شد. به عنوان مثال او نشان داد که چگونه تعدادی اصول ساده این برنامه را قادر می‌سازد تا طرحی برای رفتن به فرودگاه و رسیدن به هواپیما تولید کند. این برنامه همچنین بصورتی طراحی شده بود که می‌توانست در حال انجام عملیات خود اصول جدیدی را بپذیرد و بدون احتیاج به دوباره برنامه‌ریزی شدن در زمینه‌های جدیدی کار کند. بنابراین Advice Taker اصول اساسی نمایش دانش و استدلال را در خود داشت: بهتر است که نمایش صریح و رسمی از جهان و تاثیراتی که عامل بر

۱-Nathaniel Rochester
۲-Herbert Gelernter
۳-Arthur Samuel
۴-Checkers

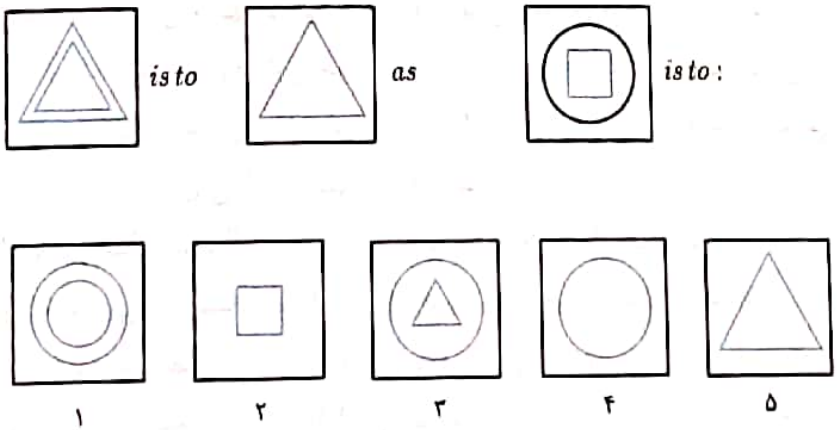
۵- فرترن یک سال زودتر از لیسپ بوجود آمد

جهان می‌گذارد در اختیار باشد و بتواند این نمایش را با استفاده از فرآیندهای استنتاجی تغییر دهد. شایان ذکر است که بخش اعظمی از مقاله‌های سال ۱۹۵۸ بعد از ۳۵ سال هنوز هم قابل استفاده می‌باشد.

سال ۱۹۵۸ سالی است که ماروین مینسکی[۱] به ام‌آی‌تی رفت. با این حال همکاری‌های ابتدایی بین او و مک کارتی دیری نپایید زیرا مک کارتی بر نمایش و استدلال با منطق رسمی تاکید داشت. در حالی که مینسکی علاقه‌مند بود که برنامه را درجهت انجام کارها به کاربگیرد و سرانجام یک دیدگاه ضد منطق را ایجاد کرد. در سال ۱۹۶۳ مک کارتی آزمایشگاه هوش مصنوعی را در استنفرد ایجاد کرد. هدف تحقیقات او که همانا استفاده از دانش در ساختن Advice Taker بود با کشف روش نتیجه‌گیری (یک الگوریتم کامل برای اثبات قضایای منطق درجه اول؛ به فصل ۶ مراجعه کنید) توسط جی ای رابینسون[۲] وارد مرحله تازه‌ای شد. کار در استنفرد بر اهمیت روشهای همه منظوره برای استدلال منطقی افزود. کاربردهای منطق موارد زیر را در بر می‌گیرد: سیستم‌های برنامه‌ریز و پاسخگوی کردل گرین[۳] (۱۹۶۹b) ، پروژه رباتیک شیکی[۴] در موسسه تحقیقاتی استنفرد (اس آر ای).

مینسکی نظارت بر تعدادی از دانشجویان را برعهده داشت که قرار بود این دانشجویان تعداد محدودی برنامه انتخاب کنند که حلشان به هوشمندی نیاز داشت. این دامنه‌های محدود میکروجهان نام گرفتند. برنامه جیمز اسلیگل[۵] (۱۹۶۳)، تحت عنوان سینت[۶]، قادر بود مسایل انتگرال حسابی فرم بسته سال اول دانشگاه را حل نماید. برنامه آنالوگی[۷] تام اوانز[۸] (۱۹۶۸) قادر بود مسایل مقایسه‌ای هندسی که در تستهای هوش کاربرد دارند را حل نماید مانند تست شکل ۱-۴. برنامه Danial Bobrow به نام دانش آموز(سال ۱۹۶۷) می‌توانست مسایل جبری نظیر آنچه که در زیر آمده است را حل کند:

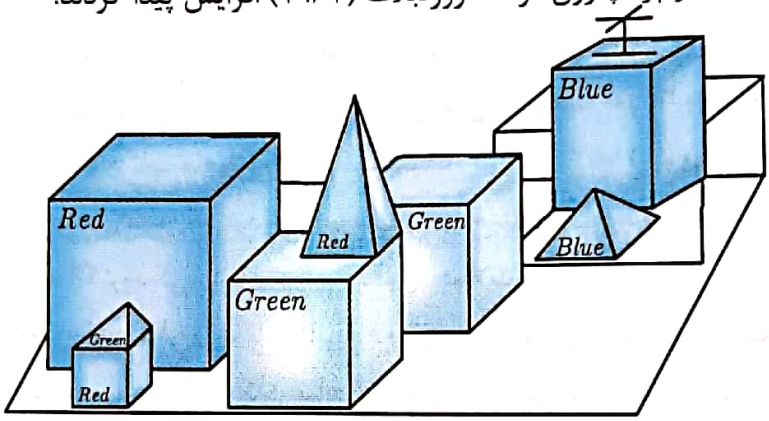
"اگر تعداد مشتریان تام دوبرابر بیست درصد تعداد آگهی‌ها به توان دو باشد که تام چاپ می‌کند و تعداد آگهی‌ها ۴۵ باشد، تام چه تعداد مشتری دارد؟"



شکل ۱-۴ یک نمونه مسئله که با برنامه شباهت یابی Evan حل شد

۱-Marvin Minsky
۲-J.A.Rabinson
۳-Cordell Green
۴-Shakey
۵-James Slogle
۶-SAINT
۷-ANALOGY
۸-Tom Evans

معروفترین ریزجهان، جهان جعبه‌ها می‌باشد که همان طور که در شکل ۱-۵ نشان داده شده است، از یک تعداد جعبه که روی یک میز (یا شبیه سازی یک میز) قرار گرفته‌اند تشکیل شده است. هدف در این جهان بدین صورت است که باید جعبه‌ها را به یک آرایش خاص در آورد و تنها وسیله ممکن دست رباتی است که در هر لحظه می‌تواند تنها یک جعبه را جابه‌جا کند. کاربرد جهان جعبه‌ای را می‌توان در پروژه بینایی دیوید هافمن[۱] (۱۹۷۱) ، کار انتشار محدودیت و بینایی دیوید والتز[۲] (۱۹۷۵)، نظریه یادگیری پاتریک وینستون[۳] (۱۹۷۰)، برنامه درک زبانهای طبیعی‌تری وینوگراد[۴] (۱۹۷۲) و برنامه ریز اسکات فالمن[۵] (۱۹۷۴) یافت.
تلاشهای گذشته مک کولچ و پیتز در زمینه شبکه‌های عصبی نیز به ثمر نشست. کارهای وینوگراد[۶] و کوان[۷] (۱۹۶۳) نشان داد که چطور تعداد زیادی از عناصر می‌توانند در کنار هم مفهوم خاصی را نشان دهند و استحکام و موازی کاری را افزایش دهند. روشهای یادگیری هب توسط برنی ویدرو (ویدرو وهاف ۱۹۶۰؛ ویدرو ۱۹۶۲)، که شبکه خود را ادالاین[۸] نامید، و پرسپترون فرانک روزنبلات (۱۹۶۲) افزایش پیدا کردند.



شکل ۱-۵ صحنه‌ای از دنیای جعبه‌ها. SHRDLU (Winograd, ۱۹۷۲) فرمان "جعبه‌ای را که از جعبه‌ای که در دست داری بلندتر است انتخاب کن و در مربع قرار بده" را تمام کرده است.

روزنبلات تئوری همگرایی پرسپترون را اثبات کرد و نشان داد که الگوریتم یادگیری او می‌تواند قدرت ارتباط یک پرسپترون را با هر داده ورودی، در صورتی که قابل تنظیم باشند، تنظیم نماید.

۱-۴-۴- طعم واقعیت (۱۹۶۶-۱۹۷۴)

از ابتدا محققان هوش مصنوعی پیش‌بینی می‌کردند که به موفقیت‌های بسیاری در این زمینه دست یابند. عبارات زیر از هربرت سیمون در سال ۱۹۵۷ نقل شده است:
هدف من غافلگیر کردن و یا شوکه کردن شما نیست، اما من می‌توانم بطور خلاصه به شما بگویم که در حال حاضر ماشینهایی در جهان وجود دارند که فکر می‌کنند، یاد می‌گیرند و خلق می‌کنند. همچنین توانایی این

---

[۱]-David Huffman
[۲]-David Waltz
[۳]-Patrick Winston
[۴]-Terry Winograd
[۵]-Scott Fahlman
[۶]-Winograd
[۷]-Cowan
[۸]-adalines

ماشینها برای انجام دادن این کارها بسرعت در حال افزایش است تا زمانی که در آینده‌ای نه چندان دور، حدود مسایلی که این ماشینها می‌توانند در بر بگیرند هم‌تراز مغز انسان خواهد شد.

اصطلاحاتی مثل آینده نزدیک بصورتهای مختلفی قابل تعبیر است ولی سیمون پیش‌بینی‌های دقیق‌تری نیز انجام داد: او پیش بینی کرد که تا ده سال آینده یک کامپیوتر، قهرمان شطرنج می‌شود و یک تئوری مهم و جدید ریاضی توسط ماشین اثبات می‌شود. البته وقوع این پیش‌بینی‌ها به جای ۱۰ سال ۴۰ سال طول کشید. ادعاهای خوشبینانه از این دست، حاصل عملکرد خوب سیستم‌های اولیه AI بر روی مثال‌های ساده بود. ولی در بیشتر زمینه‌ها این سیستم‌های اولیه در حل مسائلی که در همان سطح سادگی بوده ولی از دامنه گسترده‌تری انتخاب شده بودند، با شکست مواجهه شدند.

اولین دسته از مشکلات به این دلیل بوجود آمد که برنامه‌های قدیمی معمولا دانش کمی و در بعضی از موارد هیچ دانشی از موضوع نداشتند و تنها ازطریق عملیات‌های نحوی ساده به موفقیت می‌رسیدند. داستان معروف در این زمینه تلاشها برای ساخت ماشین مترجم است که انجمن ملی تحقیقات آمریکا به منظور سرعت بخشیدن به ترجمه مقالات علمی مربوط به پرتاب ماهواره روسی اسپاتنیک[۱] در سال ۱۹۵۷ روی این موضوع سرمایه گذاری زیادی کرد. در ابتدا به نظر می‌رسید که تبدیل‌های معنایی بر مبنای گرامرهای انگلیسی و روسی و جابجایی لغات با استفاده از لغتنامه الکترونیکی، برای ترجمه دقیق جملات کافی باشد. اما در واقعیت ترجمه نیاز به دانشی جامع در مورد موضوع موردنظر دارد تا توانایی رفع ابهامات و بیان مفاهیم را داشته باشد. در سال ۱۹۶۶ کمیته مشورتی به این نتیجه رسید که امکان ساخت ماشین مترجم متون علمی وجود ندارد. بنابراین تمام سرمایه گذاری‌های دولت امریکا برای پروژه‌های ترجمه آکادمیک لغو شد. امروزه ماشین‌های مترجم با اینکه ناکامل هستند ولی به طور گسترده در بسیاری از زمینه‌های فنی، تجاری، دولتی و اسناد اینترنتی استفاده می‌شوند.

دسته دوم انجام‌ناپذیر بودن برخی از مسائل هوش‌مصنوعی بود. بسیاری از برنامه‌های قدیمی هوش مصنوعی برای یافتن راه‌حل، تمامی ترکیبات ممکن در یک مرحله را امتحان می‌کردند. برنامه‌های اولیه بدین دلیل موفق بودند که ریزجهان‌ها تنها شامل تعداد محدودی شی بودند بنابراین تعداد کنش‌های ممکن محدود و دنباله کنش‌ها نیز کوتاه بودند. قبل از مطرح شدن تئوری پیچیدگی محاسباتی این باور وجود داشت که برای حل مسایل بزرگتر تنها به سخت افزارهای سریعتر و حافظه‌های بزرگتر نیاز داریم. این خوش‌بینی که با پیشرفت تئوری اثبات رزولوشن (Resolution) همراه بود زمانی به یاس تبدیل شد که محققان در اثبات قضایایی که در آنها حقایق زیادی وجود داشت ناکام ماندند. اینکه طبق اصول علمی یک برنامه می‌تواند یک راه حل را پیدا کند بدین معنی نیست که در عمل هم چنین اتفاقی خواهد افتاد.

ناکامی حل مشکل "انفجار ترکیبی[۲]" یکی از انتقادات اصلی در باره هوش مصنوعی بود که در گزارش لایت‌هیل[۳] (۱۹۵۹) آمده است. این گزارش باعث شد که دولت بریتانیا بجز در دو دانشگاه به حمایت خود از تحقیقات هوش مصنوعی پایان دهد. (گفته‌ها حاکی از آن است که جاه‌طلبی‌های سیاسی و خصومت‌های شخصی نیز در این زمینه دخیل بوده‌اند).

۱-Sputnic
۲- Combinatorial explosion
۳-Lighthill

دسته سوم از مشکلات به دلیل استفاده برخی محدودیتهای بنیادی در ساختارهای پایه‌ای برای تولید رفتار هوشمند به‌وجود آمدند. به عنوان مثال در سال ۱۹۶۹ کتاب مینسکی و پیپرت بنام پرسپترون[۱] ثابت کرد که اگر چه پرسپترون‌ها(فرم ساده شده‌ای از شبکه‌های عصبی) هر چیزی را که قادر به نمایش آن باشند می‌توانند یاد بگیرند ولی چیزهایی را که می‌توانند نمایش دهند بسیار محدود است. به طور خاص، یک پرسپترون دو-ورودی توانایی تشخیص دو ورودی متفاوت را ندارد.

۱-۴-۵- سیستم‌های مبتنی بر دانش: کلیدی برای رسیدن به قدرت؟ (۱۹۷۹-۱۹۶۹)

روش حل مساله‌ای که در دهه اول تحقیق در مورد هوش مصنوعی بوجود آمد، عبارت بود از یک مکانیزم جستجوی عام‌منظوره که تلاش می‌کرد روش‌های ابتدایی استدلال را برای یافتن جوابهای کامل به‌کار بگیرد. اگرچه این روش‌ها عمومی بودند اما قابل تعمیم به مسئله‌های بزرگ و پیچیده نبودند و به همین دلیل روشهای ضعیف نامیده می‌شدند. روش دیگر، استفاده از دانش قوی‌تر و تخصصی‌تر است که امکان استفاده از گام‌های بزرگتر در استدلال را فراهم می‌کند و این امکان را می‌دهد که در مواجهه با حالت‌های خاص به مشکلات کمتری برخورد کنیم. شاید بتوان اینگونه تعبیر کرد که برای حل یک مساله سخت، می‌بایست جواب مسئله را از قبل (به طور تقریبی) بدانیم.

برنامه DENDRAL (باچنن[۲] و دیگران، ۱۹۶۹) نمونه‌ای اولیه از این رویکرد بود. این برنامه در دانشگاه استنفورد نوشته شده بود، جایی که اد فاگنبام[۳] (دانشجوی هربرت سیمون)، بروس باچنن (فیلسوفی که دانشمند علوم کامپیوتر شده بود) و جاشوا لدربرگ[۴] (نسل شناس برنده جایزه نوبل) با یکدیگر تیمی تشکیل داده بودند تا مساله یافتن ساختار مولکولی از روی اطلاعات بدست آمده از طیف‌سنج جرمی را حل کنند. ورودی برنامه عبارت بود از فرمول ابتدایی مولکول (برای مثال $C_6H_{12}NO_2$) و طیف جرمی (که جرم تکه‌های گوناگون مولکول در اثر بمباران اشعه الکترونی را نشان می‌داد). برای مثال، طیف جرمی ممکن است در $m=15$ (که متناظر با جرم متیل CH_3 است) دارای یک قله (یا پیک) باشد.

نسخه ساده برنامه تمامی ساختارهای منطبق با فرمول را ایجاد و طیف متناظر با آن را پیش‌بینی و آن را با طیف واقعی مقایسه می‌کرد. همانگونه که انتظار می‌رود، برای مولکولی با اندازه استاندارد، این راه‌حل انجام‌ناپذیر است. محققین برنامه DENDRAL با تحلیل‌گران شیمی مشورت کرده و دریافتند حل این مسئله از طریق مشاهده الگوهای مشخصی از پیک‌ها در طیف (که نشان‌دهنده زیرساختاری از مولکول است) امکان‌پذیر است . برای مثال، قانون زیر برای شناسایی زیرگروه کتون[۵] (که وزنش ۲۸ است) به کار می‌رود:

اگر دو ماکسیمم در x_1 و x_2 موجود باشد به گونه‌ای که

(a) $x_1 + x_2 = M + 28$ (M جرم کل مولکول است)؛

(b) $x_1 - 28$ یک ماکسیمم بزرگ باشد؛

۱- Perceptrons
۲- Buchanan
۳- Ed Feigenbaum
۴-Joshua Lederberg
۵-Ketone (C=O)

(c) $x_2 - 28$ یک ماکسیمم بزرگ باشد؛
(d) حداقل یکی از x_1 و x_2 ماکسیمم بزرگ باشد؛
آنگاه یک زیر گروه کتون وجود دارد.
دانستن این موضوع که "مولکول شامل زیر ساختار مشخصی است" تعداد کاندیدها را به شدت کاهش می‌دهد.
DENDRAL قدرتمند بود زیرا:
تمامی دانش نظری مرتبط برای حل این مسائل از فرم عمومی آن به یک فرم ویژه و کارا نگاشته شد. (فاگنبام و دیگران، ۱۹۷۱)
اهمیت DENDRAL در این بود که اولین سیستم موفق متمرکز بر دانش بود: مهارت آن برگرفته از تعداد زیادی قوانین خاص منظوره بود. سیستم‌های بعدی هم موضوع اصلی رویکرد مک کارتی در Advice Taker یعنی جدایی کامل دانش (بصورت قواعد) و استدلال‌کننده را بکار گرفتند.
با در نظر گرفتن داستان فوق‌الذکر، فاگنبام و بقیه افراد در استنفورد پروژه برنامه‌نویسی ابتکاری (Heuristic Programming Project) را آغاز کردند (برای اینکه تحقیق کنند که تا چه حد متدولوژی جدید سیستم‌های خبره قابل استفاده در بقیه نواحی دانش انسان است). تلاش مهم دیگر، در زمینه تشخیص پزشکی بود. فاگنبام، باچنن و دکتر ادوارد شرتلیف MYCIN را برای تشخیص بیماریهای خونی به وجود آوردند. با استفاده از حدود ۴۵۰ قاعده (پزشکی) MYCIN قادر بود به خوبی متخصصین و به طور قابل ملاحظه‌ای بهتر از پزشکان تازه کار انجام وظیفه کند. MYCIN دو تفاوت با DENDRAL داشت. اول اینکه بر خلاف قوانین DENDRAL، هیچ مدل نظری عمومی وجود نداشت که قوانین MYCIN از آن‌ها استنتاج شوند. این قوانین می‌بایست از طریق مصاحبه‌های فشرده با افراد خبره (که هرکدام از آنها نیز این قوانین را از طریق مطالعه کتابهای تخصصی، یادگیری از افراد خبره دیگر یا تجربه مستقیم بدست آورده بودند) بدست می‌آمدند. دوم اینکه قوانین می‌بایست عدم قطعیت در دانش پزشکی را منعکس می‌کردند. برای این منظور MYCIN حساب عدم قطعیتی (که فاکتور قطعیت نامیده می‌شد) را بکار گرفت. این نظریه درآن زمان به خوبی منطبق بر نظر پزشکان (که مبتنی بر تاثیر شهود بر تشخیص است) بود.

اهمیت دامنه دانش در حوزه فهم زبان طبیعی نیز آشکار بود. با وجود اینکه سیستم SHRDLU وینوگراد[۱] برای فهم زبان طبیعی هیجان زیادی ایجاد نمود، ولی وابستگی آن به تحلیل نحوی باعث بوجود آمدن مشکلاتی شد که مشابه مسائل موجود در ساخت ماشین مترجم اولیه بود. این سیستم قادر بود ابهامات را رفع و ارجاعات ضمایر را بفهمد(این موفقیت بیشتر به دلیل این بود که این سیستم به طور خاص برای حوزه دنیای بلوکها طراحی شده بود). چندین محقق، از جمله ایگن کارنیاک[۲]، دانشجوی فارغ‌التحصیل وینوگراد در MIT، اظهار کردند که فهم زبان به صورت کامل نیازمند دانش عمومی در مورد جهان و یک روش عمومی برای استفاده از این دانش می‌باشد.

در ییل[۳]، راجر شنک[۱]، زبانشناسی که محقق هوش مصنوعی شده بود، بر این نکته تاکید کرد و گفت"چیزی به نام نحو (syntax) وجود ندارد". شنک و دانشجویانش چندین برنامه نوشتند که همه آنها وظیفه‌شان فهمیدن

۱-Winograd
۲-Eugene charniak
۳- Yale (دانشگاهی در آمریکا)

زبان طبیعی بود. اما تمرکز اصلی بر روی خود زبان نبود بلکه بیشتر بر روی مشکلات نمایش و استدلال بوسیله دانش مورد نیاز برای فهم زبان بود. این مشکلات شامل نشان دادن شرایط کلیشه‌ای (کالینگفورد[۲]، ۱۹۸۱)، شرح سازمان حافظه انسان(ریگر[۳]، ۱۹۷۶؛ کلدنر[۴]، ۱۹۸۳) و فهمیدن برنامه‌ها و اهداف (ویلنسکی[۵]، ۱۹۸۳) بود.
رشد گسترده برنامه‌های کاربردی برای مسائل دنیای واقعی موجب رشد همزمان نیازها برای الگوهای نمایش دانش شد. تعداد زیادی زبانهای متفاوت برای نمایش و استدلال دانش ایجاد شد. تعدادی بر مبنای منطق استوار بود - برای مثال، زبان پرولوگ که در اروپا و خانواده PLANNER که در امریکا رایج شد. بقیه، که پیرو ایده قابهای (frames) مینسکی (۱۹۷۵) بودند، رویکرد ساخت‌یافته‌تری را بکار گرفتند، واقعیات مربوط به اشیاء خاص و گونه‌های رخدادها را جمع‌آوری می‌کردند و گونه‌ها را به صورت یک ساختار طبقه بندی شده همانند طبقه بندی زیست شناسی، مرتب می‌کردند.

۱-۴-۶- هوش مصنوعی یک صنعت می‌شود (۱۹۸۸-۱۹۸۰)

اولین سیستم خبره موفقیت‌آمیز تجاری، R۱، در شرکت DEC (مک درمت[۶]، ۱۹۸۲) شروع به کار کرد. برنامه در مرتب‌سازی سفارشات مشتریان برای سیستم‌های کامپیوتری استفاده می‌شد و در سال ۱۹۸۶ در حدود ۴۰ میلیون دلار در سال برای شرکت صرفه‌جویی داشت. تا سال ۱۹۸۸ گروه هوش مصنوعی شرکت DEC، ۴۰ سیستم خبره در حال کار و تعداد بیشتری نیز آماده به کار داشتند. شرکت DU PONT، ۱۰۰ سیستم در حال کار و ۵۰۰ سیستم در حال توسعه داشت که در حدود ۱۰ میلیون دلار در سال صرفه‌جویی داشت. تقریباً هر شرکت بزرگ آمریکایی گروه هوش مصنوعی خودش را داشت و درحال تحقیق و توسعه سیستم‌های خبره خود بود.

در سال ۱۹۸۱، ژاپنی‌ها پروژه "نسل پنجم" را اعلام کردند، یک برنامه ده ساله برای ساختن رایانه‌های هوشمند که Prolog را اجرا کنند. در واکنش به این پروژه آمریکایی‌ها شرکت فناوری میکروالکترونیک و کامپیوتر یا MCC را به عنوان کنسرسیوم تحقیقاتی در جهت مقابله با ژاپنی‌ها شکل دادند. در بریتانیا، گزارش الوی[۷] باعث احیای سرمایه‌گذاری که با گزارش لایت‌هیل[۸] قطع شده بود، گردید.

روی هم رفته، صنعت از فروش چند میلیون در سال ۱۹۸۰ به فروش ۲ بیلیون دلار در سال ۱۹۸۸ رسید. ولی مدت کوتاهی پس از آن دوره موسوم به "زمستان هوش مصنوعی" فرا رسید که در آن شرکت‌ها موفق به برآورده کردن خواسته‌های نامعقول متقاضیان نبودند.

۱- Roger Schank
۲- Cullingford
۳- Rieger
۴- Kolodner
۵- Wilensky
۶-Mc Dermott
۷-Alvey
۸-Lighthill

۱-۴-۷- بازگشت شبکه‌های عصبی (۱۹۸۶- تاکنون)

با وجود اینکه علم کامپیوتر از اواخر دهه ۱۹۷۰ شاخه شبکه‌های عصبی را رها کرد، این علم در شاخه‌های دیگر ادامه پیدا کرد. فیزیکدانانی همچون هاپفیلد[۱] (۱۹۸۲) از تکنیکهایی از مکانیک آماری (statistical mechanics) استفاده کردند تا خواص ذخیره‌سازی و بهینه سازی شبکه‌ها را با در نظر گرفتن دسته بزرگی از نرونهای ساده به عنوان دسته بزرگی از اتمها تحلیل کنند. روانشناسانی از جمله دیوید راملهارت[۲] و جف هینتون[۳] مطالعه بر روی مدل‌های شبکه عصبی حافظه را ادامه دادند. محرک واقعی در اواسط دهه ۱۹۸۰ بوجود آمد، هنگامی که حداقل چهار گروه متفاوت الگوریتم یادگیری انتشار به عقب (back-propagation) را که در سال ۱۹۶۹ بوسیله برایسون[۴] و هو[۵] کشف شده بود، بازسازی کردند. الگوریتم در بسیاری از مسائل علوم کامپیوتر و روانشناسی به کار برده شد و توزیع گسترده نتایج در مجموعه‌ای با عنوان پردازش موازی توزیع شده (Rumelhart and McClelland, ۱۹۸۶) هیجان زیادی بوجود آورد.

این مدل‌های موسوم به ارتباط‌گرا در سیستم‌های هوش‌مصنوعی از دید عده‌ای به عنوان رقبای جدی مدل‌های نمادین توسعه یافته توسط سیمون و ناول و نیز رویکردهای منطقی مک کارتی و بقیه شناخته می‌شدند (Smolensky, ۱۹۸۸). واضح است که انسان‌ها نمادها را دستکاری می‌کنند-در واقع کتاب "گونه‌های نمادین" ترنس دیکن[۶] این موضوع را از خصوصیات انسان‌ها می‌داند. ولی بسیاری از ارتباط‌گراهای تندرو این امر که آیا دستکاری نمادها نقش قابل توجهی در مدل‌های شناختی مفصل‌تر دارند را مورد شک و پرسش قرار داده‌اند. این پرسش تا کنون بی‌پاسخ مانده است ولی دیدی که اکنون وجود دارد دو مدل ارتباط‌گرا و نمادین را مکمل یکدیگر می‌داند و نه رقیب هم.

۱-۴-۸- هوش مصنوعی به عنوان علم شناخته می‌شود (۱۹۸۷-تاکنون)

در سالهای اخیر شاهد دریایی از تغییرات در محتوا و متدولوژی تحقیقات در زمینه هوش مصنوعی بوده‌ایم.[۷] درحال حاضر تحقیقات بر مبنای تئوری‌های موجود انجام می‌شوند و تئوری‌های جدید ارائه نمی‌گردد. بعلاوه اینکه ادعاها بر اساس قضایای دقیق یا آزمایشات سخت بیان می‌گردند تا بر اساس شهود.

AI بر مبنای اعتراضی بر محدودیت‌های رشته‌هایی چون نظریه کنترل و آمار پایه‌گذاری شد ولی امروزه دربرگیرنده این رشته‌ها نیز می‌باشد. همان گونه که David Mc Allester (۱۹۹۸) مطرح کرد:

۱-Hopfield
۲-David Rumelhart
۳-Geoff Hinton
۴-Bryson
۵-Ho
۶-Terrence Deacon
۷- عده‌ای این تغییرات را به صورت پیروزی مرتبها – عده‌ای که تصورمی‌کنند تئوریهای هوش مصنوعی می‌بایست بر دقت ریاضیات بنا شوند – بر نامنظم‌ها – عده‌ای که ترجیح می‌دهند تعداد زیادی از ایده‌ها را امتحان کنند، تعدادی برنامه بنویسند و بعد حساب کنند که چه چیز به کار می‌آید. هر دو رویکرد مهم هستند. حرکت به سمت مرتب بودن نشان می‌دهد که رشته به سطحی از بلوغ و استحکام رسیده است. (اینکه این استحکام با یک ایده نامنظم دچار تزلزل می‌شود یا خیر خود سوال دیگری است.)

در دوران ابتدایی AI چندان بعید به نظر نمی‌رسید که با به وجود آمدن اشکال جدید محاسبات نمادین مانند شبکه‌های معنایی و فریم‌ها بسیاری از نظریه‌های کلاسیک باطل شوند. این مسئله منتهی به انزواگرایی‌ای شد که باعث جدا شدن کامل هوش مصنوعی از سایر علوم کامپیوتر گردید. این انزواگرایی در حال حاظر کنار گذاشته شده‌است. زیرا این باور وجود دارد که یادگیری ماشین و نظریه اطلاعات، استدلال غیر قطعی و مدل‌های آماری، جستجو و بهینه‌سازی و کنترل کلاسیک، استدلال ماشینی و روشهای فرمال و تحلیلهای آماری باید در کنار یکدیگر بوده و نباید از هم جدا شوند.

برای پذیرفته شدن AI به عنوان شاخه‌ای از علم، فرض‌ها باید براساس آزمایشات تجربی قوی بوده و درجه اهمیت نتایج باید با تحلیلهای آماری مورد بررسی قرارگرفته شوند. (۱۹۹۵) امروزه تکرار مرتب آزمایشات از طریق استفاده از اینترنت و مخازن تست داده به راحتی امکان‌پذیر است.

شاخه شناخت گفتاری الگوها را معرفی می‌کند. در دهه ۱۹۷۰ تعداد زیادی از معماری‌ها و رویکردهای گوناگون به کار گرفته شدند. تعداد زیادی از این ها خاص منظوره و ضعیف بودند و بر روی مثالهای خاصی نشان داده می‌شدند. در سالهای اخیر، رویکردهایی بر مبنای HMMs (Hidden Markov Models) بر این عرصه تسلط یافته‌اند. دو جنبه HMMs مرتبط با بحث فعلی هستند.اول اینکه آنها بر مبنای تئوری دقیق ریاضی استوارند. این به محققین در زمینه گفتار این امکان را داده است تا کار خود را بر مبنای نتایج ریاضی که طی دهه‌های متمادی در رشته‌های دیگر بدست آمده است استوار کنند. دوم اینکه آنها طی فرآیند یادگیری از روی حجم عظیمی از داده‌های گفتاری واقعی بدست آمده‌اند. این موضوع خدشه‌ناپذیری کارایی را تضمین می‌کند. تکنولوژی گفتار و شاخه وابسته به آن (درک حروف نوشته شده) کاربردهای زیادی در صنعت و برنامه‌های کاربردی پیدا کرده‌اند.

در این میان، شبکه‌های عصبی نیز در راستای روند فوق‌الذکر حرکت کردند. در دهه ۱۹۸۰ تلاش‌های بسیاری برای گسترش کاربردهای شبکه‌های عصبی و فهم تفاوت‌های آن با تکنیک‌های قدیمی‌تر انجام شد. با استفاده از متدهای پیشرفته و چهارچوبهای نظری این باور به وجود آمد که اکنون دیگر شبکه‌های عصبی قابل مقایسه با تکنیک‌های متناظری از شاخه‌های آمار، شناخت الگو و یادگیری ماشین است و باید برای هرکاربرد بهترین روش امتحان و به کارگرفته شود. در نتیجه این پیشرفتها تکنولوژی موسوم به داده‌کاوی (صنعت نیرومند جدیدی) را پایه‌ریزی کرد.

استدلال مبتنی بر احتمال در سیستم‌های هوشمند که توسط جودیا پرل[۱] پیشنهاد شد (۱۹۸۸) منجر به مقبولیت جدید تئوری احتمالات و تصمیم‌گیری در هوش مصنوعی شد (و به دنبال آن مقاله پیترچیزمن[۲]"در دفاع از احتمالات" باعث قوت‌گیری دوباره علاقه‌ها شد). شبکه بیزی (Baysian network) به‌منظور نمایش کارا و استدلال دقیق با دانش غیرقطعی اختراع شد. این رویکرد بر بسیاری از مشکلات سیستم‌های مبتنی بر احتمال دهه ۱۹۶۰ و دهه ۱۹۷۰ غلبه کرد و باعث تسلط تحقیقات هوش مصنوعی بر استدلال‌های غیر قطعی و سیستم‌های خبره شد. این رویکرد قابلیت یادگیری از تجربه‌ها را می‌دهد و ترکیبی مناسب از روش‌های کلاسیک AI و شبکه‌های عصبی فراهم می‌کند. کارهای جودیا پرل (۱۹۸۲a) و اریک هرویتز[۳] و دیود هکرمن[۱] (هرویتز و

---

[۱]-Judea Pearl
[۲]-Peter Cheeseman
[۳]-Eric Horvitz

هکرمن، ۱۹۸۶؛ هرویتز و دیگران، ۱۹۸۶)ایده سیستم‌های خبره ضابطه‌ای را ارتقا داد: کسانی که بر مبنای قوانین تئوری تصمیم‌گیری، به صورت عقلایی عمل می‌کنند و از انسان‌های خبره تقلید می‌کنند. سیستم عامل windows™ شامل چندین سیستم خبره ضابطه‌ای تشخیص خطا برای تصحیح مشکلات است.

تحولات آرام مشابهی در رباتیک، بینایی رایانه، یادگیری ماشین (شامل شبکه‌های عصبی)، و نمایش دانش به وقوع پیوسته است. ترکیب درک بهتر مسائل و خواص پیچیدگی آنها با پیچیدگی‌های ریاضی، منجر به لیستهای تحقیقاتی عملی‌تر و روشهای قویتر شده است. در بسیاری از موارد رسمیت بخشیدن و تخصصی کردن منجر به شکستن شاخه‌ها به بخش‌های کوچکتر شده است: مباحثی مثل بینایی و رباتیک به طور فزاینده‌ای از جریان اصلی کارهای AI فاصله گرفته‌اند. دیدگاه یکپارچه به AI به عنوان طراحی عامل عقلایی، دیدگاهی است که همه این شاخه‌های جدا شده را یکی می‌کند.

۱-۴-۹- ظهور عامل‌های هوشمند (۱۹۹۵ تاکنون)

شاید با دلگرمی پیشرفت حل زیر مسائل هوش مصنوعی، محققین بررسی دوباره مساله "عامل کامل" را آغاز کرده‌اند. کار الن ناول[۲]، جان لرد[۳] و پل رزنبلوم[۴] بر روی SOAR (نول، ۱۹۹۰؛ لرد و دیگران، ۱۹۸۷) بهترین مثال آشنا از معماری یک عامل کامل در هوش مصنوعی می‌باشد. حرکات جدید، در جهت درک فعالیت عامل در یک محیط واقعی با ورودی حسگرهای پیوسته است. یکی از مهمترین محیطها برای عامل هوشمند اینترنت است. سیستم‌های هوش مصنوعی به اندازه‌ای وارد کاربردها بر پایه وب شده‌اند که پسوند "bot-" وارد زبان روزمره شده است. علاوه بر این تکنیک‌های AI شاخه‌ای برای بسیاری از ابزارهای اینترنت مثل موتورهای جستجو، سیستم‌های پیشنهادگر و سیستم‌های ساخت وب‌سایت‌ها می‌باشد.

یکی از نتایج تلاش برای ساخت عامل کامل نیاز به سازماندهی مجدد زیر شاخه‌های منفصل هوش مصنوعی است تا نتایج آن‌ها ترکیب شده و تبدیل به طراحی واحد یک عامل شوند. به طور خاص در حال حاضر این باور وجود دارد که سیستم‌های حسگر (دیداری، صوتی،تشخیص سخن و ...) قادر نیستند اطلاعات کاملاً قابل اطمینانی در مورد محیط اطراف انتقال دهند. بنابراین استدلال و طراحی سیستم‌ها باید قادر به کنترل عدم قطعیت باشد. یک نتیجه عمده دیگر دیدگاه عاملی این است که AI به سمت ارتباط نزدیک‌تر با شاخه‌هایی از جمله نظریه کنترل و اقتصاد که آنها نیز با عامل‌ها سروکار دارند کشیده شده است.

۱-۵- تکنولوژی روز

AI امروزه چه کارهایی می‌تواند انجام دهد؟ دادن یک جواب مختصر دشوار است زیرا فعالیت‌های زیادی در زیرشاخه‌های مختلف انجام شده است. در اینجا به تعدادی از این کاربردها اشاره می‌کنیم:

طرح‌ریزی و زمانبندی خودمختار: صدها میلیون مایل دور از کره زمین برنامه عامل راه‌دور NASA به عنوان اولین برنامه طرح‌ریزی خودمختار برای کنترل زمان‌بندی عملیات یک سفینه فضایی شناخته شد. عامل راه‌دور

۱-David Heckerman
۲-Allen Newell
۳-John Laird
۴-Paul Rosenbloom

طرح‌های بلندمرتبه (مشخص شده از زمین) تولید کرده و بر فعالیتهای سفینه‌های فضایی در حال اجرای این طرح‌ها نظارت کرده و خطاها را در صورت بروز ردیابی، تشخیص و بازیابی می‌کند.

بازی کردن: برنامه موسوم به Deep Blue شرکت IBM اولین برنامه کامپیوتری بود که موفق به شکست دادن قهرمان جهان در یک مسابقه شطرنج شد و گری کاسپاروف را با حساب ۳.۵ به ۲.۵شکست داد (Goodman and Keene, ۱۹۹۷) کاسپاروف اظهار داشت نوع جدیدی از هوشمندی را در آن سوی بازی احساس کرده است. مجله نیوزویک این بازی را به عنوان آخرین مقاومت ذهن انسان توصیف کرد. بعد از این بازی ارزش سهام IBM ۱۸میلیون دلار افزایش یافت.

کنترل خودکار: سیستم بینایی کامپیوتر ALVINN برای هدایت یک خودرو برای نگه داشتن آن در مسیر آموزش داده شده بود. این سیستم در مینی ون NAVLAB دانشگاهCMU که توسط کامپیوتر کنترل می‌شد قرار داده شد و برای مسیریابی در عرض آمریکا استفاده شد. برای ۲۸۵۰ مایل در ۹۸٪زمانها برنامه هدایت فرمان را برعهده داشت و ۲٪ بقیه را که اکثراً خروجی‌های شیبدار بودند راننده انسان هدایت کرد.NAVLAB دوربینهای ویدیوئی‌ای در اختیار داشت که بوسیله آنها تصویر جاده‌ها را به ALVINN انتقال می‌داد و پس از محاسبه آنها بهترین جهت برای هدایت بر اساس تجربیات بدست آمده از آزمایشات تنظیم می‌شد.

تشخیص پزشکی: برنامه‌های تشخیص پزشکی بر پایه تحلیل‌های احتمالاتی در بسیاری از بخشهای پزشکی قادر به کار در سطح یک پزشک خبره بوده‌اند. هکرمن(۱۹۹۱) موردی را تشریح می‌کند که یک متخصص برجسته در پاتولوژی غده لنفاوی، مورد مشکلی را برای سیستم خبره توضیح می‌دهد و تشخیص آن را بررسی می‌کند. او جواب سیستم را مسخره می‌کند. سازندگان سیستم به او پیشنهاد می‌کنند که توضیحی از رایانه برای تشخیصش بخواهد. ماشین فاکتورهای اصلی در تصمیمش را بیان می‌کند و روابط ضروری چندین علامت بیماری در این مورد را توضیح می‌دهد. متخصص اشتباهش را می‌پذیرد.

برنامه‌ریزی لشکرکشی نظامی: در طول بحران سال ۱۹۹۱ خلیج فارس نیروهای آمریکا ابزار تحلیل و طراحی دوباره پویا، DART(Cross and Walker, ۱۹۹۴)، را مستقر کردند تا زمانبندی و برنامه‌ریزی لشکرکشی را برای حمل و نقل‌ها انجام دهد. این کار شامل ۵۰۰۰۰ وسیله نقلیه محموله و نفرات در آن واحد می‌شد و باید نقاط شروع، مقصد، مسیر و رفع تداخلات برای پارامترهای مختلف را نیز به حساب می‌آورد. تکنیک‌های برنامه-ریزی AI اجازه تولید یک برنامه را در چند ساعت می‌داد که با متدهای قدیمی این زمان به هفته‌ها می‌رسید. آژانس پروژه‌های تحقیقات گسترده دفاع(DARPA) اظهار داشت که تنها همین کاربرد میزان سرمایه‌گذاری شده در ۳۰ سال گذشته را جبران کرد.

رباتیک: بسیاری از جراحان اکنون از رباتها به عنوان دستیار در ریز جراحی‌ها استفاده می‌کنند. HipNav (DiGioia et al., ۱۹۹۶) سیستمی است که تکنیک‌های بینایی کامپیوتر را برای ایجاد مدل‌های سه بعدی از آناتومی درونی بیمار استفاده می‌کند و سپس از رباتیک کنترل برای هدایت جاگذاری پروتز جایگزین مفصل ران استفاده می‌کند.

درک زبان و حل مسئله: PROVERB (Littman et al., ۱۹۹۹)برنامه کامپیوتری‌ای است که جدول کلمات متقاطع را بهتر از بسیاری از انسان‌ها حل می‌کند. این کار با استفاده از محدودیتهایی روی فیلترهای ممکن برای کلمات، یک پایگاه داده بزرگ و انواع مختلف منابع اطلاعات شامل فرهنگ لغات و پایگاه داده‌های online مانند

لیستی از فیلمها و بازیگرها که در این جداول ظاهر می‌شوند انجام می‌شود. برای مثال برنامه مشخص می‌کند که کلید "داستان Nice" را با کلمه ETAGE پر کند زیرا پایگاه داده شامل زوج کلید/جواب "داستان در فرانسه/ETAge" است و چون برنامه تشخیص می‌دهد که الگوی "Nice X" و "X در فرانسه" را معمولاً جواب یکسانی دارند. این برنامه نمی‌داند که کلمه Nice شهری در فرانسه است ولی جدول را حل می‌کند.

اینها فقط مثال‌های محدودی از سیستم‌های هوش مصنوعی هستند که امروزه وجود دارند. این‌ها مثال‌های جادویی یا علمی-تخیلی نیستند بلکه دانش، مهندسی و ریاضی هستند که این کتاب تلاش می‌کند مقدمه‌ای بر آنها فراهم آورد.

۱-۶- خلاصه

این فصل به توصیف هوش مصنوعی و پیش‌زمینه فرهنگی که بر مبنای آن بوجود آمده است اشاره می‌کند. تعدادی از نکات مهم آن عبارتند از:

- افراد مختلف هوش مصنوعی را از دیدگاه‌های متفاوت می‌نگرند. دو سوال مهمی که مطرح می‌شود این است که: آیا شما به رفتار اهمیت می‌دهید یا تفکر؟ آیا شما می‌خواهید انسان را مدل کنید یا از روی یک استاندارد ایده‌آل کار کنید؟
- در این کتاب ما دیدگاهی را به کار گرفتیم که هوشمندی را به صورت رفتار معقولانه در نظر می‌گیرد. به صورت ایده‌آل، در یک وضعیت، عامل هوشمند بهترین عمل ممکن را انجام می‌دهد. ما به مساله ساختن عامل‌ها که با این توصیف، هوشمند هستند خواهیم پرداخت.
- فلاسفه (به ۴۰۰ سال قبل از میلاد برگردیم) با در نظر گرفتن ایده‌هایی از جمله این که ذهن از جهاتی شبیه به ماشین می‌باشد، ذهن بر روی دانشی که به یک زبان داخلی رمز شده است عمل می‌کند و این که فکر مورد استفاده قرار می‌گیرد تا به رفتار درستی برای انجام دادن برسیم، هوش مصنوعی را ممکن ساختند.
- ریاضیدانان ابزارهای کار کردن با عبارات منطقی قطعی و عبارات غیر قطعی احتمالی را فراهم کردند. آنها همچنین اساس استدلال در مورد الگوریتم‌ها را بنا نهادند.
- اقتصاددانان مسئله تصمیم‌گیری را فرموله می‌کنند که خروجی قابل انتظار از تصمیم گیرنده را ماکزیمم می‌کند.
- روانشناسان این ایده که انسان‌ها و حیوانات دیگر می‌توانند به صورت ماشین‌های پردازنده اطلاعات در نظر گرفته شوند را قوت بخشیدند. زبان‌شناسان نشان دادند که استفاده از زبان، مناسب با این مدل می‌باشد.
- مهندسی کامپیوتر محصولاتی را ارائه کرد که کاربردهای هوش مصنوعی را ممکن ساخت. برنامه‌های هوش مصنوعی خیلی بزرگ هستند و نمی‌توانند بدون پیشرفت‌های بزرگی که صنعت کامپیوتر در سرعت و حافظه فراهم آورده است، اجرا شوند.
- نظریه کنترل با ابزارهای طراحی‌ای سروکار دارد که به طور بهینه بر پایه بازخوردهای محیط عمل می‌کنند. در ابتدا ابزارهای ریاضی نظریه کنترل کاملاً با AI متفاوت بودند اما این دو شاخه در حال نزدیک‌تر شدن به یکدیگر هستند.

- تاریخ هوش مصنوعی دوره‌های موفقیت، خوشبینی بی جا و نتیجتاً قطع شدن اشتیاق و سرمایه‌گذاری را به خود دیده است. همچنین دوره‌هایی از بوجود آمدن رویکردهای خلاقانه و انتخاب بهترین آنها نیز وجود داشته است.
- AI در دهه‌های اخیر پیشرفت سریعتری داشته است زیرا استفاده بیشتر از روشهای عملی برای انجام آزمایشات و مقایسه رویکردها استفاده شده است.
- پیشرفتهای اخیر در فهم اساس نظری هوشمندی همگام با پیشرفت‌ها در توانایی‌های سیستم‌های واقعی به وجود آمده است.

۱-۷- تمرین‌ها

۱-۱ تعریف کنید :الف) هوش، ب) هوش مصنوعی، ج) عامل.

۱-۲ مقاله اصلی تورینگ در زمینه هوش مصنوعی را مطالعه کنید (تورینگ، ۱۹۵۰). در این مقاله، او به چندین ایراد بالقوه در مورد عمل پیشنهاد شده و تستش در مورد هوشمندی اشاره می‌کند. کدامیک از ایرادات هنوز نیز واردند؟ آیا تکذیب‌های تورینگ صحیح‌اند؟ آیا شما ایراداتی به نظرتان می‌رسد که برخاسته از پیشرفت‌ها نسبت به زمانی که او مقاله را نوشته است، باشد؟در این مقاله او پیش بینی کرده است که تا سال ۲۰۰۰ یک رایانه ۳۰ درصد شانس قبولی در تست ۵ دقیقه‌ای تورینگ با یک پرسش‌کننده بی تجربه دارد.آیا این منطقی است؟

۱-۳ هر سال جایزه Loebner به برنامه‌ای که به گذراندن یک نسخه از تست تورینگ نزدیک‌تر باشد تعلق می‌گیرد. در مورد آخرین برنده جایزه Loebner تحقیق و گزارش کنید. چه تکنیک‌هایی استفاده می‌کند؟ چگونه تکنولوژی روز هوش مصنوعی را گسترش می‌دهد؟

۱-۴ کلاس‌های مشهوری از مسائل هستند که به طور رام‌نشدنی‌ای برای رایانه‌ها مشکل هستند و کلاسهای دیگری هستند که به طور قابل اثباتی تصمیم ناپذیر برای هر رایانه هستند.آیا این بدین معنا می‌باشد که هوش مصنوعی ناممکن است؟

۱-۵ فرض کنید برنامه ANALOGY ایوان را بسط دهیم تا بتواند در یک تست آی کیو نمره ۲۰۰ بیاورد.آنگاه آیا ما برنامه‌ای داریم که هوشمندتر از یک انسان است؟ توضیح دهید.

۱-۶ چطور درونکاوی -گزارش دادن افکار درونی شخص- ممکن است نادقیق باشد؟ آیا ممکن است من در مورد آنچه فکر می‌کنم اشتباه کنم؟

۱-۷ ادبیات هوش مصنوعی را وارسی کنید و توضیح دهید که آیا وظایف زیر به طور رایج توسط رایانه‌ها انجام می‌شوند:

الف) بازی کردن تنیس رو میز(پینگ پنگ).

ب) رانندگی در مرکز شهر کایرو(Cairo).

ج) بازی کردن بریج(Bridge) در سطح رقابتی.

د) کشف و اثبات قضایای ریاضی.

ه) نوشتن عمدی یک داستان مسخره.

و) دادن یک پیشنهاد قانونی قوی در یک حوزه تخصصی قانون.

ی) ترجمه بلا درنگ انگلیسی محاوره‌ای به سوئدی محاوره‌ای.

برای کارهایی که عملی نیستند سعی کنید دریابید که مشکلات چه هستند و تخمین بزنید که چه موقع حل می‌شوند.

۱-۸ عده‌ای از نویسندگان ادعا کرده‌اند که مهارت‌های ادراکی و حرکتی عامل‌ها مهمترین بخش‌های هوشمندی هستند و توانایی‌های سطح بالاتر لزوماً اضافی اند-چیزهای اضافه شده به امکانات پایه‌ای هستند. بدون شک، قسمت اعظم تکامل و بخش عمده مغز مربوط به ادراک و حرکت می‌باشد در حالی که هوش مصنوعی کارهایی مانند بازی کردن و استنتاج منطقی را از بسیاری جهات ساده‌تر از ادراک و حرکت در دنیای واقعی یافته است. فکر نمی‌کنید که تمرکز مرسوم هوش مصنوعی بر روی توانایی‌های ادراکی سطح بالاتر بر جای نادرستی واقع شده است؟

۱-۹ چرا تکامل سیستم‌هایی را نتیجه می‌دهد که منطقی عمل می‌کنند؟ این سیستم‌ها برای دستیابی به چه اهدافی طراحی شده‌اند؟

۱-۱۰ آیا واکنش‌ها (مانند دور کردن دست از اجاق داغ) منطقی هستند؟ آیا هوشمندانه هستند؟

۱-۱۱ "قطعاً رایانه‌ها نمی‌توانند هوشمند باشند- آنها تنها کاری را انجام می‌دهند که برنامه‌نویسان به آنها بگویند." آیا جمله دوم صحیح است و آیا جمله اول را نتیجه می‌دهد؟

۱-۱۲ "قطعاً حیوانات هوشمند نیستند-آنها کاری را انجام می‌دهند که ژن‌هایشان به آنها بگوید." آیا جمله دوم صحیح است و آیا جمله اول را نتیجه می‌دهد؟

۱-۱۳ "قطعاًحیوانات،انسان‌ها و کامپیوترها نمی‌توانند هوشمند باشند-آنها کاری را که به اتم‌های سازنده‌شان توسط قوانین فیزیک گفته شده انجام می‌دهند." آیا جمله دوم صحیح است و آیا جمله اول را نتیجه می‌دهد؟

تست‌های طبقه‌بندی شده فصل اول

۱- تست تورینگ در راستای ارزیابی کدامیک از رویکردهای هوش مصنوعی طراحی شد؟ (تالیفی)

۱) تشابه به انسان – تفکر
۲) تشابه به انسان – رفتار
۳) عقلانیت – تفکر
۴) عقلانیت – رفتار

۲- کدامیک از جملات زیر درباره عامل عقلایی درست است؟ (تالیفی)

۱) عاملی که تنها بخشی از محیط را درک می‌کند نمی‌تواند عامل عقلایی باشد.
۲) یک عامل عقلایی همیشه بهتر از عامل‌های غیرعقلایی عمل می‌کند زیرا نتیجه واقعی اعمالش را می‌داند.
۳) گاهی رفتارهای عقلایی وجود دارند که بدون استنباط بدست می‌آیند.
۴) استنباط صحیح همان عقلایی بودن است.

۳ - یک عامل توانسته است تمام انسان‌ها را دریک بازی شکست دهد. کدام جمله به نظر شما صحیح می‌رسد؟(فناوری اطلاعات ۸۵)

۱) این عامل قطعا در تست تورینگ قبول خواهد شد.
۲) این عامل می‌تواند بدون توجه به نحوه تفکر انسان طراحی شده باشد.
۳) عامل صرفا برتکرار و شبیه‌سازی روش‌های بازیگران حرفه‌ای استوار است.
۴) عامل می‌تواند صرفا برسرعت بالا و حافظه حجیم سوپر کامپیوترها تکیه کند.

پاسخنامه تشریحی تست‌های فصل اول

۱) گزینه ۲ درست است.

تست تورینگ برای ارزیابی یک سیستم از جهت تشابه رفتار آن با انسان طراحی شد. برای این منظور یک انسان سوالاتی را برای کامپیوتر مطرح می‌کند و کامپیوتر در صورتی این تست را با موفقیت پشت سر می‌گذارد که پرسشگر نتواند تشخیص دهد که پاسخ داده شده از طرف یک انسان بوده یا یک کامپیوتر.

۲) گزینه ۳ درست است.

گزینه ۱ نادرست است: در صورت وجود عدم قطعیت در محیط، عامل عقلایی بهترین نتیجه مورد انتظار را می‌گیرد. بنابراین اگر عامل تنها بخشی از محیط را ببیند باز هم می‌تواند عقلایی باشد.

گزینه ۲ نادرست است: یک عامل عقلایی همیشه بهتر از عامل‌های غیرعقلایی عمل می‌کند اما از نتیجه واقعی اعمالش خبر ندارد.

گزینه ۳ درست است: گاهی رفتارهای عقلایی وجود دارند که بدون استنباط بدست می‌آیند. به عنوان مثال کشیدن دست از روی یک اجاق داغ یک عکس العمل غیر ارادی است که بسیار موفقیت آمیزتر از یک عکس‌العمل کند است که با سنجش و بررسی انجام می‌شود

گزینه ۴ نادرست است: تنها راه رسیدن به عقلانیت (عقلایی بودن)، استنباط صحیح نیست (یعنی استنباط صحیح تنها بخشی از عقلانیت است)

۳) گزینه ۲ درست است.

برای تحیل و ارزیابی هوشمندی یک عامل، می‌توان از ۴ رویکرد متفاوت استفاده کرد:

تفکر مشابه انسان

تفکر عقلایی

رفتار مشابه انسان

رفتار عقلایی

بنابراین، طراحی یک عامل با چنین ویژگی لزوما نمی‌تواند براساس تفکر مشابه انسان باشد و ممکن است رویکردی متفاوت برگزیده باشد.

فصل دوم

عامل‌های هوشمند

در این فصل در مورد طبیعت عامل‌ها (از نوع کامل و غیرکامل)، انواع محیط‌ها و انواع عامل‌هایی که در آن محیط ساخته شده‌اند بحث خواهیم کرد.

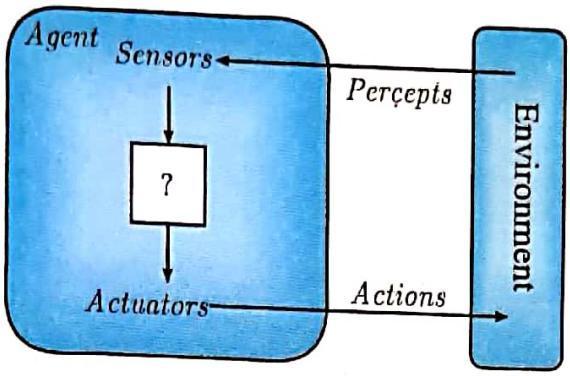
در فصل ۱ مفهوم عامل عقلایی به عنوان مرکز رویکرد ما به هوش مصنوعی در نظر گرفته شد. در این فصل این مفهوم را روشن‌تر می‌کنیم. خواهیم دید که مفهوم عقلانیت بر روی هر عامل عمل کننده در هر محیط قابل تصوری قابل اعمال است.

با آزمودن محیط، عامل و پیوند بین آنها شروع می‌کنیم. مشاهده اینکه بعضی عامل‌ها بهتر از بقیه عمل می‌کنند طبیعتاً ما را به سمت ایده عامل عقلایی (عاملی که به بهترین نحو ممکن عمل می‌کند) هدایت می‌کند. اینکه یک عامل تا چه اندازه میتواند خوب عمل کند بستگی به طبیعت محیط دارد(بعضی محیط‌ها سخت‌تر از محیط‌های دیگر می‌باشند). در این فصل یک دسته‌بندی کلی از محیط‌ها ارائه می‌دهیم و نشان می‌دهیم چگونه **مشخصه‌های یک محیط در طراحی عامل مناسب برای آن محیط تأثیرگذار است.**

۲-۱- عامل‌ها و محیط‌ها

هر چیزی که محیطش را از طریق حسگرها درک کرده و بر روی محیطش از طریق محرک‌ها تاثیر بگذارد (با استفاده از کنش‌ها) عامل نامیده می‌شود. این ایده ساده در شکل۲-۱ شرح داده شده است. یک عامل انسانی چشم، گوش و اعضای دیگری را به عنوان حسگر و دست، پا، دهان و اعضای دیگری را به عنوان محرک دارد. یک عامل رباتیک، دوربین‌ها و یابنده‌های مادون قرمز را به‌عنوان حسگر و انواع موتورها را به‌عنوان محرک جایگزین کرده است. یک عامل نرم‌افزاری، فشردن کلید، محتویات فایل و بسته‌های شبکه را به عنوان ورودی حسگر درک کرده و با نمایش بر روی صفحه، نوشتن در فایل و فرستادن بسته‌های شبکه، بر روی محیط عمل می‌کند. فرض براین است که ادراک کنش‌های یک عامل توسط خود عامل امکان‌پذیر است (اما تضمینی برای درک آثار این کنش‌ها وجود ندارد).

به ورودی‌های دریافت شده توسط عامل ادراک گفته می‌شود. تاریخچه کامل همه آنچه که عامل دریافت کرده را دنباله ادراکات می‌نامیم. به طور کلی کنشی که یک عامل در هر لحظه از خود بروز می‌دهد میتواند به کل دنباله ادراکاتی که عامل تاکنون دریافت کرده است بستگی داشته باشد. اگر بتوانیم به هر دنباله‌ی ممکن از ادراکات کنشی اختصاص دهیم، آنگاه کم و بیش توصیف کاملی از عملکرد عامل را بیان کرده‌ایم. به بیان ریاضی،رفتار یک عامل با تابع عامل توصیف می‌شود که هر دنباله ادراکی را به یک کنش متناظر می‌کند.

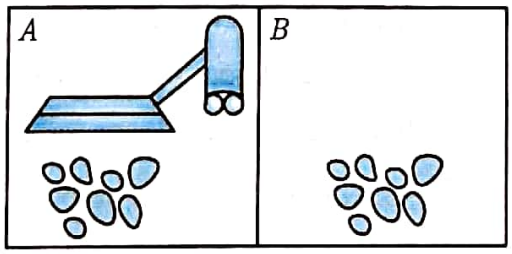


شکل ۲-۱ عامل‌ها از طریق حسگرها و محرک‌ها با محیط ارتباط برقرار می‌کنند.

می‌توان تابع عامل را به صورت یک جدول تصور کرد. برای بیشتر عامل‌ها اندازه این جدول بسیار بزرگ، یا در واقع بینهایت خواهد شد. مگر آنکه طول دنباله ادراکی را محدود کنیم (زیرا انتخاب کنش‌ها براساس دنباله ادراک‌ها انجام می‌شود). با آزمایش برروی عامل از طریق امتحان کردن تمامی دنباله‌های ادراک ممکن و ثبت کنش‌های بروز داده شده (از طرف عامل) می‌توانیم این جدول را بسازیم. البته این جدول یک مشخصه خارجی عامل است.[1] در داخل، تابع عامل با استفاده از برنامه عامل پیاده‌سازی شده است. تمییز دادن بین این دو ایده بسیار مهم است. تابع عامل توصیفی ریاضی و مجرد است در حالیکه برنامه عامل یک پیاده سازی واقعی است که روی معماری عامل در حال اجرا است.

برای تشریح این دو ایده از مثالی بسیار ساده استفاده خواهیم کرد. جهان جاروبرقی که در شکل ۲-۲ نشان داده شده‌است را درنظر بگیرید. این جهان به اندازه‌ای ساده است که میتوانیم هر آنچه که در آن اتفاق می‌افتد را توصیف کنیم.

این جهان خاص از دو محل تشکیل شده است: مربع A و مربع B. عامل جاروبرقی درک می‌کند که در کدام مربع قرار گرفته و آیا در آن گرد و خاک وجود دارد یا نه و کنش‌هایی که میتواند انتخاب کند حرکت به راست، حرکت به چپ، مکش گرد و خاک و یا انجام ندادن کنش است.



شکل ۲-۲ جهان جاروبرقی با دو خانه.

یک تابع عامل ساده می‌تواند به این شکل باشد: اگر مربع کنونی کثیف است، عمل مکش را انجام بده در غیر اینصورت به مربع دیگر برو. بخشی از جدول این تابع عامل در شکل ۲-۳ نمایش داده شده است. یک برنامه عامل ساده برای این تابع عامل در ادامه این فصل در شکل ۲-۸ ارائه شده است.

---

[1]- اگر عامل برای انتخاب کنش خود تصادفی عمل کند، باید هر رشته را بارها اجرا کنیم تا احتمال هر کنش را بدست آوریم. ممکن است به نظر برسد که تصادفی عمل کردن هوشمندانه نیست ولی در ادامه‌ی فصل خواهیم دید که میتواند بسیار هوشمندانه باشد.

با نگاه کردن به شکل ۲-۳ متوجه می‌شویم که عامل جهان جاروبرقی با پر کردن ستون سمت راست به روش‌های مختلف و به سادگی قابل تعریف است. بنابراین پرسش اصلی این است که روش درست برای پر کردن این جدول چیست؟ یا به عبارت دیگر چه چیز یک عامل را خوب، بد، هوشمند یا نادان می‌کند؟ به این پرسش‌ها در بخش بعد پاسخ داده خواهد شد.

رشته‌ی ادراکات	کنش
[A, clean]	راست
[A, dirty]	مکش
[B, clean]	چپ
[B, dirty]	مکش
[A, clean],[A, clean]	راست
[A, clean],[A, dirty]	مکش
⋮	⋮
[A, clean],[A, clean],[A, clean]	راست
[A, clean],[A, clean],[A, dirty]	مکش
⋮	⋮

شکل ۲-۳ بخشی از جدول تابع عامل ساده برای جهان جاروبرقی شکل ۲-۲

پیش از خاتمه این فصل گوشزد می‌کنیم که مفهوم یک عامل ابزاری برای تحلیل سیستم است و نه مشخصه‌ای مطلق برای تقسیم کردن جهان به دو بخش عامل‌ها و غیرعامل‌ها. ممکن است فردی یک ماشین حساب دستی را به عنوان یک عامل در نظر بگیرد که کنش نمایش عدد "۴" را برای دنباله ادراکات " = ۲ +۲" انتخاب کند، در حالیکه چنین تحلیلی با شناخت ما نسبت به ماشین حساب فاصله دارد.

۲-۲- رفتار خوب: مفهوم عقلانیت

یک عامل عقلایی عاملی است که کار درست انجام می دهد. به بیان مفهومی، همه مدخل‌ها در جدول تابع عامل به درستی پر شده‌اند. واضح است که انجام کار درست بهتر از انجام کار اشتباه است ولی انجام کار درست به چه معناست؟ در نگاه اول، عمل درست عملی است که منجر به موفقیت عامل شود (عامل موفق‌ترین باشد). بنابراین نیاز به روشی برای اندازه‌گیری موفقیت داریم. توصیفات محیط، حسگرها و محرک‌های یک عامل در مجموع مشخصه‌ای کامل از کاری که در مقابل عامل وجود دارد فراهم می‌کنند. با در دست داشتن این مشخصه، میتوانیم عقلانیت را به طور دقیق‌تری تعریف کنیم.

۲-۲-۱- معیارهای کارایی

معیار کارایی در برگیرنده معیاری برای سنجش موفقیت رفتار یک عامل است. وقتی که یک عامل در یک محیط رها می‌شود، دنباله‌ای از کنش‌ها مطابق با آنچه دریافت می‌کند تولید می‌کند. این دنباله از کنش‌ها باعث می‌شود که محیط، دنباله‌ای از حالت‌ها را تجربه کند. اگر دنباله مطلوب باشد، آنگاه عامل خوب عمل کرده است. واضح

است که برای همه عامل‌ها یک معیار خاص مطلوب نیست. می‌توانیم عقیده ذهنی خود عامل را درباره اینکه تا چه حد از کارکرد خود راضی بوده جویا شویم ولی برخی از عامل‌ها قادر به پاسخگویی نبوده و برخی دیگر خودشان را فریب می‌دهند. بنابراین بر انتخاب یک معیار خارجی برای سنجش کارایی عامل که معمولاً معیاری است که توسط طراح عامل تعیین شده است، تأکید می‌کنیم.[1]

عامل جاروبرقی را از بخش قبل در نظر بگیرید. ممکن است معیار کارایی را با مقدار گرد و خاکی که در یک نوبت هشت ساعته تمیز می‌کند در نظر بگیریم. در مورد یک عامل عقلایی هر آنچه که از عامل خواسته شود، بدست می‌آید. یک عامل عقلایی میتواند با تمیز کردن گرد و خاک و سپس دوباره خالی کردن آن روی زمین و تمیز کردن دوباره آن این میزان کارایی را به بیشترین حد برساند. معیار کارایی مناسبتر آن است که به عامل برای تمیز نگه داشتن زمین پاداش دهیم. برای مثال به ازای هر مربع تمیز در هر بازه زمان یک امتیاز پاداش داده می‌شود (و می‌توان تنبیه را برای برق مصرفی و صدای تولید شده در نظرگرفت). به عنوان یک قانون کلی، بهتر است که معیار کارایی مطابق با نتیجه مطلوب در محیط تعیین شود و نه مطابق با روشی که فکر می‌کنیم عامل برای رسیدن به نتیجه استفاده می‌کند. انتخاب معیار کارایی همیشه کار ساده‌ای نیست، برای مثال مفهوم زمین تمیز در پاراگراف قبلی بر اساس میانگین تمیزی در زمان است. با این حال یک مقدار تمیزی یکسان میتواند توسط دو عامل، با ۲ روش متفاوت به دست آید. یکی عاملی که در طول زمان کار متوسطی را انجام دهد و دیگری عاملی که با انرژی بیشتری کار می‌کند ولی زمان طولانی‌تری استراحت می‌کند اینکه کدام یک ترجیح دارد به علم فراشی (رفته‌گری) مربوط است ولی واقعیت این است که این موضوع یک پرسش فلسفی با دلالت دور از دسترس است. کدامیک بهتر است؟ یک زندگی بی‌ملاحظه با پستی و بلندی یا یک موجودیت یکنواخت و مطمئن؟ کدامیک بهتر است؟ اقتصادی که در آن همه در فقر متوسط زندگی می‌کنند یا اقتصادی که در آن عده ای ثروتمند و عده‌ای بسیار فقیر هستند؟ این پرسش‌ها را به خواننده واگذار می‌کنیم.

۲-۲-۲- عقلانیت

عقلانیت در هر عامل (در یک زمان داده شده) به چهار چیز بستگی دارد:

- معیار کارایی که میزان موفقیت را تعیین می‌کند.
- آنچه که عامل از پیش درباره محیط خود می‌داند.
- کنش‌هایی که عامل می‌تواند انجام دهد.
- دنباله ادراکات که تاکنون عامل درک کرده است.

براین اساس عامل عقلایی به صورت زیر تعریف می‌شود:

به‌ازای هر دنباله ممکن از ادراکات، عامل عقلایی باید کنشی را انتخاب کند که انتظار می‌رود معیار کارایی را بیشینه می‌کند (با توجه به دنباله ادراکات و با توجه به دانش درونی عامل).

عامل ساده جاروبرقی را در نظر بگیرید. اگر خانه‌ای کثیف باشد آن را تمیز و و درغیراین‌صورت به خانه‌ای دیگر حرکت می‌کند(که این همان تابع عامل نشان داده شده در شکل ۲-۳ است). آیا این عامل عقلایی است؟ بستگی

---

[1] عامل‌های انسانی معروف هستند به اینکه وقتی چیزی را که میخواهند به دست نمی‌آورند وانمود می‌کنند که آن را نمی‌خواستند. مثلاً "مهم نیست! جایزه‌ی نوبل را از ابتدا هم نمی‌خواستم!"

داردا ابتدا لازم است بگوییم که معیار کارایی چیست، چه چیزهایی در مورد محیط شناخته شده است و عامل چه حسگرها و محرک‌هایی دارد. بیایید فرض‌های زیر را در نظر بگیریم:

- معیار کارایی به ازای هر خانه تمیز در هر مرحله زمانی یک امتیاز می‌دهد(در یک طول عمر با هزار مرحله زمانی).
- جغرافیای محیط از پیش شناخته شده است (شکل۲-۲) ولی توزیع آلودگی و مکان اولیه عامل شناخته شده نیستند. خانه‌های تمیز، تمیز باقی می‌مانند و عمل مکش در یک خانه آن را تمیز می‌کند. کنش‌های راست و چپ عامل را به راست یا چپ منتقل می‌کنند مگر زمانی که عامل از محیط خارج شود که در اینصورت عامل همان جایی که هست باقی می‌ماند.
- تنها کنش‌های موجود راست، چپ، مکش و کنش پوچ (هیچ عملی انجام نشود) هستند.
- عامل، مکانی که در آن قرار دارد و اینکه آیا این مکان کثیف است یا نه را به درستی ادراک می‌کند.

ادعا می‌کنیم که تحت این شرایط عامل مسلماً عقلایی است. کارایی مورد انتظار هیچ عامل دیگری نمیتواند از این عامل بیشتر باشد. تمرین ۲-۴ از شما می‌خواهد که این امر را اثبات کنید. به سادگی می‌توان نشان داد که همین عامل تحت شرایط متفاوتی می‌توانست غیرعقلایی تلقی شود. به عنوان مثال، هنگامی که گرد و خاک تمیز شد، عامل بیهوده بین خانه چپ و راست نوسان می‌کند. اگر معیار کارایی یک امتیاز مجازات برای هر حرکت به چپ یا راست در نظر بگیرد آنگاه دیگر این عامل بهترین نتیجه مورد انتظار را نخواهد گرفت. تحت این شرایط عامل عقلایی، زمانی که همه خانه‌ها تمیز شدند، هیچ حرکتی نخواهد کرد. اگر خانه‌ی تمیزی دوباره کثیف شود، عامل گاه به گاه خانه‌ها را چک کرده و اگر لازم باشد آن را تمیز می‌کند. اگر جغرافیای محیط ناشناخته باشد عامل به جای اینکه بین خانه‌های A و B حرکت کند، باید محیط را جستجو کند. تمرین ۲-۴ از شما می‌خواهد که برای هریک از موارد گفته شده عامل‌هایی طراحی کنید.

۲-۲-۳- علم مطلق، یادگیری، خودمختاری

به تفاوت بین عقلانیت و علم مطلق توجه کنید. یک عامل عالِم مطلق، نتیجه نهایی عملش را می‌داند و بر طبق آن عمل می‌کند (اگرچه علم مطلق در واقعیت غیرممکن است). به مثال زیر توجه کنید: یکروز در امتداد خیابان شانزه لیزه قدم می‌زنم و یک دوست قدیمی را آن سوی خیابان می‌بینم. ماشینی از خیابان عبور نمی‌کند و من هم کار خاصی ندارم، پس از آنجایی که عقلایی هستم، شروع به عبور از عرض خیابان می‌کنم. در این بین، در ارتفاع ۳۳۰۰۰ پایی درب قسمت حمل بار هواپیمایی پایین می‌افتد و قبل از آنکه من آن طرف خیابان برسم، دچار آسیب می‌شوم. آیا من غیرعقلایی از خیابان عبور کردم؟ بعید است که در آگهی تسلیت من بنویسند:«آدم ساده و کم خردی تلاش کرد که از عرض خیابان عبور کند.»

در حقیقت این مثال نشان می‌دهد که عقلانیت با کمال یکی نیست. عقلانیت کارایی مورد انتظار را بیشینه در حالیکه کمال کارایی حقیقی را بیشینه می‌کند. اما طراحی عاملی با این مشخصات کار غیرممکنی است.

تعریف ما از عقلانیت به‌معنی علم مطلق نیست زیرا عقلانیت تنها به دنباله ادراکات در لحظه مورد نظر بستگی دارد. اما باید مطمئن شویم که ناخودآگاه به عامل اجازه ندهیم تا در برخی فعالیت‌های به وضوح غیر هوشمندانه شرکت کند. برای مثال، اگر عامل قبل از عبور از خیابان به هر دوسوی آن نگاه نکند، آنگاه دنباله ادراکاتش به او

نمی‌گوید که کامیونی با سرعت زیاد در حال نزدیک شدن است. آیا تعریف عقلانیت می‌گوید که تحت این شرایط عبور از خیابان کار درستی است؟ در حقیقت، چنین تفسیری از دو جهت اشتباه است: نخست، عقلانی نیست که از عرض جاده بدون نگاه کردن به دو سمت آن عبور کنیم. دوم، یک عامل عقلایی عمل « نگاه کردن» را قبل از قدم برداشتن در خیابان انتخاب می‌کند، چرا که نگاه کردن کارایی مورد انتظار را بیشینه می‌کند. به عبارت دیگر، بروز کنش به منظور بهبود ادراکات آینده (که گاهی جمع‌آوری اطلاعات نامیده می‌شود) بخش مهمی از عقلانیت است.

مثالی دیگر از جمع‌آوری اطلاعات از طریق اکتشاف تحقق می‌یابد که باید توسط عامل جاروبرقی در یک محیط ناشناخته اولیه انجام شود.

براساس تعریف، عامل عقلایی نه تنها باید به جمع‌آوری اطلاعات بپردازد بلکه باید تا حد ممکن از آنچه که ادراک می‌کند، درس بگیرد (یادگیری کند). تنظیمات ابتدایی عامل نشان‌دهنده شناخت اولیه او از محیط است اما به مرور زمان و با کسب تجربه، این دانش بهبود و افزایش می‌یابد. در مواردی که محیط از پیش، کاملاً شناخته شده است، عامل نیازی به دریافت و یادگیری ندارد و به سادگی و به درستی عمل می‌کند. البته چنین عاملی بسیار آسیب‌پذیر است. برای مثال سوسکی را در نظر بگیرید که بعد از حفر لانه و قرار دادن تخم‌هایش، گلوله کوچکی را برای بستن درب لانه خود حمل می‌کند حال اگر این گلوله در مسیر از دست سوسک بیفتد، سوسک تقلید وار و مانند پانتومیم بستن در لانه را بدون گلوله و بدون توجه به از دست دادن آن، ادامه می‌دهد. درحقیقت، فرضیاتی از رفتار سوسک درنظر گرفته می‌شود و درصورتی که این فرضیات نقض شوند، شاهد رفتار بی‌نتیجه‌ای از سوسک خواهیم بود. زنبور کمی هوشمندتر است. زنبور ماده، سوراخی را حفر می‌کند سپس از لانه خارج شده و کرم پروانه‌ای را نیش زده و به سمت سوراخ می‌کشاند دوباره وارد سوراخ می‌شود که مطمئن شود همه چیز رو به راه است. سپس کرم پروانه را به داخل سوراخ می‌کشاند و تخم‌گذاری می‌کند، کرم پروانه به عنوان منبع تغذیه زمانی که بچه‌ها از تخم خارج می‌شوند استفاده می‌شود. تا اینجا همه چیز خوب است. حال اگر شخصی کرم پروانه را زمانی که زنبور اوضاع را چک می‌کند چند اینچ جابه‌جا کند،زنبور عملیات را از مرحله کشاندن کرم پروانه دوباره انجام می‌دهد. حتی اگر صدها بار این کار انجام شود زنبور یاد نخواهد گرفت که طرح ذاتیش ناقص است و تغییری در برنامه‌اش ایجاد نخواهد کرد.

در عامل‌های موفق عمل محاسبه تابع عامل به سه بخش تقسیم می‌شود: زمانی که عامل طراحی می‌شود، بخشی از محاسبات توسط سازنده انجام می‌شود. دوم اینکه برای انتخاب کنش بعدی، محاسبات بیشتری انجام می‌دهد. و سوم زمانی که از تجربه‌هایش درس می‌گیرد محاسبات به مراتب بیشتری انجام می‌دهد تا درمورد چگونگی بهبود رفتارش تصمیم‌گیری کند. اگر کنش‌های عامل کاملاً بر اساس دانش اولیه (که توسط سازنده ایجاد شده است) باشد و از ادراکاتش برای تصمیم گیری استفاده نکند می‌گوییم عامل دارای کمبود خودمختاری است. یک عامل عقلایی باید خودمختار باشد (باید تاجایی که امکان دارد یاد بگیرد تا اطلاعات نادرست یا ناقص دانش پیشین خود را جبران کند). برای مثال عامل جاروبرقی که قدرت یادگیری پیش‌بینی زمان و مکان آلودگی را دارد از عاملی که این قابلیت را ندارد بسیار بهتر عمل می‌کند. به عنوان یک موضوع کاربردی: وقتی‌که عامل تجربه کمی دارد یا اصلا تجربه‌ای ندارد، باید به طور تصادفی عمل کند مگر آنکه طراح به او کمک کند. بنابراین، همانگونه که سیر تکاملی حیوانات هنگام تولد آنها را به واکنش‌های مورد نیاز مجهز می‌کند تا توانایی زندگی برای کسب تجربه را پیدا کرده و یاد بگیرند، معقول به نظر می‌رسد که برای یک عامل هوشمند مصنوعی علاوه بر قابلیت یادگیری، اندکی دانش اولیه نیز تعبیه شود. پس از به دست آوردن تجربه کافی از محیط، رفتار عامل

مستقل از دانش ابتدایی‌اش می‌شود. بنابراین فرایند یادگیری، به ما اجازه می‌دهد عامل ساده‌ای بسازیم که به طور موفقیت آمیزی در محیط‌های گسترده و متنوع قابل اجرا باشد.

۲-۳- طبیعت محیط‌ها

حال که تعریف عقلانیت مشخص شد، آماده‌ایم که ساخت عامل عقلایی را شروع کنیم. با این حال ابتدا باید در مورد محیط‌های وظیفه فکر کنیم که در واقع "مساله‌هایی" هستند که عامل‌های عقلایی "راه حلی" برای آنها می‌باشند. ابتدا درباره چگونگی مشخص کردن یک محیط وظیفه صحبت خواهیم کرد و فرایند را با چند مثال تشریح می‌کنیم. سپس نشان می‌دهیم که انواع مختلفی از محیط‌های وظیفه وجود دارند. نوع محیط وظیفه به طور مستقیم در طراحی برنامه عامل تاثیرگذار است.

۲-۳-۱- تعیین محیط وظیفه

در بحث مربوط به عقلانیت عامل ساده جاروبرقی، مجبور بودیم معیار کارایی، محیط و حسگرها و محرک‌های عامل را مشخص کنیم. حال همه اینها را تحت یک محیط وظیفه دسته بندی می‌کنیم و به اختصار آن را توصیف PEAS (مخفف کارایی[۱]، محیط[۲]، محرک[۳]، حسگر[۴]) می‌نامیم. در طراحی یک عامل،اولین گام تعیین محیط وظیفه به کامل‌ترین شکل ممکن است.

جهان جاروبرقی نمونه‌ی ساده‌ای بود. بیایید مسئله‌ی پیچیده‌تری را در نظر بگیریم: راننده تاکسی خودکار. ابتدا بایستی متذکر شویم که در حال حاضر ساختن تاکسی تمام خودکار تا حدی فراتر از قدرت تکنولوژی موجود است. شکل۲-۴ توصیف PEAS برای محیط وظیفه تاکسی را خلاصه می‌کند. هر کدام از این عناصر را در پاراگراف‌های آینده به طور جزئی بررسی می‌کنیم.

نوع عامل	معیار کارایی	محیط	محرک	حسگر
راننده تاکسی	امنیت، سریع، قانونی، مسافرت راحت، بیشینه کردن سود	جاده‌ها، سایر خودروها، مشتری‌ها، پیاده‌روها	چرخاندن فرمان، گاز، ترمز، بوق، سیگنال، نمایش	دوربینها، صدا، سرعت سنج، *GPS*، حسگر موتور، صفحه کلید

شکل ۲-۴ توصیف PEAS از محیط وظیفه برای راننده‌ی خودکار

اولاً معیار کارایی‌ای که می‌خواهیم راننده‌ی خودکارمان برای آن تلاش کند چیست؟ ویژگی‌های مطلوب شامل رسیدن به مقصد درست، کم کردن مصرف سوخت و خرابی، کاهش زمان و هزینه سفر، کاهش نقض قوانین راهنمایی رانندگی، بیشینه کردن امنیت و راحتی مسافر و بیشینه کردن سود. واضح است که بعضی از این اهداف با بقیه در تضاد هستند بنابراین دربرخی موارد مصالحه وجود دارد.

---

[۱]-Performance
[۲]-environment
[۳]-actuator
[۴]-sensor

علاوه بر آن، محیطی که راننده با آن مواجهه خواهد شد چیست؟ هر راننده‌ی تاکسی‌ای باید با جاده‌های مختلفی مثل مسیرهای روستایی و کوچه‌های شهرها و یا آزادراه‌هایی با ۱۲ مسیر برخورد کند. جاده شامل ترافیک، پیاده‌رو، حیوانات سرگردان، کارهای تعمیراتی جاده، ماشین پلیس و...می‌باشد. همچنین تاکسی با مسافران تعامل دارد. ممکن است تاکسی مجبور باشد در کالیفرنیای جنوبی فعالیت کند که مشکل برف وجود ندارد یا در آلاسکا که همیشه این مشکل وجود دارد. میتواند همیشه در سمت راست رانندگی کند و یا انعطاف‌پذیر بوده و قابلیت رانندگی در انگلیس و ژاپن را نیز داشته‌باشد. واضح است که هر چه محیط محدودتر باشد طراحی آسان‌تر است.

محرک‌های در دسترس یک راننده‌ی تاکسی خودکار کم و بیش همان محرک‌هایی هستند که در دسترس راننده‌ی انسانی هستند: کنترل موتور از طریق گاز و کنترل فرمان و ترمز. به علاوه خروجی صفحه نمایش یا صوتی برای صحبت با مسافران و شاید راهی برای ارتباط با خودروهای دیگر.

برای رسیدن به هدف، تاکسی باید بداند کجا قرار دارد، چه کس دیگری در جاده است و با چه سرعتی در حال حرکت است. این اطلاعات می‌توانند از ادراکات فراهم شده بوسیله یک یا چند دوربین تلویزیونی قابل کنترل، سرعت‌سنج و کیلومترشمار باشند. برای کنترل وسیله نقلیه به طور مناسب بخصوص سر پیچ‌ها، یک شتاب سنج لازم است. همچنین عامل نیازمند اطلاع از وضعیت مکانیکی وسیله نقلیه است پس به مجموعه‌ای از حسگرهای موتوری و حسگرهای سیستم‌های الکتریکی احتیاج دارد. شاید ابزارهایی داشته باشد که برای یک راننده انسانی متوسط در دسترس نیست: سیستم مکان‌یابی جهانی ماهواره‌ای[1] که اطلاعات دقیقی را با در نظر گرفتن نقشه الکترونیکی به آن بدهد؛ یا حسگرهای مادون قرمز یا امواج صوتی که فاصله از دیگر ماشین‌ها و موانع را شناسایی کند. نهایتا، به میکروفون یا صفحه کلید برای مسافرها نیاز دارد تا مقصدشان را اعلام کنند.

در شکل ۲-۵ عناصر پایه‌ی PEAS را برای تعداد دیگری از انواع مختلف عامل‌ها ارائه کرده‌ایم. مثال‌های بیشتر در تمرین ۲-۵ ارائه خواهند شد. شاید برای برخی از خوانندگان عجیب به نظر برسد که ما در لیست انواع عامل‌ها برخی از برنامه‌ها را گنجانده‌ایم که در محیطی کاملاً مصنوعی تعریف شده و بوسیله ورودی‌های صفحه کلید و خروجی‌های کاراکتری بر روی صفحه نمایش عمل می‌کنند. شاید با خود فکر کنید: « این محیط واقعی نیست، آیا هست؟» در حقیقت، اصلا تفاوت بین محیط‌های «واقعی» و «مصنوعی» حائز اهمیت نیست، بلکه آنچه که مهم است پیچیدگی بین رفتار عامل و دنباله ادراکات تولید شده بوسیله محیط و هدفی است که عامل برای خود فرض کرده است تا بدان برسد. برخی از محیط‌های واقعی بسیار ساده هستند. برای مثال یک ربات طراحی شده برای بازرسی قطعاتی که بوسیله یک تسمه حمل می‌شوند، میتواند فرضیات ساده‌کننده‌ای را بکار برد: روشنایی همواره وجود دارد، قطعات روی تسمه فقط از یک نوع خاص هستند، و تنها دو عمل میتواند انجام شود- پذیرش قطعه یا علامت زدن آن برای عدم پذیرش آن.

در مقابل برخی از عامل‌های نرم افزاری (یا ربات‌های نرم‌افزاری[2]) در دامنه‌های نامحدود و غنی وجود دارند. یک ربات نرم‌افزاری را تصور کنید که برای پرواز کردن با شبیه‌ساز پرواز ۷۴۷ طراحی شده است. شبیه‌ساز محیط پیچیده بسیار دقیقی است و عامل نرم‌افزاری باید از میان کنش‌های گسترده و متنوع، بیدرنگ یک کنش را

---

[1]- GPS
[2]- softbot

انتخاب کند. یا ربات نرم‌افزاری را تصور کنید که برای مرور کردن اجمالی منابع خبری برخط[۱] و نمایش بخش‌های جالب آن به مشتری، طراحی شده است. برای انجام کار درست، نیازمند پردازش زبان‌های طبیعی است، باید بداند هر مشتری به چه چیزهایی علاقمند است و باید برنامه‌اش را به صورت پویا تغییر دهد. اینترنت پیچیدگی بسیار زیادی (همچون جهان واقعیت) دارد و ساکنانش شامل عامل‌های مصنوعی زیادی هستند.

محیط	میزان کارایی	محرک‌ها	حسگرها	نوع عامل
بیمار، کارکنان بیمارستان	سلامتی بیمار، کمینه کردن هزینه و شکایات	نمایش پرسشها، معاینه ها، درمانها، مراجعه‌ها و تشخیص‌ها	ورودی صفحه کلید علائم بیماری، یافته‌ها، پاسخهای بیمار	سیستم تشخیص پزشکی
تصاویر از ماهواره در حال چرخش	دسته‌بندی صحیح	چاپ دسته بندی‌های منظره‌ها	پیکسلهای با شدت متنوع، رنگ	سیستم تحلیل تصاویر ماهواره‌ای
تسمه حامل اشیاء، صندوق‌ها	منتقل کردن اشیاء در صندوق صحیح	بازوها و دست‌های متصل	دوربین، حسگرهای با زاویه‌ی متصل	ربات جابجا کننده اشیاء
پالایشگاه، عملگرها	بیشینه کردن خلوص، بازده، امنیت	گرم کننده‌ها، پمپ‌ها، نمایش دهنده‌ها	دما، فشار، حسگر شیمیایی	کنترل کننده پالایشگاه
مجموعه ای از دانش آموزان، آژانس تست‌گیری	بیشینه کردن نمره‌های دانش آموز در امتحان	چاپ تمرینها، پیشنهادات، اصلاحات	ورودی صفحه کلید	آموزش دهنده زبان انگلیسی محاوره‌ای

شکل ۲-۵ مثالهای از انواع عامل‌ها و توصیف PEAS آنها

۲-۳-۲- خواص محیط‌های وظیفه

گاه دامنه‌ی محیط‌های وظیفه‌ای که در هوش مصنوعی ظاهر می‌شوند بسیار گسترده است. بااین حال فاکتورهای زیادی برای دسته‌بندی و تمییز دادن این محیط‌ها از یکدیگر وجود ندارد. این فاکتورها تا بخش زیادی تعیین کننده چگونگی طراحی عامل و قابلیت اعمال دسته‌ای از تکنیکها برای پیاده‌سازی عامل‌اند. ابتدا لیستی از فاکتورها را ارائه میکنیم و سپس تعدادی محیط وظیفه را برای شرح ایده تحلیل میکنیم. تعاریف ارائه شده غیر رسمی هستند. در فصلهای بعدی اصطلاحات و مثال‌های دقیق‌تری از هر نوع محیط ارائه می‌شود.

کاملاًمشاهده‌پذیر[۲] درمقایسه با پاره‌مشاهده‌پذیر[۳]

اگر سنسور عامل امکان دسترسی به حالت کامل محیط خود (در هر لحظه) را داشته باشد، می‌گوییم که محیط وظیفه کاملا مشاهده‌پذیر است[۴]. یک محیط بطور موثر مشاهده‌پذیر است اگر حسگرها همه جوانب مرتبط با

---

[۱]-online

[۲]-fully observable

[۳]-partially observable

[۴]- در ویرایش اول این کتاب از اصطلاحات دسترس‌پذیر و دسترس‌ناپذیر به جای کاملاً مشاهده‌پذیر و نیمه مشاهده‌پذیر، از غیر قطعی به جای اتفاقی و از غیر مرحله‌ای به جای ترتیبی استفاده شده بود. اصطلاحات جدید با کاربرد امروزی آن‌ها سازگارتر هستند.

انتخاب کنش را تشخیص دهند(این ارتباط نیز خود به معیار کارایی ما وابسته است). محیط‌های کاملا مشاهده‌پذیر از آن جهت مطلوب‌اند که عامل نیازی به نگه‌داشتن هیچ حالت درونی‌ای برای پیگیری (وضعیت) دنیا ندارد. از دیگر سو یک محیط ممکن است به دلیل حسگرهای نادقیق (و یا به دلیل نویز زیاد و یا به این دلیل که بخشهایی از حالات توسط حسگرها قابل دسترس نیستند) پاره‌مشاهده‌پذیر باشد.

برای مثال یک عامل جاروبرقی با یک حسگر تشخیص آلودگی محلی نمیتواند وجود آلودگی در خانه‌های دیگر را تشخیص دهد و یک **تاکسی خودکار** نمیتواند فکر راننده‌های دیگر را بخواند.

قطعی[۱] درمقایسه با غیرقطعی[۲]

اگر حالت بعدی محیط به طور کامل براساس حالت جاری و کنشهای انتخاب شده توسط عامل مشخص شوند آنگاه محیط قطعی است در غیر اینصورت غیرقطعی است. اصولاً یک عامل در یک محیط کاملا مشاهده‌پذیر و قطعی نیازی به نگرانی درباره عدم قطعیت ندارد. هرچند اگر محیط **پاره‌مشاهده‌پذیر** باشد، آنگاه ممکن است غیرقطعی نیز باشد. این امر خصوصاً وقتی درست است که محیط پیچیده باشد و ردگیری همه جوانب غیر قابل دسترس را دشوار کند. بنابراین، معمولاً بهتر است که قطعی یا غیرقطعی بودن محیط را از دید عامل بررسی کنیم. بنابراین رانندگی تاکسی به وضوح غیرقطعی است چرا که یک نفر هرگز نمیتواند به طور دقیق رفتار ترافیک را پیشبینی کند. علاوه بر این ممکن است به‌طور ناگهانی و بدون هیچ علائمی لاستیک‌ها پنچر و یا موتور داغ کند. جهان جاروبرقی همانطور که گفتیم قطعیست. اما انواع دیگر آن میتوانند شامل عناصر تصادفی، مانند پدیدار شدن تصادفی آلودگی و یا سیستم مکش غیرقابل اعتماد را شامل شوند (تمرین ۲-۱۲). اگر محیط قطعی بوده اما کنش عامل‌های دیگر، مشخص و قطعی نباشد میگوییم محیط استراتژیک است. (مثل بازی شطرنج که بازی حریف از قبل قابل پیش‌بینی و قطعی نیست).

رویدادی[۳] درمقایسه با ترتیبی[۴]

در یک محیط رویدادی تجربیات عامل به رویدادهای اتمیک تقسیم می‌شوند. هر رویداد شامل یک سری ادراکات و سپس یک کنش (توسط عامل) خواهد بود. از همه مهم‌تر اینکه رویداد بعدی به کنشهای دوره‌های قبل وابسته نیست. در محیطهای رویدادی انتخاب کنش در هر رویداد تنها به همان رویداد وابسته است. وظایفی همچون دسته‌بندی، رویدادی هستند. به‌طور مثال عاملی که باید یک قطعه معیوب در یک خط تولید را بیابد هر تصمیمی را براساس قطعه کنونی و بدون توجه به تصمیمات قبلی میگیرد. به علاوه تصمیم کنونی معیوب بودن قطعه بعدی را تحت تأثیر قرار نمیدهد. در مقابل در محیطهای ترتیبی تصمیم کنونی تمام تصمیمات بعدی را تحت تأثیر قرار میدهد. شطرنج و راندن تاکسی ترتیبی هستند: [در هر دو مورد کنشهای کوتاه مدت میتواند

۱-Deterministic
۲-stochastic
۳- ترجمه واژه، episodic که گاهی "واقعه‌ای" نیز گفته می‌شود.
۴- sequential . کلمه ترتیبی در علم کامپیوتر به عنوان متضاد موازی نیز به کار می‌رود. این دو معنا هیچ ارتباطی با یکدیگر ندارند-

پیامدهای بلندمدت داشته‌باشد. محیط‌های رویدادی خیلی ساده‌تر هستند چرا که عامل نیازی به دوراندیشی ندارد.]

ایستا[۱] درمقایسه با پویا[۲]

اگر محیط در مدت زمان اندیشیدن عامل تغییر کند آنگاه محیط مورد نظر برای آن عامل پویاست؛ و در غیر اینصورت ایستاست. کار کردن با محیط‌های ایستا آسان‌تر است چرا که عامل هنگام تصمیم‌گیری نیازی به نگاه کردن به محیط ندارد و بعلاوه اینکه نگران گذشت زمان نیست. در مقابل، محیطهای پویا به طور مداوم از عامل میپرسند که چه میخواهد بکند؛ و اگر هنوز تصمیم نگرفته باشد این گونه تلقی میشود که تصمیم گرفته کاری انجام ندهد. اگر محیط با گذشت زمان تغییر نکند ولی امتیاز کارایی عامل تغییرکند، آنگاه محیط نیمه پویاست. راندن تاکسی به وضوح پویاست: اتومبیلهای دیگر و خود تاکسی هنگام تصمیم‌گیری به حرکت خود ادامه می‌دهند. بازی شطرنج درصورتی که فاکتور زمان در آن لحاظ شود، نیمه پویاست. جدول کلمات متقاطع ایستاست.

گسسته[۳] درمقایسه با پیوسته[۴]

تمایز گسستگی و پیوستگی میتواند بر حالت محیط،روش به کارگیری زمان و ادراکات و کنشهای عامل اعمال شود. برای مثال یک محیط گسسته مانند بازی شطرنج تعدادی حالت گسسته متناهی دارد. همچنین شطرنج مجموعه‌ای از کنش‌ها و ادراکات گسسته دارد. راندن تاکسی یک مسئله زمان-پیوسته و حالت-پیوسته است: سرعت و موقعیت تاکسی و سایر وسایل نقلیه پیوسته هستند (زاویه‌ی چرخش فرمان و...). ورودیهای دوربینهای دیجیتال اگرچه به طور سختگیرانه گسسته هستند ولی معمولاً با آنها به صورت نمایشگرهای پیوسته برخورد می‌شود.

تک عاملی درمقایسه با چند عاملی

فرق بین محیطهای تک عاملی و چند عاملی ممکن است ساده به نظر بیاید. برای مثال عاملی که یک جدول کلمات متقاطع را به تنهایی حل میکند به روشنی در یک محیط تک عاملی است در حالیکه عاملی که شطرنج بازی می‌کند در یک محیط دو عاملی قرار دارد. با این حال نکات ظریفی نیز وجود دارد. اول اینکه، می‌دانیم که چگونه یک "موجودیت" ممکن است به عنوان عامل درنظر گرفته شود ولی توضیح نداده‌ایم که چه موجودیت-هایی باید به عنوان عامل تلقی شوند. آیا یک عامل A مثلاً راننده تاکسی باید با شیء B وسیله‌ی نقلیه دیگر به عنوان یک عامل رفتار کند یا میتواند مشابه امواج در ساحل یا حرکت برگها در باد تنها به صورت یک جسم بدون حس با آن برخورد کند؟

۱-Static
۲-Dynamic
۳-Discrete
۴-Continvous

فرق کلیدی این است که آیا بهترین توصیف برای رفتار B، بیشینه ساختن میزان کارایی‌ست که ارزش آن به رفتار عامل A وابسته است؟ برای مثال در شطرنج موجودیت B تلاش می‌کند میزان کارایی خود را بیشینه کند که براساس قوانین شطرنج میزان کارایی عامل A را به حداقل میرساند. بنابراین شطرنج یک محیط چند عاملی رقابتی است. در مقابل در محیط راندن تاکسی اجتناب از تصادف‌ها میزان کارایی تمام عامل‌ها را بیشینه میکند بنابراین این محیط به نوعی یک محیط چند عاملی مشارکتی‌ست. همچنین این محیط تا قسمتی نیز رقابتی‌ست. چون برای مثال تنها یک اتومبیل میتواند یک فضای پارک را اشغال کند. مسائل طراحی عامل در محیطهای چندگانه غالبا کاملاً با آنهایی که در محیطهای تک عاملی به وجود می‌آیند متفاوتند. به‌عنوان مثال، ارتباطات و تعاملات (بینی عامل‌ها) اغلب به عنوان یک رفتار عقلایی در محیطهای چند عاملی درنظر گرفته می‌شوند. همچنین در برخی از محیطهای پاره‌مشاهده‌پذیر رقابتی ، رفتار غیرقطعی عقلانی است زیرا رفتاری غیرقابل پیش‌بینی از خود بروز می‌دهد. (و رقیبان او نمی‌توانند رفتار او را پیش‌بینی نمایند).

محیط عملیاتی	قابلیت مشاهده	قطعی	رویدادی بودن	ایستایی	گسسته	عامل‌ها
جدول کلمات متقاطع	کامل	قطعی	ترتیبی	ایستا	گسسته	تک‌عامله
شطرنج با ساعت	کامل	استراتژیک	ترتیبی	نیمه پویا	گسسته	چندعامله
پوکر	پاره	غیرقطعی	ترتیبی	ایستا	گسسته	چندعامله
تخته	کامل	غیرقطعی	ترتیبی	ایستا	گسسته	چندعامله
راندن تاکسی	پاره	غیرقطعی	ترتیبی	پویا	پیوسته	چندعامله
سیستم تشخیص علائم بیماری	پاره	غیرقطعی	ترتیبی	پویا	پیوسته	تک‌عامله
سیستم تحلیل تصویر	کامل	قطعی	رویدادی	نیمه پویا	پیوسته	تک‌عامله
روبات قطعه جمع کن	پاره	غیرقطعی	رویدادی	پویا	پیوسته	تک‌عامله
کنترل کننده پالایشگاه	پاره	غیرقطعی	ترتیبی	پویا	پیوسته	تک‌عامله
معلم انگلیسی محاوره‌ای	پاره	غیرقطعی	ترتیبی	پویا	گسسته	چندعامله

شکل ۲-۶ مثال‌هایی از محیط‌های عملیاتی و ویژگی‌های آنها.

همان‌طور که انتظار می‌رود، سخت ترین محیط، پاره‌مشاهدپذیر، اتفاقی، ترتیبی، پویا، پیوسته و چند عامله است. خواهیم دید که انواع مختلف محیط‌ها برنامه‌های عامل نسبتاً متفاوتی نیاز دارند. اکثر حالتهای واقعی آنقدر پیچیده‌اند که تشخیص قطعی بودن آن‌ها خود نکته‌ای قابل بحث است؛ لذا برای کاربردهای عملی، باید با آنها به‌صورت غیر قطعی رفتار نمود.

در شکل ۲-۶ ویژگیهای تعدادی از محیطهای آشنا مشخص شده است. توجه کنید که جوابها همواره دقیق نیستند. به عنوان مثال در شکل ۲-۶ شطرنج به‌عنوان محیطی کاملاً مشاهده پذیر فرض شده است درصورتی که اشتباه است چرا که قوانینی همچون "قلعه رفتن" احتیاج به پیشینه‌ای درباره بازی دارد که به عنوان بخشی از وضعیت صفحه قابل مشاهده نیست. البته این استثناها در مقایسه با آنچه راننده تاکسی، معلم انگلیسی یا سیستم تشخیص پزشکی با آن مواجه‌اند ناچیز هستند.

برخی دیگر از پاسخها در جدول مذکور، بستگی به نوع تعریف محیط دارند. به‌عنوان مثال، وظیفه تشخیص پزشکی را به عنوان محیطی تک عاملی طبقه‌بندی کرده‌ایم زیرا روند بیماری در یک بیمار،به عنوان یک عامل

مدلسازی نمیشود. اما یک سیستم تشخیص پزشکی همچنین ممکن است با بیماران سرسخت و پرسنل بی-اعتماد (نسبت به سیستم) سر و کار داشته باشد. بنابراین این محیط میتواند چند عاملی باشد. به علاوه تشخیص پزشکی در صورتی رویدادی است، که آن را به عنوان تشخیص بیماری از روی لیستی از علائم داده شده در نظر بگیریم؛ از دیگر سو تشخیص پزشکی می‌تواند ترتیبی باشد اگر شامل تست و ارزیابی پیشرفت دوره درمان باشد. همچنین بسیاری از محیط‌ها در رده‌های بالاتر ترتیبی هستند. برای مثال یک دوره مسابقات شطرنج از یک سری بازیهای متوالی تشکیل شده‌است. هر بازی یک رویداد است، زیرا(رویهم رفته)سهم حرکتهای یک بازی در کارایی کلی عامل متاثر از حرکتهای بازی قبلی نمی باشد. در مقابل تصمیم‌گیری در طول یک بازی قطعا ترتیبی است.

مخزن برنامه مرتبط با این کتاب (aima.cs.berkeley.edu) شامل پیاده‌سازی تعدادی محیط همراه با یک شبیه‌ساز محیط است که یک یا چند عامل را در یک محیط شبیه‌سازی شده قرار داده و رفتارشان را در طول زمان مشاهده می‌کند و آنها را مطابق معیار کارایی ارزیابی می‌کند. این آزمایشات معمولا نه فقط برای یک محیط بلکه برای تعداد زیادی محیط که از یک **کلاس از** محیط‌ها استخراج شده‌اند، انجام میشود. برای مثال برای ارزیابی یک راننده تاکسی در یک ترافیک شبیه‌سازی شده نیازمند شبیه‌سازی‌های بسیاری تحت شرایط مختلف ترافیک، روشنایی و آب و هوا هستیم. اگر عامل را برای یک سناریوی خاص طراحی کنیم آنگاه می‌توانیم از ویژگی‌های خاص آن در طراحی استفاده کنیم اما طرح ما جامعیت لازم را نخواهد داشت. به این دلیل برنامه تولیدکننده محیط، محیط‌های مختلفی را (با احتمال‌های مختلفی) برروی عامل آزمایش می‌کند. برای مثال تولیدکننده محیط برای جاروبرقی الگوی اولیه آلودگی و مکان عامل را به‌طور تصادفی تعیین می‌کند. درواقع، کارایی متوسط عامل در یک کلاس از محیط‌ها برای ما حائز اهمیت است. یک عامل عقلایی برای یک دسته محیط داده شده این کارایی متوسط را بیشینه میکند. تمرین‌های ۲-۷ تا ۲-۱۲ شما را با روند گسترش یک کلاس از محیط‌ها و ارزیابی عامل‌های مختلف در آن آشنا می کنند.

۲-۴- ساختار عامل‌ها

تاکنون درباره عامل‌ها و تشریح رفتارشان (عملی را که بعد از هر دنباله ادراکی انجام میدهند) صحبت کردیم. حال، می خواهیم به اصل مطلب بپردازیم و راجع به وظیفه داخلی آن‌ها صحبت کنیم. وظیفه هوش‌مصنوعی، طراحی‌برنامه عامل است. برنامه عامل، تابع عامل (نگاشت ادراکات به کنش‌ها) را پیاده‌سازی می‌کند. این برنامه برروی یک ابزار محاسباتی بهمراه حسگرها و محرک‌ها اجرا می‌شود که به این مجموعه معماری گفته می‌شود.

بنابراین می‌توان گفت که عامل = معماری + برنامه.

واضح است، برنامه‌ای که انتخاب می کنیم باید برای معماری مناسب باشد. اگر برنامه قرار است کنشی مثل "*راه برو*" را پیشنهاد دهد معماری باید پا داشته باشد. معماری ممکن است یک کامپیوتر ساده و یا یک ماشین روبوتیک شامل چندین کامپیوتر، دوربین و حسگر باشد. به طورکلی معماری، ادراکات را از طریق حسگرها در دسترس برنامه قرار می دهد، برنامه را اجرا میکند و به محرک‌ها از طریق کنش‌های تولید شده فرمان می‌دهد.

البته همیشه معماری به مفهوم یک سخت‌افزار نیست و خود معماری میتواند یک نرم‌افزار دیگر باشد. اگرچه بیشتر مباحث این کتاب راجع به طراحی برنامه‌های عامل است.

۲-۴-۱- برنامه‌های عامل

تمامی عامل‌های هوشمندی که در این کتاب طراحی خواهیم کرد اسکلت مشابهی دارند. یعنی، ادراکات را از محیط پذیرفته و کنش‌هایی را تولید می کنند.[۱] به تفاوت بین برنامه عاملی که ادراک جاری را به عنوان ورودی میگیرد و تابع عاملی که کل تاریخچه ادراکات را میگیرد توجه کنید. برنامه عامل تنها ادراک جاری را به عنوان ورودی میگیرد چرا که چیز دیگری در مورد محیط در دسترس نیست؛ اگر کنش‌های عامل به کل دنباله ادراکات وابسته باشند، عامل به ناچار باید ادراکات را به خاطر بسپارد.

برای مثال، طبق شکل ۲-۷ یک برنامه عامل رد[۲] دنباله‌ی ادراکات را گرفته و سپس از آن برای اندیس‌گذاری جدولی از کنشها برای تصمیم‌گیری در مورد عمل بعدی استفاده می‌کند. برای ساخت یک عامل عقلایی به این روش، باید جدولی بسازیم که شامل کنش مناسب برای هر دنباله‌ای از ادراکات ممکن باشد.

توجه به اینکه چرا روش‌های مبتنی بر جدول (برای طراحی عامل) محکوم به شکست است، بسیار آموزنده است. اجازه دهید P مجموعه‌ای از ادراکات ممکن و T عمر عامل (کل تعداد ادراک‌هایی که دریافت خواهد کرد.) باشد. جدول جستجو شامل مدخل خواهد بود. تاکسی خودکار را در نظر بگیرید: ورودی یک تصویر از یک دوربین با سرعت ۲۷ مگابایت در ثانیه (۳۰ فریم در ثانیه، ۶۴۰×۴۸۰ پیکسل با ۲۴ بیت برای اطلاعات رنگ) است که مدخل‌های جدول را به تعداد $10^{250000000000}$ (برای یک ساعت رانندگی) می‌رساند. حتی جدول جستجو برای شطرنج (بخش کوچک و خوش رفتاری از جهان واقعی)حداقل 10^{150} مدخل خواهد داشت. اندازه‌ی این جدولها (تعداد اتم‌ها در جهان قابل مشاهده 10^{80} است) بدین معنی‌ست که الف: هیچ عامل فیزیکی در جهان قدرت ذخیره‌ی این حجم از اطلاعات را نخواهد داشت، ب: طراح، زمان لازم برای طراحی و ایجاد این جدول را نخواهد داشت، پ: هیچ عاملی تمام مداخل جدول را (از روی تجربه) نخواهد آموخت و ت: حتی اگر محیط به اندازه‌ای ساده باشد که اندازه‌ی جدول معقول باشد، طراح هنوز هیچ ایده‌ای در مورد چگونگی پر کردن مداخل جدول ارائه نکرده است.

برخلاف همه‌ی این‌ها، عامل مبتنی بر جدول آنچه که ما میخواهیم را انجام میدهد: تابع عامل مطلوب را پیاده‌سازی میکند. چالش اصلی در AI یافتن روشی برای نوشتن برنامه‌ای با رفتار عقلایی است که این رفتار از مقدار کوچکی (تا حد امکان) بدست آمده باشد (به جای تعداد زیادی از مدخل‌های یک جدول). نمونه‌های موفقی در این زمینه داریم: برای مثال جدولهای بزرگ ریشه‌ی دوم که توسط مهندسان و دانش‌آموزان قبل از دهه‌ی ۷۰ استفاده می‌شد امروزه با برنامه‌ای ۵ خطی براساس روش نیوتن که روی ماشین حسابهای الکتریکی

---

[۱] - گزینه‌های دیگری برای اسکلت‌بندی برنامه‌ی عامل وجود دارند. برای مثال میتوانستند برنامه‌هایی باشند که به طور غیر همزمان با محیط اجرا می‌شوند. هر کدام از این برنامه‌ها یک ورودی و یک خروجی دارد و شامل یک حلقه‌ای است که ورودی را از پورت به عنوان ادراک خوانده و کنشها را بر روی پورت خروجی می‌نویسد.

[۲] - در اینجا "رد" ترجمه فارسی کلمه انگلیسی trace است. البته برخی جاها نیز ترجمه فارسی کلمه reject به کار رفته است که از محتوای متن قابل تشخیص است.

اجرا میشود جایگزین شده‌است. سؤال این است که آیا AI میتواند برای رفتار عمومی هوشمندانه کاری را که نیوتن برای ریشه‌ی دوم انجام داد بکند یا نه؟ ما معتقدیم جواب مثبت است.

```
function Table-Driven-Agent(percept) returns an action
  Static: percepts, a sequence, initially empty
          table.a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percept
  action ← LOOKUP (percepts, table)
  return action
```

شکل ۲-۷ برنامه‌ی عامل مبتنی بر جدول که با هر ادراک جدید فراخوانده شده و هر بار یک خروجی برمی‌گرداند. رد رشته‌ی ادراکات را در ساختار داده‌ی اختصاصی خود ثبت می‌کند.

در ادامه‌ی این بخش چهار نوع برنامه عامل پایه‌ای را معرفی خواهیم کرد که اصول تشکیل دهنده همه سیستم‌های هوشمند را در بر دارند.

- عامل‌های بازتابی ساده
- عامل‌های بازتابی مبتنی بر مدل
- عامل‌های مبتنی بر هدف
- عامل‌های مبتنی بر سود

```
function Reflex-Vacuum-Agent ([location, status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

شکل ۲-۸ برنامه‌ی عامل برای عامل بازتابی ساده در دو حالت محیط جاروبرقی. این برنامه، برنامه‌ی عامل جدول شکل ۲-۳ را پیاده‌سازی میکند.

در ادامه درباره چگونگی طراحی عامل‌های یادگیرنده نیز صحبت خواهیم کرد.

۲-۴-۲- عامل‌های بازتابی ساده[1]

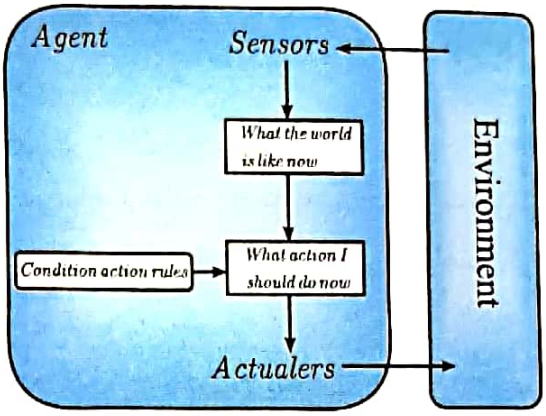
ساده‌ترین نوع عامل، عامل بازتابی ساده است. این عامل‌ها کنش‌های خود را بر پایه‌ی ادراک جاری انتخاب می‌کنند و تاریخچه‌ی ادراک‌ها را نادیده می‌گیرند. برای مثال، عامل جاروبرقی که تابع عامل آن در جدول شکل ۲-۳ نشان داده شده است، عامل بازتابی ساده است، زیرا تصمیم‌گیری آن تنها بر پایه‌ی مکان کنونی و آلوده بودن یا نبودن آن انتخاب می‌شود. یک برنامه‌ی عامل برای این عامل در شکل ۲-۸ نشان داده شده‌است.

دقت کنید که برنامه عامل جاروبرقی در مقایسه با جدول متناظر با آن، بسیار کوچکتر است. علت اصلی این کاهش ناشی از نادیده گرفتن تاریخچه ادراکات است که تعداد حالات محتمل را از 4^T به ۴ کاهش میدهد. دلیل دیگر ناشی از این واقعیت است که هنگامی که خانه‌ی کنونی کثیف است، کنش مستقل از مکان عامل است.

[1]-simple reflex agents

خودتان را به جای راننده تاکسی خودکار تصور کنید. اگر ماشین جلویی ترمز کند و چراغ ترمزش روشن شود، شما باید متوجه آن شده و ترمز را آغاز کنید. به عبارت دیگر، فرایندی بر روی ورودی دیداری به منظور تشخیص شرایطی مثل "ماشین جلویی در حال ترمز است" انجام شده است. سپس این ورودی ارتباطی را در برنامه عامل به کنش "ترمز را آغاز کن" فعال میکند. این نوع ارتباط را قانون کنش شرطی می‌نامیم که به صورت زیر نوشته می‌شود:

اگر ماشین جلویی در حال ترمز است آنگاه ترمز را آغاز کن.



شکل ۲-۹ دیاگرام طرح یک عامل بازتابی ساده.

```
function SIMPLE-REFLEX-AGENT (percept) returns an action
    static: rules, a set of condition-action rules

    state←INTERPRET-INPUT(percept)
    rule←RULE-MATCH(state, rules)
    action ←RULE-ACTION[rule]
    return action
```

شکل ۲-۱۰ یک عامل بازتابی ساده. درصورتی که شرایط آن با حالت جاری مطابقت کند آنگاه کنش مربوط به آن اجرا می‌شود.

انسان‌ها نیز چنین ارتباطاتی دارند که بعضی از آنها واکنش‌های اکتسابی (مثل رانندگی) و بعضی از آنها واکنشهای ذاتی هستند (مثل پلک زدن زمانی که چیزی به طرف چشم می‌آید). در ادامه کتاب، چندین راه مختلف برای یادگیری و پیاده‌سازی چنین ارتباطاتی خواهیم دید.

برنامه شکل۲-۸ مربوط به محیط نوع خاصی از یک جاروبرقی است. یک رویکرد کلی‌تر و انعطاف‌پذیرتر، ساخت یک مفسر عام‌منظوره برای قانون کنش شرطی و سپس ایجاد مجموعه‌ای از قوانین مخصوص برای محیط وظیفه‌ای است. شکل ۲-۹ ساختار این برنامه را در حالت کلی نشان می‌دهد. این ساختار چگونی برقراری ارتباط بین ادراک‌ها و کنش‌ها (از طریق قوانین کنش شرطی) را به خوبی نمایش می‌دهد. از مستطیل‌ها برای نشان دادن حالت درونی و جاری فرآیند تصمیم‌گیری عامل و از بیضی‌ها برای نمایش اطلاعات استفاده شده در فرایند تصمیم‌گیری استفاده می‌کنیم. برنامه‌ی عامل نیز که بسیار ساده می‌باشد در شکل ۲-۱۰ نشان داده شده است. تابع INTERPRET-INPUT یک توصیف انتزاعی از حالت جاری ادراکات تولید می‌کند و تابع RULE-MATCH

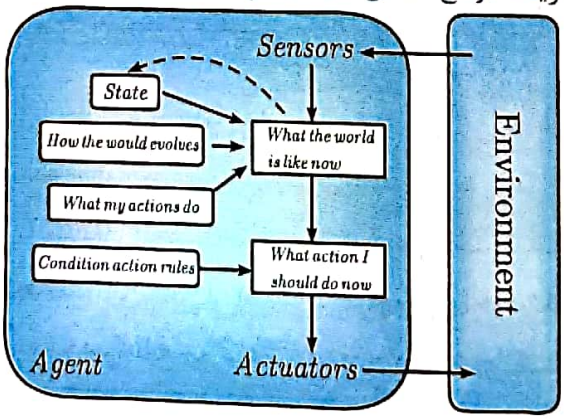
اولین قانون در مجموعه قوانینی که مطابق با توصیف حالت داده شده است را بر می‌گرداند. پیاده‌سازی واقعی می‌تواند به سادگی مجموعه‌ای از گیت‌های منطقی که یک مدار بولی را پیاده‌سازی می‌کنند باشد.

مزیت عامل‌های بازتابی ساده، سادگی آن‌هاست ولی از هوش محدودی برخوردارند. عامل بازتابی ساده (در شکل ۲-۱۰) تنها در صورتی به درستی عمل می‌کند که تصمیم جاری تنها بر پایه‌ی ادراک جاری(اگر محیط کاملاً قابل مشاهده باشد) قابل نتیجه‌گیری باشد. حتی مقدار اندکی مشاهده‌ناپذیری میتواند مشکل بزرگی ایجاد کند. برای مثال، قانون ترمز (که پیشتر ارائه شد) فرض میکند که شرایط ماشین جلویی (که در حال ترمز است) توسط ادراک جاری (تصویر ویدئویی جاری) قابل درک است. اما در این میان ماشین‌ها قدیمی وجود دارند که تنظیمات چراغ ترمز و راهنمای متفاوتی دارند و تشخیص ماشین در حال ترمز از روی یک عکس به آسانی ممکن نیست. یک عامل بازتابی ساده که پشت چنین ماشینی حرکت میکند یا به طور ادامه‌دار و به دفعات ترمز خواهد کرد و یا اینکه هیچگاه ترمز نخواهد کرد.

مشکل مشابهی را در جهان جاروبرقی مشاهده می‌کنیم. فرض کنید یک عامل جاروبرقی بازتابی ساده حسگر مکان ندارد و فقط یک حسگر آلودگی دارد. چنین عاملی فقط دو ادراک مجاز دارد: کثیف و تمیز. در پاسخ به کثیف بودن میتواند عمل مکش را انجام دهد ولی در پاسخ به تمیز بودن چه کاری انجام می‌دهد؟ حرکت به راست در صورت اینکه در خانه‌ی A باشد برای همیشه شکست می‌خورد و حرکت به چپ در صورتی که در خانه‌ی B باشد شکست می‌خورد. حلقه‌ی بی‌نهایت در عامل‌های بازتابی ساده‌ای که در محیطهای پاره‌مشاهده‌پذیر عمل میکنند غیر قابل اجتناب است.

اگر عامل کنشی را به صورت تصادفی انتخاب کند خروج از حلقه‌ی بینهایت امکان پذیر است. برای مثال، اگر عامل جاروبرقی تمیز بودن را ادراک کند ممکن است با پرتاب سکه تصمیم بگیرد که به راست یا به چپ برود. به سادگی میتوان نشان داد که عامل به طور متوسط در دو مرحله به خانه‌ی بعدی رفته و اگر کثیف باشد آن را تمیز خواهد کرد و کار تمیز کردن به پایان خواهد رسید. بنابراین ممکن است (تحت شرایطی) عامل بازتابی ساده‌ای که از انتخابهای تصادفی بهره می‌گیرد از عامل بازتابی ساده قطعی بهتر عمل کند.

در بخش قبلی اشاره کردیم که رفتار تصادفی از نوع درست میتواند در برخی محیطهای چند عامله عقلایی باشد. در محیطهای تک عامله رفتار تصادفی معمولاً عقلایی نیست. تصادفی بودن در بعضی از موارد به عامل ساده کمک شایانی می‌کند ولی در اکثریت مواقع، عامل‌های قطعی عملکرد بهتری دارند.



شکل ۲-۱۱ عامل بازتابی مبتنی بر مدل

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
  static: state, a description of the current world state
          rules, a set of condition-action rules
          action, the most recent action, initially none

  state←UPDATE-STATE(state, action, percept)
  rule←RULE-MATCH(state, rules)
  action ←RULE-ACTION[rule]
  return action
```

شکل ۲-۱۲ یک عامل بازتابی مبتنی بر مدل که رد حالت کنونی جهان را با استفاده از یک مدل داخلی حفظ می‌کند. این عامل کنش خود را مثل عامل بازتابی ساده انتخاب میکند.

۲-۴-۳- عامل‌های[۱] بازتابی مبتنی بر مدل

تأثیرگذارترین راه برای برخورد با محیطهای پاره‌مشاهده‌پذیر این است که عامل ردی از بخشی از دنیا را که نمیتواند ببیند در درون خود نگه دارد. یعنی عامل باید نوعی از حالت درونیْ را در خود نگهداری کند که به تاریخچه ادراکات وابسته است و جنبه‌های غیرقابل مشاهده حالت جاری را داراست. برای مثال، در مسئله ترمز ماشین، حالت درونی چندان گسترده نیست (فریم قبلی دوربین برای تشخیص زمان روشن شدن چراغ ترمز). برای کارهای دیگری مثل عوض کردن مسیر، اگر عامل نتواند بقیه ماشین‌ها را یکجا ببیند نیازمند نگهداری اطلاعات مکانی آن‌ها به صورت درونی است.

برای بروز رسانی این حالات درونی (در طول زمان) نیازمند به‌کارگیری دو دانش در برنامه عامل هستیم. نخست، به اطلاعاتی در مورد چگونگی تکامل دنیا (مستقل از عامل) نیازمندیم. به عنوان مثال اتومبیل در حالت سبقت با گذشت زمان به اتومبیل ما نزدیک‌تر خواهد شد. دوم، به اطلاعاتی در مورد چگونگی تاثیرگذاری کنش‌های عامل بر دنیا نیاز داریم. به عنوان مثال، اینکه وقتی عامل مسیرش را به سمت راست تغییر می‌دهد، یک جای خالی(هرچند به طور موقت) در مسیر قبلی آن به وجود می‌آید یا اینکه بعد از پنج دقیقه رانندگی در جهت شمال به نقطه‌ای در ۵ مایلی شمال مکان قبلی می‌رسیم. به این اطلاعات در مورد چگونگی کارکرد جهان(چه توسط مدارهای بولی و چه توسط نظریه‌های علمی پیچیده پیاده‌سازی شده باشد) مدل جهان می‌گوییم. عاملی که از این مدل استفاده میکند، عامل مبتنی بر مدل نامیده می‌شود.

شکل۲-۱۱ ساختار عامل بازتابی با حالت درونی را نمایش می‌دهد. در این ساختار، ادراک جاری با حالت درونی قبلی ترکیب و توصیف بهنگام شده حالت جاری را تولید می‌کند. برنامه عامل در شکل ۲-۱۲ نشان داده شده است. تابع UPDATE-STATE مسئول ایجاد توصیف حالت درونی جدید است. علاوه بر تفسیر ادراک جدید به کمک دانش موجود در موردحالات، این تابع از اطلاعات مربوط به نحوه تکامل دنیا برای ردگیری بخشهای دیده نشده دنیا استفاده می‌کند. همچنین تابع مورد نظر باید در مورد تاثیر کنش‌ها برروی حالت دنیا نیز آگاهی داشته باشد.

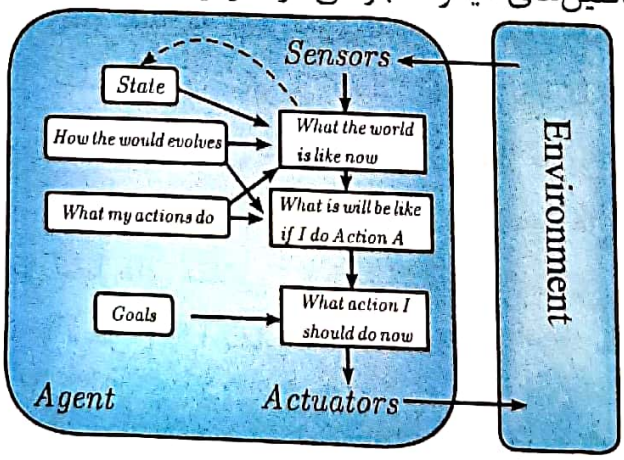
۱- Model – based reflex agents : گاهی به آن انعکاس نیز می‌گویند.

۲-۴-۴- عامل‌های[۱] مبتنی بر هدف

حالت جاری محیط همیشه برای تصمیم‌گیری کافی نیست. به عنوان مثال، در تقاطع یک خیابان، تاکسی می‌تواند به چپ یا راست بپیچد یا مستقیم برود اما تصمیم درست بستگی به مقصد تاکسی دارد. به عبارت دیگر علاوه بر تعریفی ازحالت جاری، عامل به اطلاعاتی درمورد هدف که توصیف‌کننده موقعیت‌های مطلوب‌اند نیازمند است(به عنوان مثال، قرار داشتن در مسیر مقصد مسافر). برنامه عامل می‌تواند این اطلاعات را با اطلاعات مربوط به نتایج کنشهای ممکن (اطلاعاتی که برای بهنگام‌سازی حالت درونی در عامل بازتابی به کار می‌رفت) ترکیب و کنشهایی را که به هدف منتهی می‌شوند را انتخاب کند. شکل ۲-۱۳ ساختار عامل مبتنی بر هدف را نشان میدهد.

گاهی اوقات انتخاب کنش مبتنی بر هدف کار ساده‌ای است (زمانی که ارضای هدف در اثر اجرای یک کنش تنها حاصل می‌شود) و گاهی اوقات کمی دشوار است(زمانی که عامل مجبور است دنباله‌های درازی از پیچها و گردش‌ها را برای پیدا کردن مسیری در جهت دستیابی به هدف در نظر بگیرد). جستجو (فصلهای ۳تا۴) و طرح‌ریزی (فصل ۱۱) زیرشاخه‌هایی از هوش‌مصنوعی هستند که برای یافتن دنباله کنشهایی که به اهداف عامل منتهی می‌شوند، تلاش می‌کنند.

توجه داشته باشید که تصمیم‌گیری از این نوع اساساً با قوانین کنش شرطی که قبلاً توصیف شدند متفاوت است (از این نظر که باید آینده را نیز در نظرگرفت ـ "اگر من اینگونه و آنگونه عمل کنم چه اتفاقی خواهد افتاد ؟"و" آیا مرا کامیاب خواهد کرد؟"). در طراحی عامل‌های بازتابی، این اطلاعات صریحاً مورد استفاده قرار نمی‌گیرند، چرا که طراح از قبل کنشهای صحیح را برای حالات مختلف محاسبه می‌کند. عامل بازتابی هنگام دیدن روشن شدن چراغ ترمز، ترمز می‌کند. از دیگر سو، یک عامل مبتنی بر هدف، با دیدن روشن شدن چراغ ترمز، کم شدن سرعت اتومبیل جلویی را استدلال می‌کند. براساس نوع قوانین دنیا، تنها کنشی که به هدف یعنی عدم برخورد با ماشین‌های دیگر منجر می‌شود ترمز است.



شکل ۲-۱۳ یک عامل مبتنی بر هدف . مدل رد حالات دنیا به همراه مجموعه‌ای از اهداف که برای رسیدن به آنها تلاش میکند را نگه‌داری میکند و کنش خود را در راستای رسیدن به اهداف انتخاب میکند.

عامل مبتنی بر هدف کارایی کمتر اما انعطاف‌پذیری بیش‌تری دارد. اگر باران شروع به باریدن کند، عامل می‌تواند دانش مرتبط با اثرگذاری عملکرد ترمزهایش را بهنگام کند. این امر منجر به تطابق خودکار تمامی رفتارهای

[۱]- Goal-based agents

عامل با شرایط جدید می‌شود. از دیگر سو، در عامل بازتابی لازم بود که ما تعداد زیادی از قوانین کنش شرطی را بازنویسی کنیم. امّا، عامل مبتنی بر هدف از نظر رسیدن به مقصدهای مختلف انعطاف‌پذیرتر است. صرفا با مشخص کردن یک مقصد جدید، ما می‌توانیم رفتار جدیدی را برای عامل مبتنی بر هدف مطرح نماییم. در حالیکه قواعد عامل بازتابی برای اینکه کی بچرخد و کی مستقیم حرکت کند فقط برای یک مقصد به درد می خورند(برای رفتن به یک جای جدید همه آنها باید جایگزین شوند).

۲-۴-۵- عامل‌های[۱] مبتنی بر سود (مطلوبیت)

هدفها به تنهایی برای تولید رفتاری با کیفیت بالا کافی نیستند. به عنوان مثال، دنباله کنشهای زیادی وجود دارند که تاکسی را به مقصدش می‌رسانند و بدین وسیله به هدف دست می‌یابند، اما بعضی از آنها سریعتر، بی-خطرتر، قابل اعتمادتر یا ارزان‌تر از بقیه هستند. اهداف فقط یک تمایز خام بین حالات "کامیاب" و"ناکامیاب" ایجاد می‌کنند. در حالیکه یک معیار عملکرد کلی‌تر باید اجازه یک مقایسه بین حالات مختلف دنیا(یا دنباله ای از حالات) را بدهد؛ بر این اساس که آنها چقدر عامل را کامیاب خواهند کرد. از آنجا که کلمه "کامیاب" چندان علمی به نظر نمی‌رسد، اصطلاح مرسوم آن است که بگوییم اگر یک حالت دنیا بر دیگری ارجحیت داشته باشد، آنگاه دارای مطلوبیت[۲] بالاتری برای عامل خواهد بود.

بنابراین مطلوبیت تابعی است که یک حالت را به یک عددحقیقی می‌نگارد که درجه کامیابی مربوطه را توصیف می‌کند. تعیین کامل مشخصات تابع مطلوبیت، در مواردی که تنها دانستن هدف برای تصمیم‌گیری عقلایی کافی نیست، بسیار کمک‌کننده است. نخست، وقتی که اهداف مغایر وجود دارند و فقط بعضی از آنها قابل دستیابی باشند(به عنوان مثال سرعت و امنیت)؛ در این موارد تابع[۳] مطلوبیت گزینه مناسب‌تر را انتخاب می‌کند. دوم، وقتی که اهداف متعددی وجود داشته باشند و هیچ کدام از آنها به طور قطع قابل دسترسی نباشند. در این صورت مطلوبیت راهی را برای درنظر گرفتن احتمال موفقیت در مقابله با ارزیابی اهمیت اهداف فراهم می‌کند.

هر عامل عقلایی باید به گونه‌ای رفتار کند که انگار دارای تابع مطلوبیت است و تابع مطلوبیت در آن تلاش می‌کند که مقدار مورد انتظار را بیشینه کند. بنابراین عاملی که صریحا دارای تابع مطلوبیت است می‌تواند تصمیمات عقلایی بگیرد. همچنین این کار (گرفتن تصمیمات عقلایی) از طریق الگوریتم عام‌منظوره‌ی مستقل از تابع مطلوبیت نیز امکان‌پذیر است. به این طریق، تعریف کلی عقلانیت(عامل‌هایی که بالاترین کارایی را دارند)به یک محدودیت محلی برای طراحی عامل‌های عقلایی تبدیل می‌شود که ازطریق یک برنامه ساده قابل بیان است. ساختار عامل‌های مبتنی بر سود در شکل ۲-۱۴ نشان داده شده‌است.

۲-۴-۶- عامل‌های یادگیرنده

تا اینجا، برنامه‌های عامل مختلفی با متدهای متفاوتی برای انتخاب و تصمیم‌گیری در مورد کنش‌ها معرفی کرده‌ایم. اما تاکنون شرح نداده‌ایم که برنامه‌های عامل چگونه به وجود می‌آیند. تورینگ (۱۹۵۰) در مقاله اولیه و

---

[۱]-Utility-based agents

[۲]- کلمه‌ی سودمندی (مطلوبیت) به عنوان معادلUtility ، انتخاب گردیده است.

[۳]-Utility function

معروف خود، ایده‌ی برنامه‌ریزی دستی ماشین‌های هوشمند را مورد بررسی قرار داد. او میزان کار مورد نیاز برای این کار را تخمین و به این نتیجه می‌رسد که احتمالا روش‌های بهتری نیز برای این منظور وجود دارد. شیوه‌ای که وی پیشنهاد میکند ساختن ماشینهای یادگیرنده و سپس آموختن به آنهاست. در بسیاری از زمینه‌های هوش مصنوعی این شیوه اکنون شیوه‌ای ارجح برای ساختن سیستم‌های با تکنولوژی روز می‌باشند.

یادگیری مزیت دیگری نیز دارد. همانطور که قبلاًاشاره کردیم: یادگیری به عامل امکان فعالیت در محیطهای (در ابتدا) ناشناخته را داده و او را توانمندتر از حالت اولیه خود (دانش اولیه عامل) می‌سازد. در این بخش ما به طور خلاصه ایده‌های اصلی عامل‌های یادگیرنده را معرفی میکنیم.

همانند شکل ۲-۱۵ یک عامل یادگیرنده به چهار مولفه مفهومی قابل تقسیم است. مهمترین تفاوت در عنصر یادگیرنده[۱] (که مسئول پیشرفت می‌باشد) و عنصر کارایی[۲] (که مسئول انتخاب کنش‌های بیرونی‌ست) وجود دارد. عنصر کارایی چیزی‌ست که ما پیشتر آنرا کل عامل در نظر گرفتیم: ادراک‌هایی را دریافت کرده و در مورد کنش‌ها تصمیم‌گیری می‌کند. عامل یادگیرنده از بازخورد "منتقد[۳]" به‌منظور بهبود عملکرد عامل استفاده می‌کند.

طراحی عامل یادگیرنده به میزان زیادی به طراحی عامل کارایی وابسته است. وقتی میخواهیم عاملی را طراحی کنیم که یک قابلیت خاص را یاد بگیرد سؤال اول این نیست که "چگونه این را به عامل یاد بدهیم؟" بلکه اولین پرسش این است که "عامل من پس از یادگیری به چه نوع عنصر کارایی احتیاج خواهد داشت؟". با فرض داشتن طرح یک عامل، مکانیزمهای یادگیری میتوانند برای بهبود تمام بخشهای عامل، ایجاد شوند.

منتقد باتوجه به یک استاندارد کارایی ثابت به عنصر یادگیرنده میگوید که عامل با چه کیفیتی عمل میکند. وجود منتقد الزامی‌ست چرا که ادراکات به خودی خود هیچ نشانه‌ای از میزان موفقیت عامل را فراهم نمی‌کنند. مثلاً یک برنامه شطرنج ممکن است مفهومی را که نشان می‌دهد حریفش را مات کرده‌است ادراک کند. اما به یک استاندارد کارایی احتیاج دارد تا بداند که این یک کنش خوب است. زیرا مفاهیم به خودی خود چنین چیزی را منتقل نمی‌کنند. علاوه براین استاندار کارایی نیز باید ثابت باشد. درحقیقت باید به آن به عنوان چیزی کاملاً خارج از عامل، نگاه کرد زیرا عامل نباید آنرا اصلاح کند تا با رفتار خودش جور در بیاید.

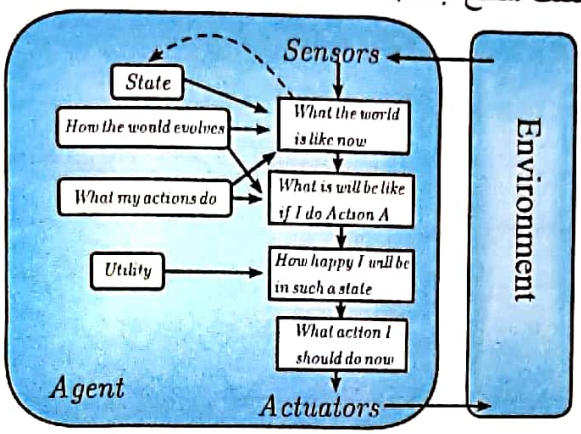
آخرین جزء از یک عامل یادگیرنده، تولیدکننده مسئله است. این جزء مسئول پیشنهاد کنش‌هایی است که به تجارب جدید و آگاهی‌بخش می‌انجامد. نکته اینجاست که اگر عنصر کارایی، راه خود را ادامه دهد انتخاب بهترین کنش‌ها را براساس دانش موجود انجام می‌دهد. اما اگر عامل مایل به کاوش (اکتشاف) باشد و تعدادی کنش غیربهینه (در کوتاه مدت) را امتحان کند آنگاه ممکن است کنش‌های بهتری (در بلند مدت) را کشف کند. کار تولیدکننده همان تولید و پیشنهاد کنش‌های اکتشافی است. این همان کاری است که دانشمندان هنگام انجام آزمایشات می‌کنند.

برای این که طرح کلی را محکم‌تر کنیم، اجازه دهید به مثال راننده‌ی تاکسی برگردیم. عنصر کارایی شامل مجموعه‌ای از دانش و رویه است که تاکسی برای انتخاب کنش رانندگی انتخاب میکند. تاکسی با استفاده از این عنصر کارایی در جاده رانندگی میکند. منتقد، جهان را مشاهده و اطلاعات را به عنصر یادگیری میدهد. برای مثال، فرض کنید راننده تاکسی یک چرخش به راست سریع درمیان خطوط جاده انجام دهد. منتقد رفتار شوکه

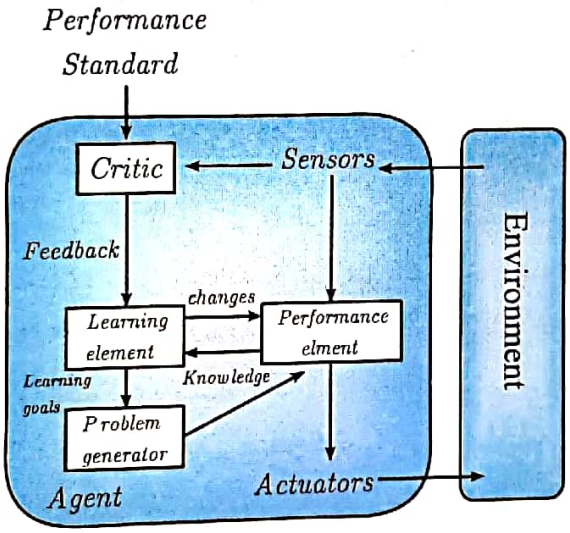
---

[۱]-Learning element
[۲]-Performance element
[۳]-Critic

شده رانندگان دیگر را مشاهده می‌کند. با استفاده از این تجربه، عنصر یادگیری قادر به فرموله‌سازی قانونی است که بر اساس آن این کنش را نادرست تلقی می‌کند و عنصر کارایی را مطابق قانون جدید تغییر می‌دهد. تولید کننده مسئله ممکن است بخش‌های خاصی از رفتار را برای بهبود و پیشنهاد آزمایشات جدید مناسب ببیند (مثل ترمز کردن در شرایط مختلف سطح جاده).



شکل ۲-۱۴ عامل مبتنی بر سود که مدل جهان را به همراه تابع مطلوبیت که ترجیحش را در حالات جهان اندازه می‌گیرد استفاده میکند. سپس کنشی را که به بهترین سود مورد انتظار (که با امید ریاضی نتایج محتمل حالات محاسبه میشود) منتهی میشود، انتخاب میکند.



شکل ۲-۱۵ مدل عمومی عامل یادگیرنده

عنصر یادگیرنده میتواند برروی بخشهای مختلف"دانش" نشان‌داده‌شده در دیاگرام عامل (شکل۲-۹،۲-۱۱،۲-۱۳و۲-۱۴) تغییراتی را اعمال کند. ساده‌ترین مورد شامل یادگیری مستقیم از رشته‌ی ادراک‌هاست. مشاهده زوج حالات پسین محیط به عامل فرصت یادگیری اینکه "جهان چگونه تغییر میکند" را می‌دهد و مشاهده‌ی نتایج کنش‌هایش به عامل فرصت یادگیری"کنشهای من چه اثری دارند" را میدهد. برای مثال اگر تاکسی مقدار خاصی از فشار را برای ترمز کردن در سطح خیس متحمل شود، به زودی درخواهد یافت که سرعتش چقدر کاهش یافته است. به وضوح، این دو فرایند یادگیری در محیط‌های پاره‌مشاهده‌پذیر، دشوارتر می‌شوند.

فرم‌های یادگیری فوق‌الذکر(در پاراگراف قبلی) نیازی به استاندارد کارایی خارجی ندارند (منظور از استاندارد کارایی خارجی، معیار کارایی عمومی است که پیش‌بینی‌هایی منطبق با آزمایشات انجام می‌دهد). موقعیت عامل مبتنی بر

سود (مطلوبیت) که می‌خواهد به یادگیری اطلاعات مطلوبیت بپردازد کمی سخت‌تر است. برای مثال فرض کنید مسافرانی که در طول مسیر تکانهای شدیدی را تحمل کرده‌اند کرایه خود را ندهند. استاندارد کارایی خارجی باید به عامل اطلاع دهد که از دست دادن پول کرایه تأثیر منفی بر کارایی کلی دارد و بنابراین عامل میتواند یاد بگیرد که مانورهای خشن در طول مسیر موجب افزایش بهره‌اش نمی‌شود. در واقع، استاندارد کارایی بخشی از ادراک را به عنوان جایزه (یا تنبیه) (که موجب بازخورد مستقیم بر کیفیت رفتار عامل می‌شود) ارزیابی می‌کند.

به طورخلاصه، عامل یادگیرنده، از بخش‌های مختلفی تشکیل شده است و این بخشها به شکلهای مختلفی در برنامه‌ی عامل قابل نمایش هستند. بااین‌حال، یک کلیت واحد وجود دارد. فرآیند یادگیری در عامل‌های هوشمند به‌اختصار به‌معنی: تغییر بخش‌های مختلف عامل برای نزدیکتر شدن این بخشها به اطلاعات بازخورد (برای بهبود کارایی کلی عامل) است.

۲-۵- خلاصه

نکات اصلی‌ای که باید یادآوری کنیم عبارتند از:

- عامل چیزی است که از محیط اطراف خود ادراک کرده و برروی آن کنش انجام می‌دهد. تابع عامل مشخص-کننده کنشی است که باید عامل در پاسخ به یک رشته از ادراکات از خود نشان دهد.
- معیار کارایی رفتار یک عامل در محیط را می‌سنجد. عامل عقلایی به گونه‌ای عمل می‌کند (با توجه به دنباله ادراکی که دریافت کرده است) که مقدار مورد انتظار معیار کارایی بیشینه شود.
- یک محیط وظیفه‌ای شامل میزان کارایی، محیط خارجی، حسگرها و محرک‌ها است. گام اول در طراحی عامل مشخص کردن کامل این محیط است.
- محیطهای وظیفه‌ای چند بعد اصلی دارند. ممکن است کاملاً یا پاره مشاهده‌پذیر باشند، قطعی یا غیرقطعی باشند، رویدادی یا ترتیبی باشند، ایستا یا پویا، گسسته یا پیوسته، تک عاملی یا چند عاملی باشند.
- برنامه‌ی عامل تابع عامل را پیاده‌سازی میکند. طراحی‌های پایه‌ای زیادی برای برنامه‌ی عامل وجود دارد که نوع اطلاعاتی که در تصمیم‌گیری به‌کار گرفته شده‌اند را مشخص میکند. طراحی‌های مختلف در کارایی، فشردگی و انعطاف‌پذیری باهم تفاوت دارند. طراحی مناسب برای عامل به نوع محیط بستگی دارد.
- عامل‌های بازتابی ساده بلافاصله به ادراکات پاسخ می‌دهند؛ عامل‌های بازتابی مبتنی بر مدل، رد آن جنبه از جهان را که در حال حاضر قابل مشاهده نیست نگه میدارند. عامل‌های مبتنی بر هدف طوری عمل می‌کنند که آنها را به اهدافشان می‌رساند؛ و عامل‌های مبتنی بر مطلوبیت سعی می‌کنند که "کامیابی" خود را به بیشترین حد برسانند.
- همه عامل‌ها از طریق یادگیری می‌تواند کارایی‌شان را بهبود بخشند.

۲-۶- تمرین‌ها

۲-۱ اصطلاحات زیر را تعریف کنید: عامل، تابع عامل، برنامه‌ی عامل، عقلانیت، خودمختاری، عامل بازتابی ساده، عامل بازتابی مبتنی بر مدل، عامل مبتنی بر هدف، عامل مبتنی بر سود و عامل یادگیرنده.

۲-۲ تفاوت بین معیار کارایی و تابع مطلوبیت چیست؟

۲-۳ این تمرین برای تمایز بین برنامه‌ی عامل و تابع عامل داده شده‌است.

آ) آیا یک تابع عامل توسط چند برنامه عامل قابل پیاده‌سازی است؟ چگونه؟ با مثال توضیح دهید.

ب) آیا توابع عاملی وجود دارند که نتوانند با هیچ برنامه‌ای پیاده‌سازی شوند؟

ج) با داشتن یک معماری ثابت آیا هر برنامه‌ی عامل تنها یک تابع عامل را پیاده‌سازی میکند؟

د) با داشتن یک معماری که شامل n بیت حافظه است. چند برنامه عامل مختلف میتواند موجود باشد؟

۲-۴ در این تمرین می‌خواهیم عقلانیت را برای عامل جاروبرقی بررسی کنیم:

آ) نشان دهید که تابع عامل جاروبرقی شکل ۲-۳ تحت فرضیات گفته‌شده عقلایی است.

ب) تابع عامل عقلایی‌ای برای تغییر معیار کارایی برای کم کردن یک امتیاز به ازای هر حرکت بنویسید. آیا برنامه‌ی عامل متناظر با آن نیاز به حالت درونی دارد؟

ج) طراحی‌های ممکن برای عامل جاروبرقی را برای حالاتی که خانه‌ی تمیز ممکن است کثیف شود و نیز جغرافیای محیط ناشناخته است بنویسید. آیا در این موارد منطقی است که عامل از تجربیاتش یاد بگیرد؟ چه چیزهایی را باید یاد بگیرد؟

۲-۵ برای هریک از عامل‌های زیر توصیف PEAS را برای محیط وظیفه‌ای بنویسید:

آ) ربات فوتبالیست

ب) عامل خرید کتاب اینترنتی

ج) مریخ‌پیمای خودمختار

د) دستیار اثبات مسئله برای ریاضیدان‌ها

۲-۶ برای هریک از عامل‌های تمرین قبل، مشخصات محیط را توصیف کرده و طراحی مناسبی برای آن انتخاب کنید.

تمرینات زیر همگی شامل پیاده‌سازی محیط و عامل برای جهان جاروبری هستند.

۲-۷ شبیه‌ساز معیار کارایی محیط برای عامل جاروبرقی شکل ۲-۲ را پیاده‌سازی کنید. پیاده‌سازی شما باید ماژولار بوده و ویژگی‌های محیط(مثل اندازه،شکل،توزیع آلودگی و...)، حسگرها،محرک‌ها به آسانی قابل تغییر باشند.

۲-۸ عامل بازتابی ساده را برای جهان جاروبرقی تمرین قبل پیاده‌سازی کنید. شبیه‌ساز و این عامل را با تمام حالات اولیه‌ی ممکن آلودگی و مکان عامل اجرا کنید. امتیاز کارایی را برای هر حالت و امتیاز متوسط را ثبت کنید.

۲-۹ نسخه تغییریافته‌ی تمرین ۲-۷ را در نظر بگیرید که به ازای هر حرکت یک امتیاز کم شود.

آ) آیا عامل بازتابی ساده برای این محیط کاملاً عقلایی است؟ شرح دهید.

ب) عامل بازتابی با حالت درونی چه‌طور؟این عامل را طراحی کنید.

ج) پاسخ آ و ب چگونه تغییر میکنند اگر عامل حسگری داشته باشد که آلودگی در همه‌ی خانه‌ها را مشخص کند؟

۲-۱۰ نسخه‌ی تغییر یافته تمرین۲-۷ را در نظر بگیرید که جغرافیای محیط مثل موانع و مرزها، و نیز حالت اولیه آلودگی ناشناخته هستند و عامل میتواند به بالا و پایین نیز حرکت کند.

آ) آیا عامل بازتابی ساده برای این محیط کاملاً عقلایی است؟ شرح دهید.

ب) آیا عامل بازتابی با رفتار تصادفی از عامل بازتابی ساده بهتر عمل می‌کند؟ این عامل را طراحی کرده و کارایی‌اش را برای محیطهای مختلف اندازه بگیرید.

ج) آیا میتوانید محیطی طراحی کنید که عامل با رفتار تصادفی عملکرد خیلی ضعیفی داشته‌باشد؟ نتایج را نشان دهید.

د) آیا یک عامل با حالت درونی از عامل ساده بهتر عمل میکند؟ این عامل را طراحی کرده و کارایی‌اش را برای محیطهای مختلف اندازه بگیرید. آیا میتوانید عامل عقلایی‌ای از این نوع را طراحی کنید؟

۲-۱۱ تمرین ۲-۱۰ را با عاملی که دارای حسگر "تصادم" برای اطلاع از برخورد به موانع و مرزها دارد تکرار کنید. فرض کنید حسگر "تصادم" از کار بیفتد. عامل باید چگونه رفتار کند؟

۲-۱۲ محیطهای جاروبرقی تمرینات قبل همگی قطعی بودند. برنامه‌های عامل ممکن برای هر کدام از نسخه‌های تصادفی این محیطها را مشخص کنید.

آ) قانون مورفی:در ۲۵٪ مواقع عمل مکش آلودگی را پاک نمیکند و آلودگی ایجاد میکند (اگر خانه تمیز باشد.) اگر حسگر آلودگی در ۱۰٪مواقع اشتباه باشد برنامه‌ی عامل چگونه تغییر میکند؟

ب) بچه‌های کوچک: در هر زمان،هر خانه‌ی تمیز ۱۰٪ امکان کثیف شدن دارد. آیا میتوانید عامل عقلایی‌ای برای این مورد طراحی کنید؟

تست‌های طبقه‌بندی‌شده فصل دوم

۱- یک عامل مبتنی بر هدف *(Goal-Based)* در مسئله‌ای با اهداف نسبتا متناقض روبروست. برای یافتن بهترین عمل چه تغییری در این عامل لازم است؟ (کامپیوتر ۸۴)

۱) احتیاج به تغییری نیست.

۲) به معماری بازتابی (Reflex) مجهز باشد.

۳) خود را به تابع مطلوبیت (Utility-function) مجهز کند.

۴) خود را به استدلال مبتنی بر منطق (Logic-Based) مجهز کند.

۲- کدامیک از عبارات زیر صحیح است؟ (کامپیوتر ۸۴)

۱) عاملی که تنها بخشی از محیط را درک می‌کند نمی‌تواند عامل عقلایی یا معقول باشد.

۲) هر عاملی که از رویه‌های استنتاج Sound استفاده کند می‌تواند در تست تورینگ موفق شود.

۳) یک عامل معقول (Rational) همیشه بهتر از عامل‌های غیرمعقول عمل می‌کند چون نتیجه واقعی اعمالش را می‌داند.

۴) عاملی که به زبان طبیعی ارتباط برقرار می‌کند معمولا در یک محیط پاره قابل مشاهده عمل می‌کند.

۳- کدامیک از عبارات زیر صحیح است؟ (کامپیوتر ۸۵)

۱) همه محیط‌های نیمه قابل مشاهده غیر قطعی هستند.

۲) عاملی که به زبان طبیعی محاوره می‌کند در یک محیط نیمه قابل مشاهده عمل می‌کند.

۳) هر عاملی که فقط بخشی از محیط را حس (دریافت) می‌کند نمی‌تواند عقلایی(Rational) باشد.

۴) عاملی که در محیط کاملا قابل مشاهده عمل می‌کند نیازی به حالت درونی ندارد.(Internal state)

۴- کدامیک از جملات زیر صحیح است؟ (کامپیوتر ۸۶)

۱) ممکن است Agent Functionای وجود داشته باشد که نتوان آن را با هیچ Agent Programای پیاده‌سازی نمود.

۲) یک عامل مبتنی بر دانش (Knowledge-based) را نمی‌توان با کمک معماری انعکاسی ساده ساخت.

۳) عامل مبتنی بر مدل (Model based) برای محیط‌های با حالات و اعمال پیوسته مناسب نیست.

۴) در محیط‌های کاملا قابل مشاهده دلیلی برای داشتن حالات داخلی (internal state) نیست.

۵- عاملی که تخته نرد بازی می‌کند در چه محیطی قرار دارد؟ (کامپیوتر ۸۷)

۱) ایستا ـ قطعی ـ مشاهده‌پذیر ـ گسسته ـ ترتیبی

۲) پویا ـ قطعی ـ مشاهده‌پذیر ـ پیوسته ـ واقعه‌ای

۳) ایستا ـ غیرقطعی ـ مشاهده‌پذیر ـ گسسته ـ ترتیبی

۴) ایستا ـ قطعی ـ مشاهده‌پذیر ـ گسسته ـ واقعه‌ای

۶- به کدامیک از دلایل زیر استفاده از مدل در یک عامل می‌تواند مفید باشد؟ (فناوری اطلاعات ۸۷)

۱) گسستگی محیط

۲) استفاده از روش‌های جستجو

۳) پیوستگی محیط

۴) مشاهده ناپذیر بودن محیط

پاسخنامه تشریحی فصل دوم

۱) گزینه ۳ درست است.

تابع مطلوبیت، یک حالت را روی یک عدد حقیقی نگاشت می‌کند، که این عدد میزان مطلوبیت و رضایت عامل از آن هدف را نشان می‌دهد. استفاده از تابع مطلوبیت به ما در تصمیم‌گیری در موارد زیر کمک می‌کند:

نخست، وقتی که اهداف مغایر وجود داشته باشد و فقط بعضی از آنها قابل دستیابی باشند (به عنوان مثال سرعت و امنیت).

دوم، وقتی که اهداف متعددی وجود داشته باشد که عامل می‌خواهد به آن‌ها برسد و هیچ کدام از آنها به طور قطع قابل دسترسی نباشند، در این حالت تابع مطلوبیت راهی را فراهم می‌کند که در آن احتمال موفقیت در مقابله با اهمیت اهداف مورد ارزیابی قرار می‌گیرد.

۲) گزینه ۴ درست است.

خاصیت پردازش زبان طبیعی، ابهام بالا در اطلاعات است. اگر عاملی با پردازش زبان طبیعی ارتباط ایجاد کند (یعنی سنسورها یا action هایش توسط زبان طبیعی مدلسازی شود)، به معنای آن است که دارای دریافت‌های مبهم یا action های مبهم است که در هر حالت به عنوان عدم اشراف به کل محیط است و محیط پاره‌مشاهده‌پذیر خواهد بود.

گزینه ۱ نادرست است.تعریف عامل عقلایی: عامل عقلایی، عاملی است که به ازای دریافت دنباله‌ای از ادراکات، کنش‌هایی را بروز دهد که کارایی آن ماکزیمم باشد.

بنابراین از یک عامل عقلایی انتظار می‌رود که بهترین نتیجه را بدست آورد و یا در صورت وجود عدم قطعیت، بهترین نتیجه ممکن را بگیرد.

گزینه ۲ نادرست است. تعریف استنتاج Sound: اگر تمامی نتایج به صورت rational از حقایق و قوانین استنتاج شوند آنگاه آن استنتاج، Sound خواهد بود.

تعریف تست تورینگ: تست تورینگ که توسط آلن تورینگ ارائه شد (۱۹۵۰)، طراحی شده بود تا یک تعریف عملی قابل قبول از هوش ارائه دهد. تورینگ به جای ارائه لیستی طولانی و گاهاً بحث برانگیز از ویژگیهایی لازم برای هوشمندی، تستی را بر پایه‌ی قابلیت تمییز از موجودی که هوشمندی‌اش غیرقابل انکار است (انسان) پیشنهاد کرد. کامپیوتر در صورتی این تست را پشت سر می گذارد که فرد سوال‌کننده پس از ارائه سؤالاتی به صورت کتبی، نتواند تشخیص دهد که پاسخ-های داده شده به این سؤال‌ها از طرف یک انسان بوده است یا یک کامپیوتر.

گزینه ۳ نادرست است. عامل عقلایی همواره بهتر از عامل غیر معقول کار می‌کند اما همواره از نتیجه اعمالش با خبر نیست.

۳) گزینه ۲ درست است.

نکته: یک محیط ممکن است به دلایل زیر محیطی پاره مشاهده‌پذیر باشد:

- سنسورهای نادقیق

- تاثیر نویز در سنسورها

- درک نکردن بخشی از اطلاعات محیط توسط سنسور

برای مثال در عامل، تاکسی اتوماتیک، عامل نمی‌تواند درک درستی از افکار راننده‌های دیگر داشته باشد. بنابراین محیطی پاره مشاهده‌پذیر دارد.

خاصیت پردازش زبان طبیعی، ابهام بالا در اطلاعات است. اگر عاملی با پردازش زبان طبیعی ارتباط ایجاد کند (یعنی سنسورها یا action هایش توسط زبان طبیعی مدلسازی شود) ، به معنای آن است که دارای دریافت های مبهم یا action های مبهم است که در هر حالت به عنوان عدم اشراف به کل محیط است و محیط پاره‌مشاهده‌پذیر خواهد بود.
گزینه ۱ ناردست است. تعریف محیط نیمه قابل مشاهده: محیط های نیمه قابل مشاهده در مقابل محیط‌های کاملا مشاهده‌پذیر مطرح می‌شوند. اگر سنسورهای عامل، اطلاعات کاملی از وضعیت فعلی محیط در هر زمان دلخواه را به عامل بدهند آنگاه به آن محیط کاملا مشاهده‌پذیر گوییم.

نکته: تعریف محیط غیرقطعی: محیط غیرقطعی در مقابل محیط قطعی مطرح می‌شود. محیط قطعی، محیطی است که به صورت یکتا با استفاده از ۲ فاکتور حالت فعلی و کنش بروز داده شده معین شود.

نکته: اگر محیطی نیمه مشاهده‌پذیر باشد، در بعضی از موارد می‌توان آن را غیر قطعی دانست. این موضوع درباره محیط‌های پیچیده بیش‌تر صدق می‌کند.

گزینه ۳ نادرست است. در صورتی که بخشی از محیط قابل مشاهده باشد، از عامل عقلایی انتظار می‌رود بهترین کنش ممکن را بروز دهد.
گزینه ۴ نادرست است.

نکته: مزیت محیط های کاملا مشاهده پذیر این است، که جهت نگهداشت دریافت اطلاعات نیاز به نگهداری حالت‌های درونی ندارد ولی با توجه به آنکه عامل هوشمند به یک sequence از دریافت ها نیاز دارد ونه فقط یک دریافت ، بنابراین بایستی sequence های قبلی بصورت حافظه یا همان حالتهای درونی (internal state) تعریف گردد.

۴) گزینه ۱ درست است.

چون ممکن است وظایفی برای یک عامل تعریف گردد که قابلیت برنامه ریزی مستقیم وجود نداشته باشد.
گزینه ۲ نادرست است چون معماری انعکاسی قابلیت تعریف پایگاه قواعد (دانش) را دارد و مشکلی ندارد.
گزینه ۳ نادرست است چون عامل مبتنی بر مدل که از حافظه استفاده میکند میتواند حالات و اعمال مختلف را نگهداری کند و عملا این دو موضوع ربطی به هم ندارند.
گزینه ۴ نادرست است. نکته: مزیت محیط های کاملا مشاهده پذیر این است، که جهت نگهداشت دریافت اطلاعات نیاز به نگهداری حالت‌های درونی ندارد ولی با توجه به آنکه عامل هوشمند به یک sequence از دریافت ها نیاز دارد ونه فقط یک دریافت ، بنابراین بایستی sequence های قبلی بصورت حافظه یا همان حالتهای درونی (internal state) تعریف گردد.

۵) گزینه ۳ درست است.

محیط بازی تخته نرد: ایستا است: زیرا زمانی که عامل در حال فکر کردن است تغییری در محیط بازی ایجاد نمی‌شود.
غیرقطعی است: زیرا در آن تاس انداخته می‌شود.

نکته: زمانی به یک محیط قطعی می‌گوییم که حالت بعدی محیط با دو ویژگی زیر به صورت یکتا مشخص شود:

حالت فعلی محیط

کنش بروز داده شده توسط عامل

بنابراین از آنجا که در بازی تخته نرد کنش انجام شدن پرتاب کردن تاس است بنابراین حالت بعدی به صورت یکتا مشخص نمی‌شود. زیرا از قبل نمی‌دانیم عدد تاس چه خواهد بود.

مشاهده‌پذیر است: عامل در بازی تخته نرد در هر لحظه تمامی صفحه را می‌بیند. بنابراین محیط آن مشاهده‌پذیر است.

گسسته است: زیرا تعداد حالت‌های متفاوت در آن متناهی است و همچنین مجموعه‌ای از ادراک‌ها و کنش‌های گسسته دارد.

ترتیبی است: زیرا زمان‌های بازی به قسمت‌های جدا از هم قابل تقسیم نبوده و کنش‌های عامل در طول بازی به یکدیگر وابسته‌اند.

۶) گزینه ۴ درست است.

یکی از کارآمدترین راه‌ها برای حل مشکل مشاهده‌ناپذیر بودن محیط استفاده از عامل مبتنی بر مدل است. در این روش، عامل یکسری حالت درونی که وابسته به ادراکات و مشاهدات قبلی او است، را در خود نگهداری می‌کند و با استفاده از آن قسمت‌های مشاهده‌ناپذیر حالت فعلی را حدس می‌زند.

برای به‌روزرسانی این حالت درونی، عامل نیاز به دو مجموعه اطلاعات دارد:

اول اینکه جهان پیرامون عامل، صرف نظر از وجود عامل و مستقل از آن، چگونه تکامل یافته و چگونه تغییر می‌کند.

دوم اینکه کنش‌های عامل چگونه برروی جهان اطراف او اثر می‌گذارد.

این اطلاعات که درباره چگونگی تکامل و قوانین جهان است را مدلی از جهان گویند. عاملی که از چنین مدلی استفاده می‌کند را نیز عامل مبتنی بر مدل گویند.

تست‌های تألیفی

۱- بازی نقطه – خط دارای کدامیک از خواص زیر است:

۱) پاره مشاهده پذیر- قطعی-Episodic- ایستا- پیوسته- تک عامله

۲) کاملاً مشاهده پذیر- قطعی-Episodic- ایستا- گسسته- چند عامله

۳) کاملاً مشاهده پذیر- قطعی-sequential- ایستا- گسسته- چند عامله

۴) کاملاً مشاهده پذیر- غیرقطعی-Episodic- ایستا- گسسته- چند عامله

۲- یکی از مشکلات اصلی عامل های واکنشی ساده (Simple Reflex Agent)، قرار گرفتن عامل در حلقه بی-نهایت (infinite loop) است. این مشکل را با کدامیک از روشهای زیر می‌توان حل کرد:

۱) استفاده از قوانینی که حالتهای تکراری را چک کند

۲) استفاده از حافظه و تبدیل عامل به model based

۳) استفاده از روشهای تصادفی (randomize)

۴) گزینه های الف و ب

۳- کدامیک از عوامل زیر سبب نمی‌شود که عامل با مشکل اطلاعات جزئی جهت تصمیم‌گیری مواجه شود:

۱) عامل سنسور ندارد.

۲) عامل نداند که خروجی action هایش قبل از اجرای action ، چیست.

۳) عامل نداند که چه actionهایی را می‌تواند اجرا کند.

۴) عامل نداند که تعداد کل حالتهای موجود در فضای مسئله چقدر است.

۴- در کدام گزینه، همه ی مسائل هوش مصنوعی دارای محیطهایی با خواص گسسته، کاملا مشاهده پذیر، ایستا، قطعی هستند:

۱) جدول کلمات متقاطع، شطرنج، بازی مار-پله

۲) جدول کلمات متقاطع، سیستم تشخیص طبی، بازی نقطه –خط

۳) بازی تخته نرد

۴) هیچکدام

۵- عامل هوشمند به حافظه نیاز دارد چون:

۱) می‌خواهد مسائلی که در محیط پاره مشاهده پذیر است را حل کند.

۲) چون می‌خواهد بداند که قبلا چه action هایی انجام داده است.

۳) هر سیستمی که بخواهد مسئله ای حل کند، علاوه بر پردازش به حافظه نیاز دارد.

۴) هیچکدام

۶- عامل جاروبرقی را در نظر بگیرید. فرض کنید معیار کارایی برای این عامل، مقدار گرد و خاک جمع‌آوری شده توسط آن در یک بازه زمانی است. کدامیک از جملات زیر توصیف‌کننده رفتار عقلایی از این عامل است؟

۱) تمیز نگه داشتن محیط مورد نظر در طول بازه زمانی.

۲) جمع‌آوری و خالی کردن گردوخاک به صورت متناوب در طول بازه زمانی.

۳) جستجوی محیط برای کشف خانه‌های ناشناخته.

۴) هیچکدام از رفتارهای توصیف شده عقلایی نیستند.

پاسخنامه تشریحی تست‌های تألیفی

۱) گزینه ۲ صحیح است.

۲) گزینه ۴ درست است.

هم حالت الف و هم ب درست است. نمی‌توان بدون حافظه قوانینی طرح کرد که حالتهای تکراری را کنترل کرد.

۳) تمام گزینه های ۱و ۲و ۳ سبب می‌شود که عامل با مشکل Partial information مواجه شود.

۴) گزینه ۴ درست است.

می‌شود و غیرقطعی است. سیستم تشخیص طبی سیستم پاره مشاهده پذیر است.بازی تخته نرد ، غیر قطعی است.

۵) مورد ۱، برای حل مسائلی که پاره مشاهده پذیر هستند، استفاده از حافظه پیشنهاد می‌شود که در آن جاهایی که قبلا دیده شده است، ذخیره می گردد.

۶) گزینه ۲ درست است.

شاید پاسخ این سوال، در نگاه اول کمی تعجب‌برانگیز به نظر برسد. اما برای توصیف رفتار عقلایی باید به‌دنبال بیشینه کردن معیار کارایی باشیم. حتی اگر معیار توصیف شده، سنجش درستی از کارایی عامل نباشد. با توجه به معیار کارایی تعریف شده در این سوال، گزینه ۲ رفتاری عقلایی است (زیرا معیار کارایی را بیشینه می‌کند).

فصل سوم

جستجوی ناآگاهانه

ساده‌ترین عاملهای بحث شده در فصل ۲ عامل‌های بازتابی ساده بودند که کنش آنها بر اساس یک تناظر مستقیم از حالت به کنش انتخاب می‌شد. این عامل‌ها در محیط‌هایی که این تناظر برایشان بزرگتر از آن است که قابل ذخیره‌سازی یا قابل یادگیری باشد (زیرا زمان زیادی طول می‌کشد) بخوبی عمل نمی‌کنند. در مقابل عاملهای مبتنی بر هدف با در نظر گرفتن کنشهای آینده و نیز مقبولیت نتیجه‌ی کنش‌ها بخوبی عمل می‌کنند.

در این فصل، یک نوع از عامل‌های مبتنی بر هدف به نام عامل‌های حل مسئله را توصیف می‌کنیم. عامل‌های حل کننده مسئله با پیدا کردن یک رشته از کنشهایی که به حالت مطلوب می‌رسند، درمورد کنش‌های خود تصمیم-گیری می‌کنند. با تعریف دقیق عناصر تشکیل‌دهنده "مسئله " و "پاسخ" آن جستجوی شروع کرده و مثالهایی برای تشریح این تعاریف ارائه می‌کنیم. سپس چند الگوریتم جستجوی عام‌منظوره، برای حل این مسائل توصیف و مزایای آنها را با یکدیگر مقایسه خواهیم نمود. از آنجا که به جز تعریف مسئله، دانش دیگری در رابطه با مسئله در اختیار این الگوریتم‌ها نیست به آن‌ها الگوریتم‌های ناآگاهانه[۱] می‌گوییم. فصل ۴ با الگوریتم‌های جستجوی آگاهانه سر و کار دارد که دانشی در مورد نقاط احتمالی جواب مسئله در اختیار دارد. در این فصل از یک سری مفاهیم مربوط به تحلیل الگوریتم‌ها استفاده می‌شود.

۳-۱- عامل‌های حل‌کننده مسئله

فرض بر این است که عامل‌های هوشمند به گونه‌ای عمل می‌کنند که معیار کارایی را بیشینه کنند. همان‌طور که در فصل ۲ گفته شد، این کار (بیشینه کردن معیار کارایی) از طریق در نظر گرفتن هدف و تلاش برای رسیدن به آن بسیار آسان‌تر خواهد بود. اجازه دهید در ابتدا ببینیم که چرا و چگونه یک عامل می‌تواند این کار را انجام دهد.

فرض کنید که عامل در شهر آراد[۲] رومانی است و از مسافرت ایام تعطیلش لذت می‌برد. معیار کارایی عامل فاکتورهای زیادی در بر دارد: بخواهد بیشتر از این برنزه شود، زبان کشور رومانی را بهتر بیاموزد، از مناظر بیشتری لذت ببرد و غیره. مسئله تصمیم‌گیری، مسئله پیچیده‌ای است که نیازمند مصالحه[۳]ها و خواندن دقیق کتاب راهنمای سفر می‌باشد. حال فرض کنید عامل یک بلیط برای پرواز به بخارست[۴] دارد و بلیط قابل پس‌دادن نیست. در این حالت عاقلانه است که عامل هدف رسیدن به بخارست را مدنظر بگیرد. کنشهایی که باعث می شوند که عامل به موقع به بخارست نرسد بدون هیچ ملاحظه‌ای باید نادیده گرفته‌شوند. در واقع با این کار مسئله

۱-Uninformed
۲- Arad
۳-Trade-off
۴- Bucharest

تصمیم‌گیری عامل به حد زیادی ساده می‌شود. اولین قدم در حل مسئله فرموله‌سازی هدف براساس موقعیت کنونی و معیار کارایی عامل است. بعد از فرموله‌سازی هدف، عامل می‌تواند در مورد انتخاب فاکتورهای دیگری که در مطلوبیت رسیدن به هدف تاثیر گذارند، تصمیم‌گیری کند.

ما هدف را به صورت مجموعه‌ای از حالت‌های موجود در محیط در نظرمی‌گیریم. همان حالت‌هایی که هدف در آنها صدق می‌کند. وظیفه عامل یافتن دنباله‌ای از کنشهاست که او را به یک حالت هدف برساند. اما قبل از آن، عامل باید در مورد کنش‌ها و حالت‌های خود تصمیم‌گیری کند. در این مثال اگر عامل کنش‌هایش را در سطحی مثل "پای چپت را یک اینچ به جلو ببر" یا "فرمان را یک درجه به سمت چپ بچرخان "انتخاب کند، هیچ‌وقت راه خروج از پارگینگ را پیدا نمی‌کند چه برسد به راه رسیدن به بخارست! چون انتخاب کنش‌ها در این سطح از جزئیات، عدم قطعیت زیادی دارد. همچنین توجه به این سطح از جزئیات باعث طولانی شدن دنباله راه‌حل خواهد شد. *فرموله‌سازی مسئله* یعنی روند تصمیم‌گیری اینکه چه حالت‌ها و کنش‌هایی با توجه به هدف داده شده باید در نظر گرفته شوند. ما این روند را با جزئیات بیشتر مورد بررسی قرار خواهیم‌داد. در حال حاضر فرض کنید که عامل کنش‌هایش را در سطح رسیدن از یک شهر به شهر دیگر در نظر می‌گیرد. در نتیجه حالت‌هایی که در نظر می گیرد به معنی حضور در یک شهر خاص است[۱].

عامل ما رسیدن به بخارست را به عنوان هدف خود قبول‌کرده است. حال، بررسی می‌کند که چه شهری را بعد از آراد انتخاب کند. سه جاده از آراد خارج می‌شود، یکی به سمت سیبیو[۲]، یکی به سمت تی‌می‌سوآرا[۳] و یکی به سمت زرینـد[۴]. هیچ‌کدام از این راه‌ها مستقیماً به هدف نمی‌رسند و از آنجایی که عامل با نقشه رومانی آشنا نیست، نمی‌داند که کدام راه را انتخاب کند. در واقع عامل نمی‌داند که کدام کنش بهتر است. چون اطلاعات کافی در مورد حالت و نتیجه حاصل از کنش ندارد. اگر عامل هیچ دانش اضافه‌ای نداشته باشد، گیر می‌افتد. در این حالت بهترین کاری که می‌تواند انجام دهد این است که یکی از راه‌ها را به صورت تصادفی انتخاب کند.

اما حالا فرض کنید که عامل یک نقشه از رومانی در اختیار دارد، چه روی کاغذ و چه در حافظه‌اش. هدف نقشه این است که اطلاعات کافی در مورد انتخاب حالت بعدی و کنشی که باید انجام دهد در اختیار عامل قراردهد. عامل می‌تواند از این اطلاعات استفاده کرده و مراحل بعدی سفر خود را در انتخاب هر یک از سه راه ممکن حدس بزند و به این ترتیب تشخیص دهد که کدام سفر او را به بخارست می‌رساند. بعد از پیدا کردن یک مسیر از آراد به بخارست از روی نقشه، عامل می‌تواند با انتخاب کنشهای مناسب مطابق با قدم‌هایی که در طول مسیر باید طی شود، به هدف برسد. در حالت کلی، عاملی که انتخاب‌های متعددی پیش رو دارد و نمی‌داند ارزش و هزینه هر کدام چه‌قدر است، می‌تواند همه رشته‌های ممکن از کنش‌ها را که به یک حالت با ارزش مشخص می رسند امتحان و بهترین آنها را انتخاب کند. به فرایند پیدا کردن بهترین رشته، جستجو می‌گویند. یک الگوریتم جستجو یک مسئله را به عنوان ورودی گرفته و راه‌حل آن را به صورت رشته‌ای از کنشها برمی‌گرداند. وقتی یک راه‌حل پیدا شد، کنشهای پیشنهاد شده توسط این راه‌حل قابل انجام هستند. به این کار مرحله اجرا می‌گویند.

---

[۱]- توجه کنید که هر کدام از این حالات در واقع متناظر با یک مجموعه بزرگ از حالات جهان است زیرا یک حالت از جهان واقعی همه‌ی جنبه‌های واقعیت را مشخص میکند. توجه به تمایز بین حالات حل مسئله و حالات جهان بسیار مهم است.

[۲]-Sibiu

[۳]- Timisoara

[۴]- Zerind

بنابراین، برای طراحی هر عامل یک روش ساده به صورت "فرموله‌سازی، جستجو، اجرا"وجود دارد که در شکل ۳-۱ نشان داده شده است. بعد از فرموله‌سازی هدف و مسئله‌ای که باید حل شود، عامل یک رویه جستجو را برای حل مسئله صدا می‌زند. سپس از راه‌حل به دست‌آمده برای هدایت کنشها استفاده می‌کند. به این ترتیب که کنش بعدی را بر طبق چیزی که راه‌حل پیشنهاد می‌کند انتخاب می‌کند (که معمولاً اولین کنش در دنباله است) و سپس این قدم را از رشته حذف می‌کند. وقتی‌که راه‌حل اجراشد، عامل یک هدف جدید را فرموله خواهد کرد.

ابتدا فرآیند فرموله‌سازی مسئله را توصیف و سپس نسخه‌های گوناگون تابع Search را معرفی خواهیم نمود.

قبل از وارد شدن به جزئیات اجازه بدهید مکث کوتاهی بکنیم و ببینیم عاملهای حل مسئله در کجای مبحث عامل‌ها و محیط‌های فصل ۲ قرار میگیرند. طراحی عامل شکل ۳-۱ فرض می‌کند، محیط ایستا است زیرا فرموله‌سازی و حل مسئله بدون هیچگونه توجهی به تغییرات محیط انجام شده‌اند. همچنین طراحی عامل فرض کرده است که حالت اولیه مشخص است زیرا میدانیم محیط قابل مشاهده ساده‌ترین محیط ممکن است. در نظر گرفتن "گزینه‌های ممکن برای کنش‌ها" به‌معنی گسسته بودن محیط است. سرانجام و از همه مهمتر طراحی عامل فرض میکند که محیط قطعی است. زیرا جواب مسائل تنها دنباله‌ای از کنشها هستند، بنابراین نمیتوانند اتفاقات غیرمنتظره را کنترل کنند. علاوه بر این جوابها بدون در نظر گرفتن ادراک‌ها اجرا میشوند. عاملی که با چشم بسته نقشه‌هایش را اجرا میکند باید از اتفاقات اطرافش کاملاً مطمئن باشد. محققان در حوزه کنترل به این موضوع، سیستم حلقه باز می‌نامند زیرا بی توجهی به ادراک‌ها، حلقه بین عامل و محیط را می‌شکند. معنی همه این فرضیات این است که ما با ساده‌ترین نوع محیط سروکار داریم و دلیل اینکه این فصل در ابتدای کتاب آمده‌است نیز همین است

```
function SIMPLE–PROBLEM-SOLVING-Agent(percept) returns an action
    inputs:percept, a percept
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ←UPDATE-STATE (state, percept)
    if seq is empty then do
        goal ←FORMULATE-GOAL (state)
        problem ←FORMULATE- PROBLEM (state, goal)
        seq←SEARCH (problem)
    action ←FIRST (seq)
    seq←REST (seq)
    return action
```

شکل ۳-۱ عامل حل مسئله ساده. ابتدا هدف و مسئله را فرموله می‌کند. سپس رشته‌ای از کنش‌ها که مسئله را حل می‌کند، پیدا کرده و کنش‌ها را یکی یکی اجرا می‌کند. وقتی کل کنش‌ها اجرا شد، هدف جدیدی را فرموله کرده و دوباره فرایند را آغاز می‌کند. دقت کنید که در حال اجرای رشته، ادراک‌ها را نادیده گرفته و فرض می‌کند که راه حل یافته شده حتماً موفق است.

۳-۱-۱- مسائل و راه‌حل‌های خوش- تعریف

یک مسئله به صورت رسمی از ۴ جزء تشکیل شده‌است:

- حالت اولیه‌ای که عامل در آن شروع می‌کند. برای مثال، حالت اولیه برای عامل ما در رومانی به صورت In(Arad) توصیف می‌شود.

- توصیف کنشهای مجاز در دسترس عامل. شایع‌ترین نوع فرموله‌سازی از یک تابع با نام successor function (تابع پسین) استفاده می‌کند. فرض کنید x یک حالت خاص است، تابع successor-FN(x) مجموعه ای از زوجهای مرتب (کنش، حالت بعدی) را برمی‌گرداند که کنش‌ها در آن، یکی از کنش‌های مجاز حالت x و حالت‌های بعدی نشان‌دهنده حالتی است که بعد از اجرای کنش روی حالت x بدست می‌آید. برای مثال، اگر حالت x برابر In(Arad) باشد آنگاه تابع successor پاسخ زیر را برمی‌گرداند:

{(Go (Sibiu), In (Sibiu)), (Go (Timisoara), In (Timisoara)), (Go (Zerind), In (Zerind))}

با این روش، فضای حالت مسئله تعریف می‌شود(مجموعه تمام حالت‌هایی که از حالت اولیه قابل دسترسی هستند). فضای حالت گرافی را تشکیل میدهد که رئوس نشان دهنده حالات و یالها نشان‌دهنده کنشها هستند. (اگر جاده‌ها را در شکل ۳-۲ را به صورت کنش رانندگی دوطرفه در نظر بگیریم آنگاه نقشه رومانی به عنوان گراف فضای حالت قابل تفسیر است). یک مسیر در فضای حالت، تعدادی از حالت‌های متوالی است که با ترتیبی از کنشها به هم متصل شده‌اند.

- تست هدف، مشخص می‌کند که آیا یک حالت داده شده حالت هدف است یا خیر. بعضی وقت‌ها یک مجموعه صریح از حالات هدف ممکن، وجود دارد و تست هدف چک می‌کند که آیا هدف داده شده در این مجموعه قرار دارد یا خیر. هدف عامل در رومانی رسیدن به مجموعه تک عضوی {In (Bucharest)} است. گاهی اوقات هدف با یک خصوصیت انتزاعی توصیف می‌شود (به جای مجموعه‌ای صریح از حالات). برای مثال، در بازی شطرنج، هدف رسیدن به حالت "کیش و مات " است، یعنی حالتی که مهره شاه حریف در حرکت بعدی قابل حذف است (مستقل از اینکه حریف چه حرکتی انجام می‌دهد).

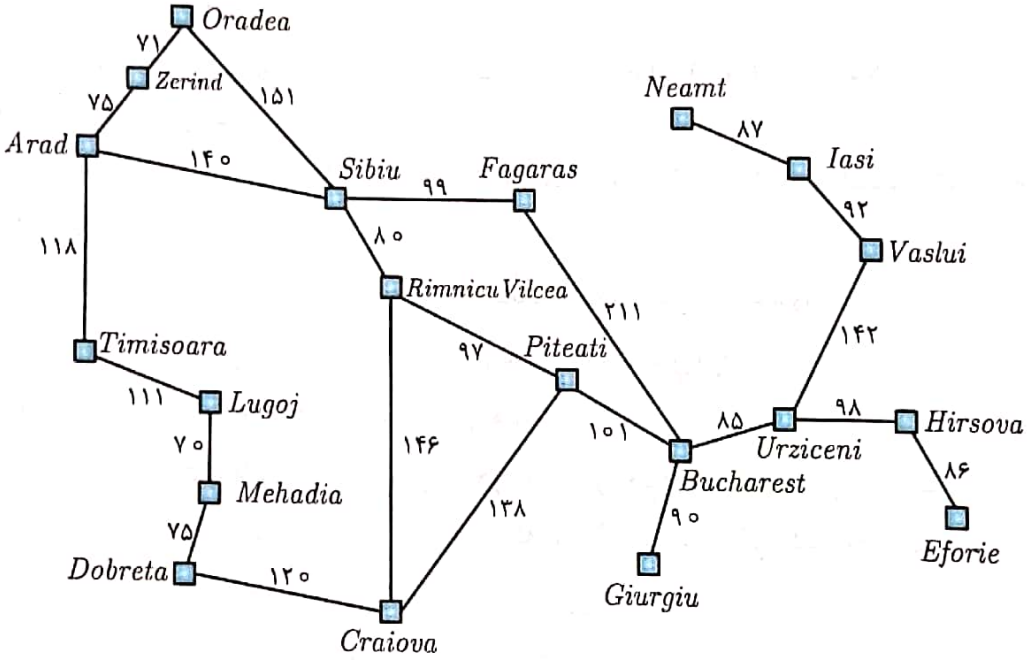
- تابع هزینه مسیر، تابعی است که به هر مسیر هزینه‌ای نسبت می‌دهد. عامل حل کننده مسئله تابع هزینه‌ای را انتخاب می‌کند که منعکس‌کننده میزان کارایی است. برای عاملی که تلاش می‌کند به بخارست برسد زمان حیاتی است. بنابراین هزینه یک مسیر ممکن است طول آن به کیلومتر باشد. در این فصل فرض می‌کنیم که هزینه یک مسیر مجموع هزینه کنشهایی است که در طول مسیر انجام شده‌اند. هزینه انجام یک کنش a برای رفتن از حالت x به حالت y با c(x, a, y) نشان داده میشود. هزینه‌های مراحل مختلف در رومانی در شکل ۳-۲ به عنوان فاصله مسیرها نشان داده شده‌اند. فرض می‌کنیم که هزینه‌ی گام‌ها نامنفی هستند.

در مجموع، حالت اولیه، مجموعه کنش‌ها، حالت هدف و تابع هزینه، مسئله را تعریف می‌کنند و با هم ساختمان داده‌ای را تشکیل می‌دهند که به عنوان ورودی به عامل حل مسئله داده می‌شود.

راه‌حل مسئله، در واقع مسیری از حالت اولیه به حالتی با خصوصیات حالت نهایی یا هدف است. کیفیت راه حل با استفاده از تابع هزینه مسیر اندازه‌گیری می‌شود. پاسخ بهینه دارای کمترین هزینه مسیر در بین همه‌ی پاسخ-های ممکن است.

۳-۱-۲- فرموله‌سازی مسئله

در بخش قبل فرموله‌سازی مسئله برای رسیدن به بخارست با استفاده از حالت اولیه، تابع پسین، حالت هدف و هزینه مسیر پیشنهاد شد .این فرموله‌سازی عاقلانه به نظر می‌رسد. با این حال،جنبه‌های زیادی از جهان را حذف می‌کند. توصیف ساده شده حالت In(Arad) را با یک سفر واقعی (که حالات جهان در آن شامل جزئیات بی-شماری می‌شود) مقایسه کنید: مسافران،برنامه‌ای که رادیو پخش میکند، منظره‌ای که از پنجره دیده میشود، پلیسهایی که در نزدیکی هستند، فاصله تا استراحتگاه بعدی، شرایط جاده، هوا و غیره. همه‌ی اینها از توصیف ما حذف شده‌اند زیرا آنها به مسئله پیدا کردن مسیر به بخارست مربوط نیستند. فرایند حذف جزئیات از نمایش حالت را انتزاع[۱] گویند.



شکل ۳-۲ نقشه‌ی ساده شده‌ی بخشی از رومانی

علاوه بر انتزاع توصیف حالات، باید کنشها را هم انتزاعی کنیم. یک کنش رانندگی ممکن است اثرات زیادی داشته‌باشد. همچنین عوض شدن مکان و سرنشینان خودرو زمان بر است، سوخت مصرف میکند،آلودگی ایجاد می‌کند و عامل را تغییر میدهد. در فرموله‌سازی کنش، ما فقط تغییر مکان را در نظر می‌گیریم. همچنان، کنش های زیادی وجود دارند که همگی را حذف می‌کنیم: روشن کردن رادیو، نگاه کردن به بیرون از پنجره، کم کردن سرعت برای پلیس و غیره.

آیا می‌توانیم تعریف سطح انتزاع مناسب را به طور دقیق‌تری بررسی کنیم؟ به حالات و کنش‌های انتزاعی که متناظر با حالات و کنش‌های جهان (که بسیار پرجزئیات هستند) در نظر گرفته‌ایم فکر کنید. حال یک راه حل برای مسئله انتزاعی در نظر بگیرید: برای مثال، مسیر از آراد به سیبیو به ریمنیسو[۲] ویلسا به پیتستی[۳] به بخارست. این پاسخ انتزاعی متناظر با تعداد زیادی مسیر و با جزئیات فراوان است. برای مثال، می‌توانیم با رادیو

۱-Abstraction
۲-Rimnicu Vilcea
۳- Pitesti

روشن بین سیبیو و ریمنیسو ویلسا حرکت کرده و در ادامه مسیر رادیو را خاموش کنیم. انتزاع معتبر است اگر بتوانیم پاسخ انتزاعی را به پاسخ جهان واقعی بسط دهیم. انتزاع وقتی مفید است که اجرای هر کنش در راه‌حل، آسان‌تر از مسئله واقعی باشد. بنابراین انتخاب انتزاع خوب شامل حذف بیشترین جزئیات ممکن و نیز اطمینان از اجرای آسان کنش‌های انتزاعی است.

۳-۲- مسائل نمونه

رویکرد حل مسئله برروی گستره زیادی از محیط‌های وظیفه‌ای اعمال شده است. حال لیستی از مسائل معروف به این شکل را معرفی خواهیم نمود.

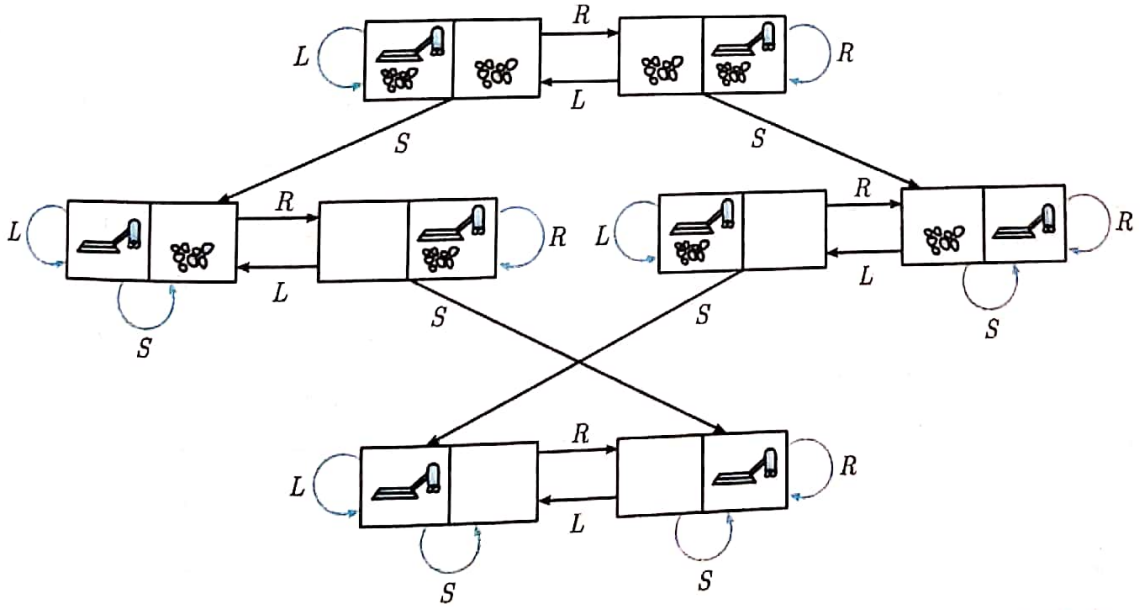
۳-۲-۱- مسائل ساختگی

اولین نمونه‌ای که معرفی می‌کنیم جهان جاروبرقی است که در فصل ۲ معرفی شد(شکل ۲-۲). این جهان به شکل زیر قابل فرموله سازی است:

- حالت‌ها: عامل در یکی از دو خانه است که هر کدام ممکن است کثیف یا تمیز باشند. بنابراین $۸ = ۲^۲ \times ۲$ حالت ممکن وجود دارد.
- حالت اولیه: هر کدام از حالتها میتوانند به عنوان حالت اولیه در نظر گرفته شوند.
- تابع پسین: حالتهای ممکن حاصله از سه کنش (راست، چپ، مکش) را تولید می‌کند.
- تست هدف: چک میکند که آیا همه خانه‌ها تمیز هستند یا نه.
- هزینه مسیر: هر گام هزینه‌اش ۱ است. بنابراین هزینه مسیر تعداد گامها در مسیر می‌باشد.

در مقایسه با جهان واقعی این مسئله سرگرمی، دارای خانه‌های گسسته، کثیفی گسسته، تمیزکننده قابل اعتماد است و وقتی خانه‌ای تمیز شد دیگر آلوده نمی‌شود.

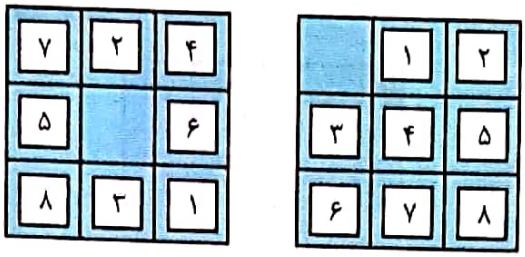
نکته مهمی که باید مورد توجه قرار گیرد این است که حالت هم از روی آلودگی و هم از مکان عامل تعیین میشود. محیط بزرگتری با n خانه $n \times ۲^n$ حالت دارد.



شکل ۳-۳ فضای حالت برای جهان جاروبرقی. یالها نشان دهنده‌ی کنش‌های: چپ=L ، راست=R و مکش=S هستند.

معمای ۸ پازل

نمونه‌ای از معمای ۸- پازل در شکل ۳-۴ نشان داده شده است که شامل یک صفحه ۳×۳ با ۸ مربع شماره‌دار در یک صفحه خالی است. هر مربع که همسایه خانه خالی باشد، می تواند به درون آن خانه برود. هدف رسیدن به ساختاری است که در سمت راست شکل نشان داده شده است.



شکل ۳-۴ یک نمونه از معمای ۸- پازل

حال کار را بر مبنای تعاریف زیر ادامه می دهیم.

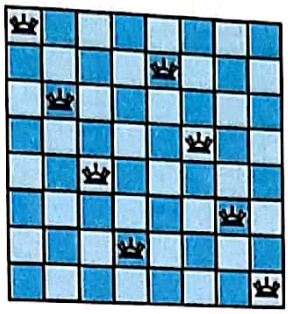
- حالتها: وضعیت محل هریک از ۸ مربع را در یکی از ۹ خانه صفحه مشخص می کند. برای کارائی بیشتر، بهتر است که فضاهای خالی نیز ذکر شود.
- حالت اولیه: هر حالت میتواند به عنوان حالت اولیه در نظر گرفته شود. توجه کنید که هر هدف دقیقاً از نصف حالات اولیه ممکن قابل دسترس است.
- تابع پسین: حالتهای مجاز (حرکت فضای خالی به چپ، راست، بالا و پایین) حاصله از اجرای کنش را تولید می‌کند.
- تست هدف: وضعیت با ساختار شکل ۳-۴ مطابقت می‌کند.
- هزینه مسیر: هر قدم ارزش ۱ دارد، بنابراین هزینه مسیر همان طول مسیر می‌باشد.

چه انتزاعهایی را در این مسئله به کار برده‌ایم؟ کنشها به حالت ابتدایی و پایانی‌شان انتزاعی شده‌اند زیرا خانه-های میانی که خانه خالی در آنها حرکت میکند نادیده گرفته شده‌اند. کنشهایی مثل تکان دادن صفحه، زمانی که خانه‌ها گیر کرده‌اند، یا خارج کردن قطعه‌ها با چاقو و برگرداندن آنها سر جایشان را حذف کرده‌ایم و تنها توصیفی از قوانین بازی بدون درگیر شدن با جزئیات فیزیکی باقی می‌ماند.

معمای ۸ متعلق به خانواده بلاک‌های پازل متحرک می باشد. این کلاس عمومی به عنوان NP-کامل شناخته می شود. بنابراین کسی انتظار ندارد که روش‌هائی پیدا شوند که بهتر از الگوریتم‌های جستجوی تعریف شده در این فصل و فصل بعدی باشند. معمای ۸ ، ۱۸۱۴۴۰ = ۹۱/۲ حالت قابل دسترس دارد که به راحتی حل می-شود. معمای ۱۵ (در صفحه ۴×۴)، حدود ۱.۳ تریلیون حالت دارد و نمونه‌های تصادفی می‌توانند به طور بهینه در چند میلی ثانیه با بهترین الگوریتمهای جستجو حل شوند. معمای ۲۴ (در صفحه ۵×۵) حدود $۱۰^{۲۵}$ حالت دارند و نمونه‌های تصادفی هنوز به طور بهینه با ماشینها و الگوریتمهای کنونی به آسانی قابل حل نیستند.

مسئله ۸ وزیر

هدف از مسئله ۸ وزیر، قرار دادن ۸ وزیر بر روی صفحه شطرنج می‌باشد به صورتی کـه هـیچ وزیـری نتوانـد بـه دیگری حمله کند. (یک وزیردرحالت افقی، عمودی و قطری می تواند حمله کند) شکل۳-۵ راه حلی را برای این مسئله نشان می دهد که با شکست مواجه شده است. وزیرواقع در سمت راست ترین ستون توسط وزیر واقـع در سمت چپ بالا مورد حمله قرار گرفته است.



شکل ۳-۵ حالت نزدیک به راه‌حل مسئله‌ی ۸ وزیر. (راه‌حل به عنوان تمرین واگذار شده‌است) .

گرچه الگوریتم‌های خاص‌منظوره‌ای برای این مسئله و تمام خانواده مسئله n وزیر وجود دارند با این حال این مسئله برای تست الگوریتمهای جستجو بسیار جالب است. دو نوع فرموله‌سازی اصلی وجود دارد. فرموله‌سازی افزایشی که با جایگزینی وزیرها به صورت تک تک کار می‌کند و دیگری فرموله‌سازی حالت کامل که با تمام ۸ وزیر روی صفحه شروع و آنها را حرکت می‌دهد. در هر دو مورد، هزینه مسیر به غیر از مرحله آخر، قابل چشم‌پوشی است. از این رو الگوریتمها فقط براساس هزینه جستجو مقایسه می شوند. بنابراین ما تست هدف و هزینه مسیر را به صورت زیر خواهیم داشت:

- حالات: هر ترتیبی از ۰ تا ۸ وزیر روی صفحه یک حالت است.
- حالت اولیه: هیچ وزیری روی صفحه نباشد.
- تابع پسین: اضافه کردن یک وزیر به یک خانه خالی.
- تست هدف: ۸ وزیر روی صفحه که با هم برخورد ندارند.

در این فرموله‌سازی $۱۰^{۱۴} \times ۳ = ۵۷ \times \ldots \times ۶۳ \times ۶۴$ نتیجه ممکن برای بررسی وجود دارد. فرموله‌سازی بهتری نیز وجود دارد که از قرار دادن وزیر در خانه‌ای که تحت حمله است جلوگیری می‌کند:

- حالات: به ترتیب از n وزیر ($۸ \geq n \geq ۱$) هر یک در یکی از چپ ترین n ستون بدون برخورد با وزیرهای دیگر.
- تابع پسین: یک وزیر را در خالی ترین ستون سمت چپ جایگزین کنید که هیچ برخوردی با بقیه نداشته باشد.

این فرموله‌سازی فضای حالت را از $۱۰^{۱۴} \times ۳$ به ۲۰۵۷ کاهش می‌دهد که مسلما یافتن راه حل در آن بسیار آسان‌تر است. از طرف دیگر برای ۱۰۰ وزیر فرموله‌سازی اولیه حدود $۱۰^{۴۰۰}$ حالت دارد در حالیکه فرموله‌سازی بهبود یافته، حدود $۱۰^{۵۲}$ حالت دارد (تمرین۳-۵). این کاهش بزرگی است ولی فضای حالت بهبود یافته هنوز

برای الگوریتمهای این فصل بسیار بزرگ است. فصل ۴ فرموله‌سازی حالت کامل را توصیف می‌کند و فصل ۵ یک الگوریتم ساده را ارائه میدهد که حتی مسئله یک میلیون وزیر را به راحتی حل میکند.

۳-۲-۲- مسائل دنیای واقعی

مسیریابی

قبلاً دیدیم که چطور مسیریابی تحت عنوان مکانها و ارتباطات ویژه در طول اتصالات بین آنها ، تعریف می‌شود. الگوریتم‌های مسیریابی کاربردهای فراوانی دارند، مانند مسیریابی در شبکه‌های کامپیوتری، برنامه‌ریزی عملیات نظامی و سیستم‌های برنامه‌ریزی مسافرتی هوایی. بیان مشخصات این مسائل معمولاًبسیار پیچیده هستند. نمونه ساده شده مسئله خطوط هوایی را در نظر بگیرید:

- حالات: هر کدام با یک مکان (به عنوان مثال فرودگاه) و زمان حاضر نمایش داده می‌شود.
- حالت اولیه: توسط مسئله تعیین میشود.
- تابع پسین: حالاتی را برمی‌گرداند که از انتخاب هر پرواز برنامه‌ریزی شده بدست می‌آید که زمان پرواز آن بعد از زمان کنونی به اضافه زمان انتقال از این فرودگاه به فرودگاه مقصد است.
- تست هدف: آیا در زمان مورد نظر در مقصد هستیم؟
- هزینه مسیر: بستگی دارد به هزینه مالی،زمان انتظار، زمان پرواز، پروسه امور مهاجرت، کیفیت صندلی، زمان پرواز، نوع فرودگاه، تخفیف برای مسافرانی که زیاد پرواز میکنند.

سیستم‌های تجاری مشاور مسافرت از این نوع فرموله‌سازی همراه با پیچیدگی‌های بیشتری برای مسئله استفاده می‌کنند.

مسئله‌های مسافرتی بسیار نزدیک به مسائل مسیریابی هستند امّا یک تفاوت عمده دارند. مسئله "هر شهر در شکل۳-۲ را حداقل یکبار ملاقات کنید به طوری که شروع و پایان در بخارست باشد" را در نظر بگیرید. مانند مسأله مسیریابی، کنشها متناظر با سفر بین شهرهای مجاور هستند. اما برای این مسئله، فضای حالت باید اطلاعات بیشتری را حفظ کند. علاوه بر مکان عامل، هر حالت باید مجموعه شهرهایی را که عامل ملاقات کرده، نگه دارد. بنابراین حالت اولیه " In Bucharest;visited{Bucharest}" خواهد بود.

یک حالت واسط "In Vaslui;visited{Bucharest,Urziceni,Vaslui}" خواهد بود و تست هدف کنترل خواهد کرد که آیا عامل در بخارست است و تمام ۲۰ شهر را ملاقات کرده است.

TSP

مساله فروشنده دوره گرد (TSP) مسئله مشهوری است که در آن هر شهرحداقل یکبار باید ملاقات شود. هدف یافتن کوتاهترین مسیر است. این مسئله NP- سخت است. اما کوشش‌های بسیاری برای بهبود قابلیت‌های الگوریتم TSP انجام شده است. علاوه بر برنامه‌ریزی سفر برای فروشنده دوره گرد این الگوریتم‌ها برای اعمالی نظیر برنامه‌ریزی حرکات مته خودکار سوراخ کننده برد مدار استفاده می‌شود.

برای الگوریتمهای این فصل بسیار بزرگ است. فصل ۴ فرموله‌سازی حالت کامل را توصیف می‌کند و فصل ۵ یک الگوریتم ساده را ارائه میدهد که حتی مسئله یک میلیون وزیر را به راحتی حل میکند.

۳-۲-۲- مسائل دنیای واقعی

مسیریابی

قبلاً دیدیم که چطور مسیریابی تحت عنوان مکانها و ارتباطات ویژه در طول اتصالات بین آنها ، تعریف می‌شود. الگوریتم‌های مسیریابی کاربردهای فراوانی دارند، مانند مسیریابی در شبکه‌های کامپیوتری، برنامه‌ریزی عملیات نظامی و سیستم‌های برنامه‌ریزی مسافرتی هوایی. بیان مشخصات این مسائل معمولاًبسیار پیچیده هستند. نمونه ساده شده مسئله خطوط هوایی را در نظر بگیرید:

- حالات: هر کدام با یک مکان (به عنوان مثال فرودگاه) و زمان حاضر نمایش داده می‌شود.
- حالت اولیه: توسط مسئله تعیین میشود.
- تابع پسین: حالاتی را برمی‌گرداند که از انتخاب هر پرواز برنامه‌ریزی شده بدست می‌آید که زمان پرواز آن بعد از زمان کنونی به اضافه زمان انتقال از این فرودگاه به فرودگاه مقصد است.
- تست هدف: آیا در زمان مورد نظر در مقصد هستیم؟
- هزینه مسیر: بستگی دارد به هزینه مالی،زمان انتظار، زمان پرواز، پروسه امور مهاجرت، کیفیت صندلی، زمان پرواز، نوع فرودگاه، تخفیف برای مسافرانی که زیاد پرواز میکنند.

سیستم‌های تجاری مشاور مسافرت از این نوع فرموله‌سازی همراه با پیچیدگی‌های بیشتری برای مسئله استفاده می‌کنند.

مسئله‌های مسافرتی بسیار نزدیک به مسائل مسیریابی هستند امّا یک تفاوت عمده دارند. مسئله "هر شهر در شکل۳-۲ را حداقل یکبار ملاقات کنید به طوری که شروع و پایان در بخارست باشد" را در نظر بگیرید. مانند مسأله مسیریابی، کنشها متناظر با سفر بین شهرهای مجاور هستند. اما برای این مسئله، فضای حالت باید اطلاعات بیشتری را حفظ کند. علاوه بر مکان عامل، هر حالت باید مجموعه شهرهایی را که عامل ملاقات کرده، نگه دارد. بنابراین حالت اولیه "In Bucharest;visited{Bucharest}" خواهد بود.

یک حالت واسط "In Vaslui;visited{Bucharest,Urziceni,Vaslui}" خواهد بود و تست هدف کنترل خواهد کرد که آیا عامل در بخارست است و تمام ۲۰ شهر را ملاقات کرده است.

TSP

مساله فروشنده دوره گرد (TSP) مسئله مشهوری است که در آن هر شهرحداقل یکبار باید ملاقات شود. هدف یافتن کوتاهترین مسیر است. این مسئله NP- سخت است. اما کوشش‌های بسیاری برای بهبود قابلیت‌های الگوریتم TSP انجام شده است. علاوه بر برنامه‌ریزی سفر برای فروشنده دوره گرد این الگوریتم‌ها برای اعمالی نظیر برنامه‌ریزی حرکات مته خودکار سوراخ کننده برد مدار استفاده می‌شود.

طرح‌بندی VLSI

مسئله طرح‌بندی VLSI نیازمند قرار دادن میلیون‌ها قطعه و مشخص کردن اتصالات آن‌ها برروی یک چیپ به گونه‌ای است که مکان اشغال شده توسط آن کمینه، تاخیر مدار کمینه و بهره کارخانه بیشینه شود. مسئله طرح‌بندی بعد از مرحله فاز طراحی منطقی بوده و به دو بخش اصلی تقسیم می‌شود: طرح‌بندی سلول و مسیریابی کانال. در طرح‌بندی سلولی، قطعات اصلی مدار در گروه‌هایی به نام سلول قرار می‌گیرند که هریک از آن‌ها وظایف خاصی را انجام می‌دهند. هر سلول دارای سایز و شکل مشخصی است و نیازمند اتصالات مشخصی به دیگر سلول‌ها است. هدف قرار دادن سلول‌ها برروی چیپ بدون هم‌پوشانی آن‌ها بایکدیگر و در نظر گرفتن فضایی برای اتصالات بین سلول‌هاست. مسیریابی کانال مسیر خاصی را برای هر سیم در بین سلول‌ها پیدا می‌کند. مسائل جستجوی این چنینی به شدت پیچیده هستند. در فصل ۴ با الگوریتم‌هایی که قابلیت حل این مسائل را دارند آشنا می‌شویم.

رهیابی ربات

رهیابی ربات تعمیم مسئله مسیریابی است که پیشتر درباره آن سخن گفتیم. یک ربات می‌تواند در یک فضای پیوسته با یک مجموعه نامحدود از حالات و کنش‌های ممکن حرکت کند. برای یک ربات چرخشی ساده که روی یک سطح صاف حرکت می کند فضا دو بعدی است. زمانی که ربات دارای بازو و پا است نیاز به کنترل دارد و در اینصورت فضای حالت چند بعدی می‌شود. تکنیک‌های پیشرفته‌ای مورد نیاز است تا فضای جستجو را محدود سازند. علاوه برآن رباتهای واقعی باید قابلیت تصحیح اشتباهات را در خواندن حسگرها و کنترل موتور داشته باشند.

مونتاژ خودکار

مونتاژ اشیای پیچیده توسط ربات اولین بار توسط رباتی به نام FREDDY (میچی، ۱۹۷۲) انجام شد. بعد از آن پیشرفت کند بود ولی عملی بودن مونتاژ اشیایی مانند موتورهای الکتریکی از نظر اقتصادی قابل توجیه بود. در مسائل مونتاژ مشکل یافتن قانونی برای جمع‌آوری تکه‌های مختلفی از اشیاء است. اگر ترتیب نادرست انتخاب شود باید تمامی مراحل بعدی مجددا تکرار شوند.

تست درستی هر بخش از مراحل تولید شده در راستای یک مسئله جستجوی پیچیده هندسی است که ارتباط نزدیکی با رهیابی ربات دارد. از این رو بدست آوردن یک برنامه درست یکی از پر هزینه‌ترین مراحل مونتاژ است و استفاده از الگوریتمهای آگاهانه برای کاهش جستجو ضروری است که به کاهش هزینه‌ها می‌انجامد. یکی از مهم‌ترین مسائل مونتاژ، طراحی پروتئینهاست که در آن هدف پیدا کردن رشته‌ای از آمینو اسیدهاست که به پروتئین سه بعدی با ویژگی‌های خاص برای درمان بیماری تبدیل شوند. در سالهای اخیر تقاضاها برای رباتهای نرم‌افزاری جستجوی اینترنتی برای یافتن پاسخ پرسشها، و یا یافتن اطلاعات مرتبط با یک موضوع ویا انجام قراردادهای خرید به شدت افزایش یافته است. این موضوع کاربردی مناسب برای تکنیکهای جستجو می‌باشد زیرا تبدیل مفهوم اینترنت به عنوان گرافی از گره‌ها (صفحات) که توسط لینک‌ها به هم وصل شده‌اند آسان است.

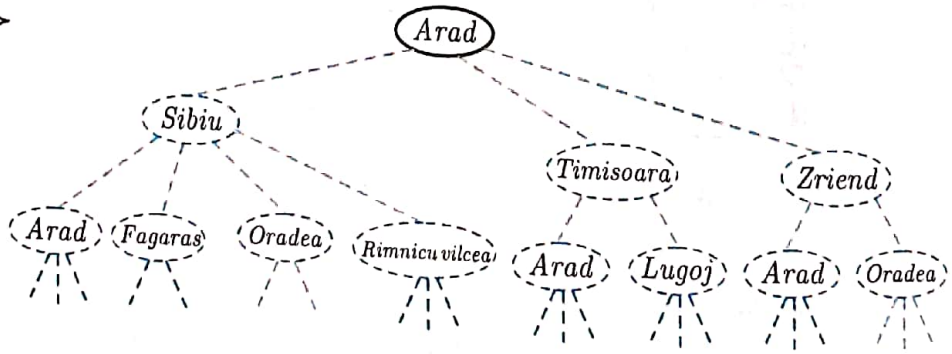
۳-۳- جستجو برای پیدا کردن راه‌حل

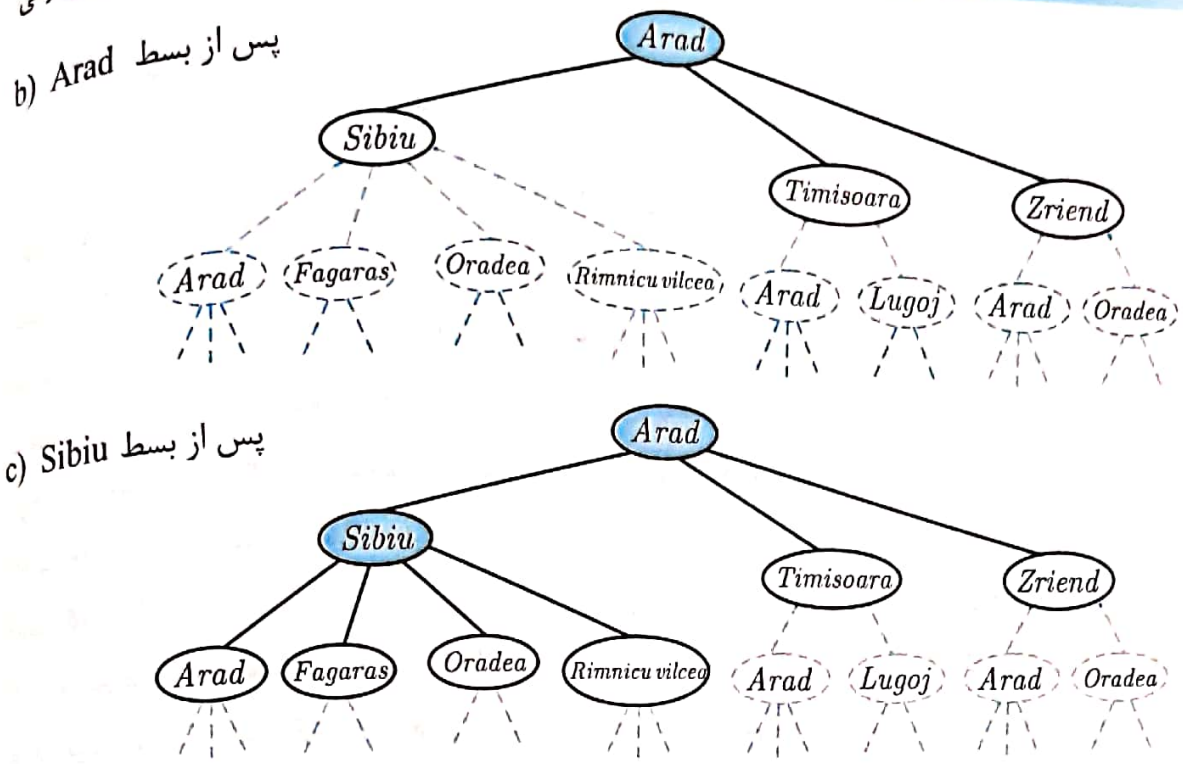
پس از فرموله‌سازی مسئله، باید آن را حل کنیم. این کار با جستجو در فضای حالت انجام می‌شود. این فصل تکنیک‌های جستجویی را معرفی می‌کند که از درخت جستجویی استفاده می‌کنند که از حالت اولیه و تابع پسین (که در مجموع فضای حالت را تشکیل میدهند) ساخته می‌شود. در حالت کلی، وقتی یک حالت از چند مسیر قابل دسترس باشد آنگاه به جای درخت جستجو یک گراف جستجو خواهیم داشت.

شکل ۳-۶ برخی از بسط‌های درخت جستجو برای یافتن مسیری از آراد به بخارست را نشان می‌دهد. ریشه درخت جستجو یک گره جستجو متناظر با حالت اولیه In(Arad) است. در شکل ۳-۶- گره‌هایی که در داخل ناحیه خط چین هستند عملاً تولید نشده‌اند و در مراحل بعدی ممکن است تولید شوند. گام اول بررسی این است که آیا این حالت یک حالت هدف است یا خیر که به وضوح این طور نیست (اما این بررسی ضروری است که بتوانیم مسئله‌ای همچون شروع از آراد برای رفتن به آراد را حل کنیم). از آنجایی که این حالت یک حالت هدف نیست، لازم است که حالت‌های بیشتری را بررسی کنیم. این کار با بسط حالت جاری، با استفاده از تابع پسین بر روی حالت کنونی و تولید یک مجموعه جدید از حالت‌ها انجام می‌شود. در این مورد سه حالت جدید داریم: In(Sibiu)، In(Timisoara) و In(Zerind). حال باید یکی از این حالات محتمل را برای بررسی بیشتر انتخاب کنیم.

در واقع اساس جستجو بدین شکل است: انتخاب یک گزینه و کنار گذاشتن بقیه برای بعد (در مواردی‌که گزینه‌ی اول به حل مسئله منجر نشود). فرض کنید که ما سیبیو را انتخاب کردیم. ابتدا کنترل می‌کنیم که این حالت هدف نباشد و سپس آن را دنبال می کنیم تا به In (Arad)، In(Oradea)، In(Fagaras) و In(Rimnicu Vilcea) برسیم. سپس می توانیم یکی از این ۴ حالت را انتخاب و یا به عقب برگردیم و تی میسوار یا زرینند را انتخاب کنیم.فرآیند تست و انتخاب را تا زمانی که راه‌حلی انتخاب نشده ادامه می‌دهیم مگر اینکه دیگر هیچ حالت دیگری برای گسترش وجود نداشته باشد. انتخاب اینکه کدام حالت باید بسط پیدا کند توسط استراتژی جستجو تعیین می‌شود. در شکل ۳-۷ الگوریتم کلی درخت جستجو به طور غیر رسمی توصیف شده است.

a) حالت اولیه

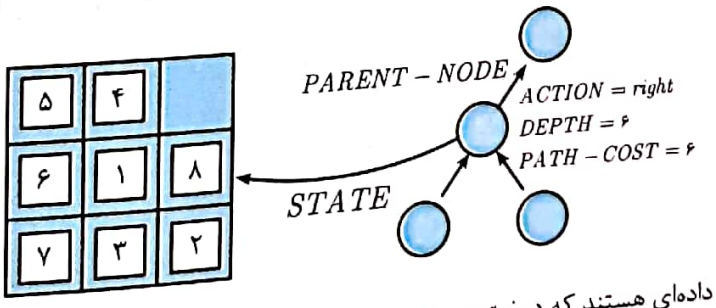




شکل ۳-۶ بخشی از درخت جستجو برای پیدا کردن مسیر از آراد به بخارست. گره‌های بسط یافته، خاکستری و گره‌هایی که تولید شده ولی بسط نیافته‌اند، پررنگ شده‌اند. گره‌های تولید نشده با خط چین نشان داده شده‌اند.

```
function TREE-SEARCH (problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

شکل ۳-۷ توصیف غیر رسمی الگوریتم عمومی جستجوی درختی



شکل ۳-۸ گره‌ها ساختار داده‌ای هستند که درخت جستجو را تشکیل میدهند. هرگره شامل والد، یک حالت و غیره، می‌باشد. فلش‌ها از فرزند به سمت والد اشاره می‌کنند.

مهم است که بین فضای حالت و درخت جستجو فرق قائل شویم. برای مسئله مسیریابی فقط ۲۰ حالت در فضای حالت وجود دارد (یکی برای هر شهر). اما تعداد نامحدودی از مسیرها در این فضای حالت وجود دارند بنابراین درخت جستجو تعداد نامحدودی گره دارد.

راههای زیادی برای نمایش گره‌ها وجود دارد اما در این فصل ما گره را به عنوان یک ساختار داده با پنج قسمت زیردر نظر می‌گیریم:

- حالت: حالتی در فضای حالت که گره متناظر با آن است.

- گره والد: گره‌ای که در جستجوی درخت گره مورد نظر را تولید کرده است.
- کنشی که بر گره والد اعمال شده و گره مورد نظر را تولید کرده است.
- عمق: تعداد گام‌های مسیر از ریشه تا گره مورد نظر.
- هزینه مسیر: این هزینه با g(n) نمایش داده‌می‌شود، هزینه مسیر از حالت اولیه تا گره است که با اشاره‌گرهای گره والد مشخص می‌شود.

اینکه بین گره‌ها و حالت‌ها تفاوت قائل شویم مسئله مهمی است. یک گره نگهدارنده ساختار داده‌ای است که توسط یک الگوریتم برای نمایش درخت جستجوئی که در یک مسئله خاص تولید شده است استفاده می شود. در حالیکه یک حالت بیان‌کننده مشخصه‌ای از دنیا است. از این رو گره‌ها (برخلاف حالات) دارای والد هستند. بعلاوه داشتن یک حالت مشابه برای دو گره متفاوت امکان‌پذیر است زیرا ممکن است آن حالت توسط دو دنباله مختلف از کنش‌ها تولید شده باشد. ساختارداده گره در شکل ۳-۸ نشان داده شده است.

همچنین نیاز به جمع‌آوری گره‌هایی داریم که تولید شده ولی بسط نیافته‌اند. این مجموعه حاشیه[1] نامیده میشود. هر عنصر در حاشیه، یک گره برگ است، یعنی گره‌ای که در درخت فرزندی (منظور همان حالت پسین است) ندارد. در شکل ۳-۶، حاشیه‌های درخت با خطوط خارجی پررنگ‌تر نمایش داده شده‌اند. ساده‌ترین نوع نمایش حاشیه مجموعه‌ای از گره‌ها است. استراتژی جستجو تابعی خواهد بود که گره بعدی را (که در واقع گره‌ای است که برای بسط انتخاب می‌شود) از حاشیه انتخاب می‌کند. اگرچه از نظر ذهنی این امر بسیار ساده است ولی از نظر محاسباتی بسیار گران تمام می شود. زیرا تابع استراتژی می بایست تمام عناصر مجموعه را به منظور انتخاب بهترین عنصر بررسی کند. بنابراین فرض می‌کنیم که مجموعه عناصر به صورت یک صف پیاده‌سازی می‌شوند. عملیات روی یک صف به صورت زیر می باشد:

- MAKE-QUEUE (elements): صفی را با عناصر داده شده تشکیل می‌دهد.
- EMPTY? (queue): در صورتی که صف فاقد عنصر باشد مقدار Trueبرمی‌گرداند.
- FIRST(queue): عنصر اول صف را برمیگرداند.
- REMOVE-FIRST (Queue): First(queue) را برمیگرداند و آن را از صف حذف میکند.
- INSERT (element, queue): عنصر را در صف قرار داده و صف حاصل را برمیگرداند. صف‌های متفاوت عناصر را با ترتیب‌های متفاوتی وارد می‌کنند.
- INSERT-ALL (elements, queue): عناصر را به صف اضافه کرده و صف حاصل را برمی‌گرداند.

با این تعاریف می‌توانیم صورت رسمی‌تری از الگوریتم جستجوی عمومی را بنویسیم. این الگوریتم در شکل ۳-۹ نشان داده شده است.

۳-۳-۱- ارزیابی کارایی حل مسئله

خروجی الگوریتم‌های حل مسئله یا شکست و یا راه‌حل خواهد بود. (برخی الگوریتم‌ها ممکن است در حلقه بینهایت به دام افتاده و هیچ خروجی برنگردانند). کارایی یک الگوریتم به چهار روش ارزیابی می‌شود:

- کامل بودن[2]: آیا الگوریتم تضمین می‌دهد که راه‌حلی را پیدا کند؟ (در صورتی که وجود داشته باشد.)

---

[1]- Fringe
[2]-Complete

- بهینه بودن[۱]: آیا استراتژی راه‌حل بهینه را می‌یابد؟
- پیچیدگی زمانی[۲]: پیدا کردن یک راه‌حل چقدر طول می کشد؟
- پیچیدگی حافظه[۳]: چقدر حافظه برای انجام جستجو مورد نیاز است؟

پیچیدگی زمان و حافظه همیشه با توجه به معیاری از سختی مسئله در نظر گرفته میشود. در علوم کامپیوتر نظری، این معیار اندازه گراف فضای حالت است. زیرا به گراف به عنوان ساختار داده‌ای مشخص نگاه می‌شود که به عنوان ورودی به برنامه جستجو داده میشود (نقشه رومانی نمونه‌ای از این است). در AI که گراف به طور غیر مستقیم توسط حالت اولیه و تابع پسین نشان داده میشود و معمولاًبینهایت است، پیچیدگی با سه مقدار تعیین میشود: b، فاکتور انشعاب یا بیشترین تعداد بچه‌های یک گره، d عمق کم عمق‌ترین هدف و m بیشترین طول مسیر در فضای حالت.

```
function TREE-SEARCH (problem,fringe) returns a solution, or failure
    fringe ← INSERT (MAKE-NODE (INITIAL-STAte [problem]), fringe)
    loop do
        if EMPTY? (fringe) then return failure
        node ←REMOVE-FIRST (fringe)
        if GOAL- TEST [problem] applied to STATE [node] succeeds
            then return SOLUTION (node)
        fringe ←INSERT- ALL (EXPAND (node-problem),fringe
```

```
function EXPAND(node,problem) returns a set of nodes

    successors ← the empty set
    for each(action, result) in SUCCESSOR-FN[problem](STATE[node]) do
        s← a new NODE
        STATE[s]← result
        PARENT-Node[s]← node
        ACTION[s]← action
        PATH-COST[s]← PATH-COST[node]+STEP-COST(node, action, S)
        DEPTH [s]← DEPTH[node] + ۱
        add s to successor
    return successors
```

شکل ۳-۹ الگوریتم عمومی جستجوی درختی (دقت کنید که صف باید خالی باشد و نوع صف ترتیب جستجو را تغییر می‌دهد). راه‌حلی که تابع برمیگرداند رشته‌ای از کنش‌هاست که با ادامه‌دادن مسیر از اشاره‌گر والد تا ریشه به‌دست می‌آید.

زمان معمولاً با تعداد گره‌های تولید شده[۴] در طول فضای جستجو و فضا با بیشترین گره‌های ذخیره شده در حافظه اندازه‌گیری می‌شوند.

۱- Optimal
۲- Time complexity
۳- Space complexity
۴- بعضی متون زمان را با تعداد گره‌های بسط داده شده اندازه می‌گیرند. دو معیار تنها با مضربی از b تفاوت دارند.

برای محاسبه میزان تأثیر الگوریتم جستجو، می‌توانیم تنها هزینه جستجو را در نظر بگیریم که معمولاً به پیچیدگی زمانی بستگی دارد(ولی می‌تواند شامل فاکتور میزان مصرف حافظه نیز باشد). و یا می‌توانیم از فاکتور هزینه کلی استفاده کنیم که هزینه جستجو را با هزینه مسیر راه‌حل یافت‌شده ترکیب می‌کند. در مسئله پیدا کردن مسیری از آراد به بخارست، هزینه جستجو، زمانی است که الگوریتم جستجو برای پیدا کردن راه‌حل نیاز دارد و هزینه راه‌حل طول مسیر به کیلومتر است. پس برای محاسبه هزینه کلی می‌توانیم کیلومترها و میلی‌ثانیه‌ها را با هم جمع بزنیم. این کار همیشه ساده نیست، چون هیچ "معیار مقایسه معتبری "بین این دو وجود ندارد. در اینجا عامل می‌تواند با استفاده از سرعت متوسط تخمینی خودرو کیلومتر را به زمان تبدیل کند، زیرا چیزی که برای عامل اهمیت دارد زمان است.این موضوع به عامل اجازه می‌دهد نقطه مصالحه بهینه را انتخاب کند.

۳-۴- استراتژی‌های جستجوی ناآگاهانه

این بخش شامل پنج استراتژی جستجو تحت عنوان جستجوی ناآگاهانه است (گاهی به این جستجو، جستجوی کورکورانه نیز می گویند). منظور این است که به جز اطلاعات فراهم شده توسط صورت مسئله، اطلاعات دیگری در رابطه با حالات وجود ندارد. در واقع، در این نوع جستجوها تنها بسط حالتها و تست یک حالت هدف درمقابل حالت غیر هدف امکان‌پذیر است.

در مقابل، استراتژی‌هایی وجود دارند که ازپیش اطلاعاتی در مورد اینکه کدام حالت غیرهدف "نوید بخش‌تر" است، دارند که به آن‌ها جستجوهای آگاهانه یا ابتکاری گفته می‌شود. در فصل ۴ به تفصیل درباره آنها صحبت خواهیم کرد. همه‌ی استراتژی‌های جستجو براساس ترتیب بسط گره‌ها از یکدیگر تمییز داده می‌شوند.

۳-۴-۱- جستجوی سطح- اول[۱]

یک استراتژی آسان جستجو، جستجوی سطح-اول است. در این روش، ابتدا گره ریشه بسط می‌یابد، سپس همه گره‌های ایجادشده توسط گره ریشه (فرزندانش) بسط می یابند و سپس فرزندان(حالت پسین) آنها، و به همین ترتیب تا آخر. در کل، همه گره‌های با عمق d در درخت جستجو، قبل از گره‌های با عمق $d+1$ بسط می‌یابند.

جستجوی سطح- اول با صدا زدن الگوریتم جستجوی عمومی با یک رویه صف‌بندی که گره‌های تازه تولید شده را در ته صف (بعد از همه گره‌های ایجاد شده قبلی) می‌گذارد قابل پیاده‌سازی است. به عبارت دیگر فراخوانی TREE-SEARCH(problem, FIFO-QUEUE) الگوریتم سطح-اول را نتیجه می‌دهد. صف با استراتژی اولین ورودی- اولین خروجی همه فرزندهای(حالت پسین) جدید تولید شده را در ته صف قرار می‌دهد یعنی گره های کم‌عمق قبل از گره‌های عمیق بسط می‌یابند. شکل ۳-۱۰ روند جستجو را روی یک درخت دودویی ساده نشان می‌دهد. الگوریتم سطح-اول را با ۴ معیار بخش قبل ارزیابی می‌کنیم. جستجوی سطح- اول کامل است. زیرا اگر کم‌عمق‌ترین گره هدف در عمق d باشد، جستجوی سطح- اول جواب را پس از بسط تمام گره‌های کم-عمق‌تر پیدا خواهد کرد (اگر فاکتور انشعاب b محدود باشد).

کم‌عمق‌ترین هدف لزوماً بهینه نیست. اصولاً جستجوی سطح-اول بهینه است اگر هزینه مسیر یک تابع غیر نزولی از عمق گره باشد. (برای مثال، وقتی همه کنشها هزینه یکسان دارند.)

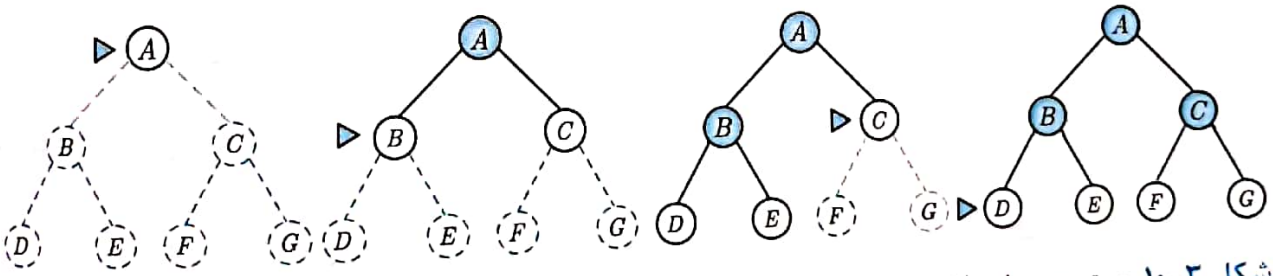
---

[۱]- Breadth-first search

تا به حال، نتایج جستجوی سطح-اول خوب بوده است. برای اینکه ببینیم چرا همیشه این استراتژی انتخاب نمی‌شود، باید زمان و هزینه‌ای را که برای کامل کردن جستجو صرف می‌کند را در نظر بگیریم. برای انجام این کار، یک فضای حالت فرضی را در نظر می‌گیریم که هر حالت آن می تواند b حالت جدید تولید کند. ریشه درخت جستجو، در اولین سطح، b گره تولید می‌کند که هر کدام b گره دیگر تولید می‌کنند، که جمع تعداد گره ها در سطح ۲ برابر b^2 می‌شود. هر کدام از این‌ها b گره دیگر تولید می‌کنند و در سطح سوم b^3 گره تولید می‌شود، و همین طور الی آخر. حال فرض کنید جواب مساله مسیری به طول d است. در بدترین حالت، باید همه گره‌ها را به جز آخرین گره در سطح d بررسی کنیم (چون خود هدف بسط پیدا نمی‌کند) که $b^{d+1} - b$ گره در سطح $d + 1$ تولید می‌شود. بنابراین بیشترین تعداد گره تولید شده برابر است با:

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

تمام گره‌های تولید شده باید در حافظه باقی بمانند زیرا یا بخشی از صف و یا جد یکی از گره‌های صف هستند. بنابراین پیچیدگی حافظه با پیچیدگی زمانی برابر است (به علاوه یک گره برای ریشه). افرادی که با تحلیل پیچیدگی سر و کار دارند، وقتی حد پیچیدگی نمایی‌ای مانند $O(b^{d+1})$ را می‌بینند نگران می‌شوند. شکل ۳-۱۱ علت این موضوع را نشان می‌دهد. این شکل زمان و حافظه مورد نیاز برای یک جستجوی سطح-اول با فاکتور انشعاب $b = ۱۰$ و مقادیر مختلف عمق جواب d را نشان می دهد. این جدول فرض می‌کند که در هر ثانیه ۱۰۰۰۰ گره میتوانند برای تست هدف بودن یا نبودن بررسی شوند و بسط پیدا کنند و هر گره ۱۰۰۰ بایت حافظه نیاز دارد. بسیاری از مسایل جستجو وقتی روی یک کامپیوتر شخصی مدرن اجرا می‌شوند، تقریباً با این فرضیات متناسبند.(کم و بیش با فاکتور ۱۰۰)



شکل ۳-۱۰ جستجوی سطح- اول بر روی یک درخت دودویی ساده. در هر مرحله، گره‌ای که باید بسط یابد توسط علامت مثلث نشان داده‌شده‌است.

Depth	Nodes	Time	Memory
۲	۱۱۰۰	۱۱ seconds	۱ megabyte
۴	۱۱۱,۱۰۰	۱۱ seconds	۱۰۶ megabytes
۶	$۱۰^۷$	۱۹ minutes	۱۰ gigabytes
۸	$۱۰^۹$	۳۱ hours	۱ terabytes
۱۰	$۱۰^{۱۱}$	۱۲۹ days	۱۰۱ terabytes
۱۲	$۱۰^{۱۳}$	۳۵ years	۱۰ petabytes
۱۴	$۱۰^{۱۵}$	۳,۵۲۳ years	۱ exabyte

شکل ۳-۱۱ مصرف زمان و حافظه برای جستجوی سطح- اول. اعداد نشان داده شده با فرض $b = ۱۰$، ۱۰۰۰۰ گره بر ثانیه و ۱۰۰۰ بایت برای هر گره.

شکل ۳-۱۱ دو درس را به ما می‌آموزد. اول، برای جستجوی سطح- اول نیازمندی‌های حافظه مشکل بزرگتری از زمان اجرا است. زیرا اگرچه ۳۱ ساعت زمان زیادی برای به دست آوردن راه‌حل یک مسئله مهم در عمق ۸ نیست اما تعداد خیلی کمی از کامپیوترها حافظه‌ای با حجم ترابایت دارند. خوشبختانه، استراتژی‌های جستجوی دیگری نیز وجود دارند که نیاز به حافظه کمتری دارند.

درس دوم این است که نیازمندی‌های زمانی هنوز فاکتور بسیار مهمی محسوب می‌شود. اگر مساله شما، جوابی در عمق ۱۲ داشته باشد (با توجه به فرضیات ما) یک جستجوی ناآگاهانه ۳۵ سال طول می‌کشد. در کل، مسایل جستجو با پیچیدگی نمایی(بجز در ابعاد کوچک) با الگوریتم‌های جستجوی ناآگاهانه قابل حل نیستند.

۳-۴-۲- جستجوی هزینه-یکنواخت[1]

جستجوی سطح- اول بهینه است در صورتی که هزینه همه گام‌ها برابر باشند، زیرا همیشه کم‌عمق‌ترین گره بسط نیافته را بسط می‌دهد. پیشرفت ساده‌ای در ایده‌ی این جستجو ما را به الگوریتمی می‌رساند که با هر تابع هزینه‌ای (منظور تابع هزینه گام‌های الگوریتم است) بهینه است. به جای بسط کم‌عمق‌ترین گره، جستجو با هزینه‌ی یکنواخت، گره n با کمترین هزینه مسیر را بسط می‌دهد. دقت کنید که اگر هزینه‌های همه‌ی گام‌ها برابر باشند، این الگوریتم همان جستجوی سطح ـ اول است.

در جستجوی هزینه-یکنواخت تعداد گامهای مسیر مهم نیست و فقط هزینه‌ی کل اهمیت دارد. بنابراین، اگر گره‌ای با کنشی با هزینه صفر که به نقطه‌ی کنونی برمی‌گردد مثل کنش NoOp (انجام ندادن هیچ کاری) را بسط دهد، در حلقه‌ی بی‌نهایت گیر می‌کند. بنابراین الگوریتم کامل است اگر هزینه‌ی هر گام بزرگتر یا مساوی یک مقدار ثابت مثبت کوچک ϵ باشد. این شرایط برای بهینه بودن نیز کافی است. یعنی هزینه‌ی مسیر همیشه در طول مسیر افزایش می‌یابد. از این مشخصه معلوم است که الگوریتم گره‌ها را به ترتیب افزایش هزینه‌ی مسیر بسط می‌دهد. بنابراین اولین گره هدف انتخاب شده برای بسط، راه حل بهینه است (به خاطر داشته باشید جستجوی درختی آزمون هدف را بر روی گره‌هایی که برای بسط انتخاب می‌شوند اعمال می‌کند). جستجوی هزینه-یکنواخت، به جای عمق گره با هزینه مسیر راهنمایی می‌شود، بنابراین پیچیدگی آن با b و d به راحتی بیان نمی‌شود. به جای آن اجازه دهید C^* را به عنوان هزینه‌ی جواب بهینه داشته باشیم و هر کنش حداقل هزینه ϵ داشته‌باشد. بنابراین در بدترین حالت پیچیدگی زمان و حافظه ($O(b^{\lceil C^*/\epsilon \rceil})$ است که می‌تواند بسیار بزرگتر از b^d باشد. زیرا این جستجو ابتدا قسمت‌هایی از درخت را که دارای گامهای زیادی با هزینه کم هستند بررسی می‌کند وسپس به بررسی قسمت‌هایی از درخت که دارای گامهای بزرگ است (و احتمال یافتن جواب در آن بیشتر است) می‌پردازد. وقتی همه‌ی گام‌ها برابر باشند، $b^{\lceil C^*/\epsilon \rceil}$ برابر با b^d می‌شود.

[1]-Uniform-cost search

۳-۴-۳- جستجوی عمق- اول[۱]

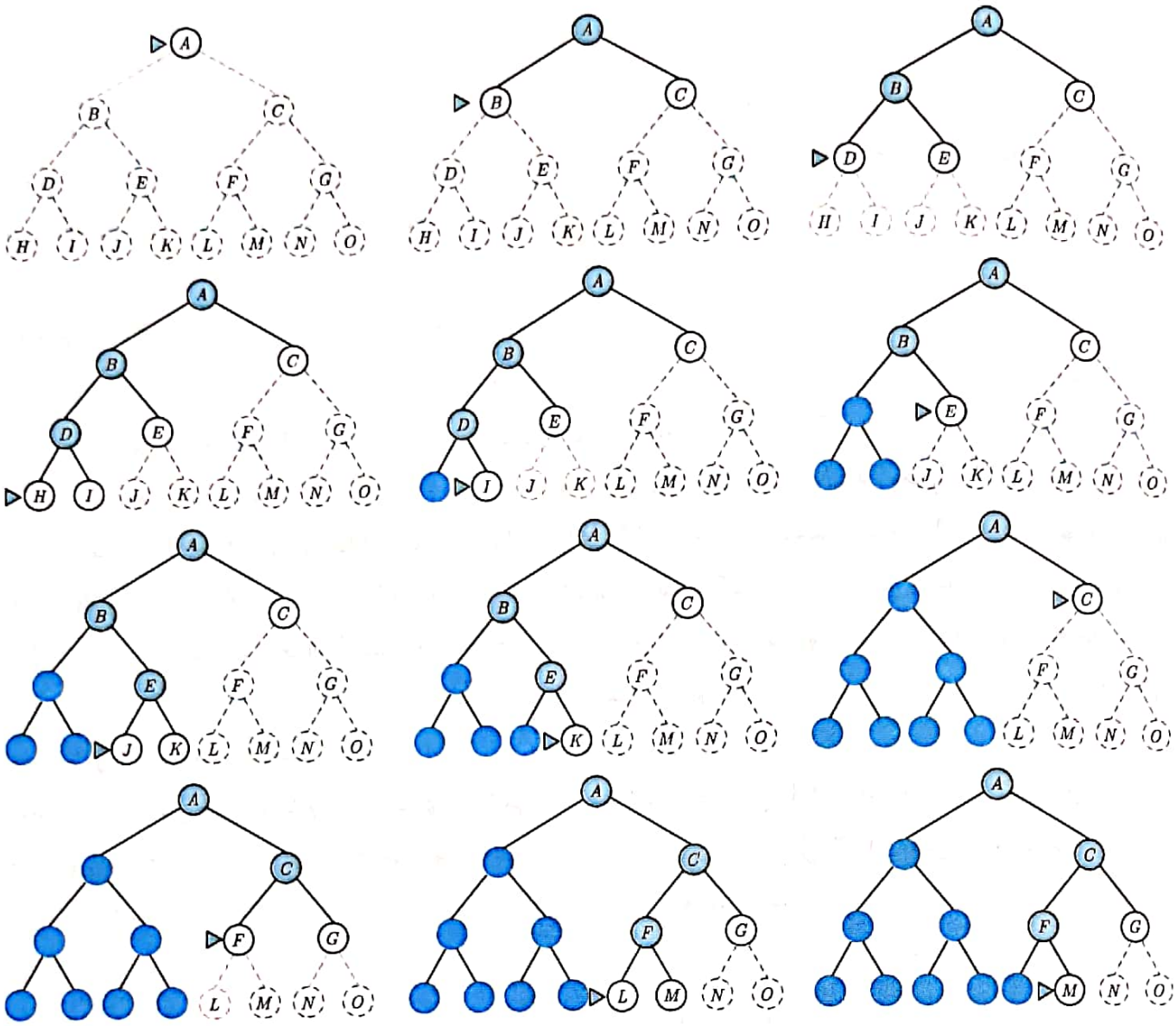
جستجوی عمق- اول همیشه عمیق‌ترین گره را در حاشیه جاری درخت جستجو بسط می‌دهد. روند جستجو در شکل ۳-۱۲ شرح داده شده است. جستجو بلافاصله به عمیق‌ترین سطح درخت جستجو می‌رود که گره‌ها فرزندی ندارند. بعد از بسط، این گره‌ها از حاشیه بیرون انداخته می‌شوند و جستجو به عمیق‌ترین گره بعدی که فرزندانش هنوز بررسی نشده‌اند حرکت می‌کند.

این استراتژی با جستجوی درختی و با LIFO (آخرین ورودی-اولین خروجی) پیاده‌سازی می‌شود که پشته نامیده میشود. علاوه بر پیاده‌سازی جستجوی درختی، پیاده‌سازی جستجوی عمق-اول با تابع بازگشتی که فرزندانش را به نوبت فراخوانی می‌کند نیز پرکاربرد است (یک الگوریتم عمق- اول بازگشتی با محدودیت عمق در شکل ۳-۱۳ نشان داده شده است) جستجوی عمق-اول نیاز به حافظه کمی دارد و فقط نیاز دارد که یک مسیر از ریشه تا برگ را همراه با گره‌های برادر باقی‌مانده‌ی بررسی نشده (برای هر گره روی مسیر) ذخیره کند. وقتی یک گره بسط یافت، بعد از اینکه همه‌ی فرزندانش بررسی شدند از حافظه پاک می‌شود (شکل ۳-۱۲) برای هر فضای حالت با فاکتور انشعاب b و بیشترین عمق m، جستجوی عمق-اول نیاز دارد که $bm + 1$ گره را ذخیره کند. با استفاده از فرضیاتی مشابه شکل ۳-۱۱ و با فرض اینکه گره‌های با عمق هدف گره پسین ندارند در می‌یابیم که جستجوی عمق- اول ۱۱۸ کیلوبایت به جای ۱۰ پتابایت در عمق $d = ۱۲$ حافظه نیاز دارد (حافظه‌ای کمتر با فاکتور ۱۰ میلیون).

نوع دیگر از جستجوی عمق-اول، به نام جستجوی عقبگرد، از حافظه کمتری استفاده می‌کند. در عقبگرد به جای همه گره‌های پسین در هر لحظه تنها یک گره پسین تولید می‌شود. هر گره‌ای که بسط می‌یابد، فرزند بعدی‌ای که باید تولید شود را به یاد می‌سپارد. به این طریق تنها حافظه $O(m)$ به جای $O(bm)$ لازم است. جستجوی عقبگرد راه دیگری نیز برای صرفه‌جویی در حافظه و زمان دارد: ایده تولید یک گره پسین با استفاده از تغییر توصیف حالت جاری به جای کپی کردن آن. این کار تنها نیاز به حافظه‌ای برابر یک توصیف حالت و $O(m)$ کنش دارد. برای این منظور بهنگام عقب‌گرد و تولید گره پسین بعدی باید بتوانیم تغییرات را به حالت اول برگردانیم. برای مسائلی با توصیف حالت بزرگ مثل مونتاژ ربات، این تکنیکها برای موفقیت حیاتی هستند.

مشکل جستجوی عمق-اول این است که می‌تواند با یک انتخاب اشتباه در یک مسیر طولانی یا بینهایت پیشرفت کرده و گیر بیفتد. درحالیکه ممکن بود در مسیر دیگری نزدیک ریشه به جواب برسد. برای مثال، در شکل ۳-۱۲ حتی اگر C گره هدف باشد، جستجوی عمق-اول، کل زیر درخت سمت چپ را بررسی می‌کند. همچنین اگر گره J هدف می‌بود، آنگاه الگوریتم آن را به عنوان راه‌حل مسئله برمی‌گرداند. درحقیقت جستجوی عمق- اول بهینه نیست. اگر زیر درخت سمت چپ نامحدود باشد و حتی راه‌حل مسئله در آن وجود نداشته باشد بازهم هیچگاه الگوریتم خاتمه نمی‌یابد. در بدترین حالت، جستجوی عمق-اول همه‌ی $O(bm)$ گره در درخت جستجو را تولید می‌کند (اگر m را ماکزیمم عمق گره در نظر بگیریم). دقت کنید که m میتواند بسیار بزرگتر از d(عمق کم‌عمق‌ترین جواب) باشد و اگر درخت محدود نباشد مقدار آن، بی‌نهایت است.

۱- Depth-first search



شکل ۳-۱۲ جستجوی عمق- اول روی درخت دودویی. گره‌های بسط داده شده‌ای که فرزندی (descendants) در حاشیه ندارند از حافظه حذف می‌شوند (در شکل با رنگ مشکی نمایش داده شده‌اند). گره‌های عمق ۳ فاقد گره پسین هستند و M تنها گره هدف است.

```
function DEPTH-LIMITED-SEARCH (problem,limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS (MAKE- NODE (INITIAL-STATE [problem]), problem,limit)

function RECURSIVE-DLS (node, problem,limit) returns a solution, or failure/cutoff
    cutoff-occurred?← false
    if GOAL-TEST[problem] (STATE [node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ←Recursive-DLS(successor,problem, limit)
        if result = cutoff then cutoff-occurred? ←true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

شکل ۳-۱۳ پیاده‌سازی بازگشتی جستجوی با عمق محدود

۳-۴-۴- جستجوی با عمق محدود[۱]

جستجو با عمق محدود، مشکل درخت نامحدود را با تعیین کردن محدودیت عمق l روی بیشینه عمق مسیر، حل می‌کند. یعنی با گره‌های عمق l مشابه برگ رفتار می‌شود. این روش را جستجو با عمق محدود می‌نامند. این محدودیت عمق مشکل مسیر بینهایت را حل می‌کند. متأسفانه در این روش اگر l کوچکتر از d انتخاب شود (یعنی کم‌عمق‌ترین هدف پایینتر از محدودیت عمق قرار دارد) مشکل ناکامل بودن را ایجاد می‌کند (که در مسائلی که d نامعلوم است محتمل است). جستجوی عمق- اول همچنین غیربهینه است اگر $d \ll l$ انتخاب شود. پیچیدگی زمانی برابر با $O(b^l)$ و پیچیدگی حافظه آن برابر $O(b^l)$ است. جستجوی عمق-اول را میتوانیم به عنوان حالت خاصی از جستجوی با عمق محدود با $\infty = l$ در نظر گرفت.

گاهی، محدودیت عمق می‌تواند بر پایه‌ی دانش مسئله باشد. برای مثال روی نقشه رومانی، ۲۰ شهر وجود دارد، پس می‌دانیم اگر جوابی وجود داشته باشد، حداکثر باید دارای عمق ۲۰ باشد بنابراین $l = ۱۹$ می‌تواند گزینه خوبی باشد. اما در واقع اگر نقشه را با دقت بررسی کنیم، کشف میکنیم که هر شهر از هر شهر دیگر با ۹ گام قابل دسترس است. این تعداد "قطر" فضای حالت نامیده می‌شود که محدودیت عمق بهتری را به ما معرفی کرده که به جستجوی با عمق محدود کاراتری منجر میشود. با این حال در بیشتر مسائل، مشخص کردن این عمق، قبل از حل مسئله میسر نیست.

جستجوی با عمق محدود با تغییر ساده‌ای در الگوریتم جستجوی درختی یا الگوریتم بازگشتی جستجوی عمق-اول قابل پیاده‌سازی است. شبه کد جستجو با عمق محدود بازگشتی در شکل ۳-۱۳ نشان داده شده‌است. توجه کنید که جستجوی با عمق محدود می‌تواند با دو نوع شکست متوقف شود: شکست عادی در صورت پیدا نکردن جواب و شکست برش (cutoff) که نشان می‌دهد هیچ جوابی در محدوده عمق تعیین شده یافت نشده است.

۳-۴-۵- جستجوی عمیق‌شونده تکراری[۲]

جستجوی عمیق‌شونده تکراری، یک استراتژی به‌صورت ترکیبی با الگوریتم جستجوی عمق ـ اول به کار می‌رود و بهترین محدودیت عمق را پیدا می‌کند. این کار با افزایش محدودیت عمق انجام می‌گیرد: ابتدا عمق ۰، سپس عمق ۱، بعد عمق ۲ و همین طور الی آخر. این روند تا زمانی که محدودیت عمق به d (عمق کم عمق‌ترین هدف) برسد، ادامه پیدا می‌کند. الگوریتم در شکل ۳-۱۴ نشان داده شده است. در نتیجه جستجوی عمیق‌شونده تکراری مزایای جستجوی عمق- اول و جستجوی سطح-اول را باهم ترکیب می‌کند. مثل جستجوی سطح- اول، کامل و بهینه است، ولی فقط نیازمندی‌های حافظه نسبتا کم $O(bd)$ جستجوی عمق- اول را دارد. مقدار بسط گره‌ها مثل جستجوی سطح-اول است، بجز اینکه بعضی گره‌ها چند بار بسط می‌یابند. شکل ۳-۱۵ چهار تکرار اول جستجوی عمیق‌شونده تکراری را روی یک درخت جستجوی دودویی نشان می‌دهد.

ممکن است جستجوی عمیق‌شونده تکراری پرمصرف به‌نظر برسد زیرا هریک از حالات در آن چندین بار بسط می‌یابند. با این وجود این الگوریتم چندان پرهزینه نیست. زیرا جستجوی درختی با فاکتور انشعاب مساوی (یا

۱- Depth-Limited Search
۲- Iterative Deepening Search

تقریبا مساوی) در همه‌ی سطوح آن، اکثر گره‌ها در لایه‌های پایینی قرار دارند و بنابراین تولید چندین باره گره های بالایی چندان حائز اهمیت نیست. در واقع در این الگوریتم، گره‌های آخرین سطح، یک بار، گره‌های سطح ماقبل آخر، ۲ بار و ... تا ریشه که d بار تکرار می‌شوند. بنابراین تعداد کل گره‌های تولید شده برابراست با:

$$N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d$$

که پیچیدگی زمانی برابر $O(b^d)$ به ما می‌دهد که می‌توانیم با گره‌های تولید شده‌ی جستجوی سطح- اول مقایسه کنیم:

$$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

توجه کنید که جستجوی سطح-اول بعضی گره‌ها را در عمق $d+1$ تولید می‌کند در حالی‌که جستجوی عمیق-شونده تکراری این‌کار را نمی‌کند. در نتیجه، الگوریتم عمیق‌شونده تکراری با وجود تکرار تعدادی از حالات سریعتر از الگوریتم سطح- اول می‌باشد. برای مثال اگر $b = ۱۰$ و $d = ۵$ باشد. تعداد گره‌ها:

$$N(IDS) = ۵۰ + ۴۰۰ + ۳۰۰۰ + ۲۰۰۰۰ + ۱۰۰۰۰۰ = ۱۲۳۴۵۰$$

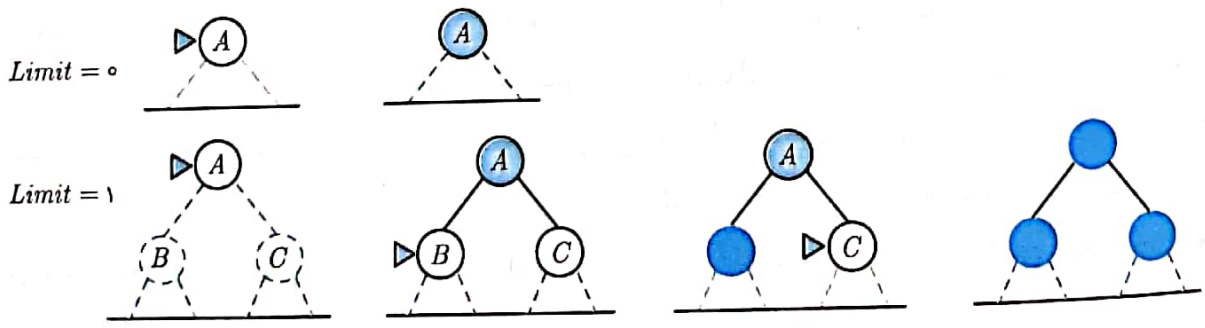
$$N(BFS) = ۱۰ + ۱۰۰ + ۱۰۰۰ + ۱۰۰۰۰ + ۱۰۰۰۰۰ + ۹۹۹۹۹۰ = ۱۱۱۱۱۰۰$$

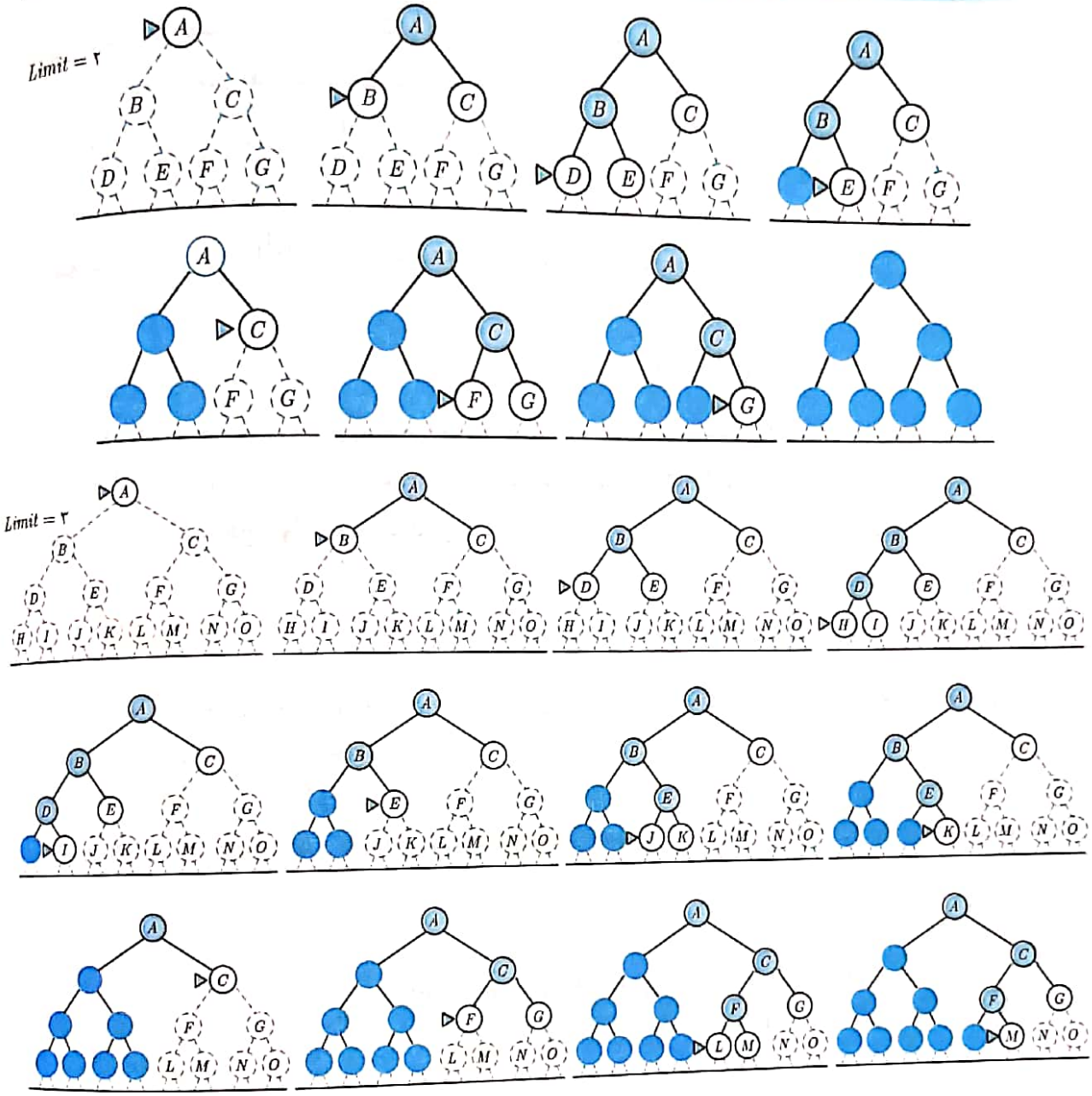
در کل، وقتی که فضای جستجوی بزرگی وجود دارد و عمق جواب مشخص نیست، جستجوی تعمیق تکراری یک روش جستجوی ناآگاهانه ارجح است.

```
function ITERATIVE- DEEPENING- SEARCH (problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← ۰ to ∞ do
        result ← Depth –Limited –Search (problem, depth)
        if result ≠cutoff then return result
```

شکل ۳-۱۴ جستجوی عمیق‌شونده تکراری که به طور مکرر محدودیت عمق جستجو با عمق محدود را زیاد می‌کند. الگوریتم زمانی خاتمه می‌یابد که جوابی بیابد یا اگر عمق جستجوی با عمق محدود "شکست" را برگرداند که به معنی عدم وجود جواب می‌باشد.





شکل ۳-۱۵ چهار تکرار از جستجوی عمیق‌شونده تکراری بر روی درخت دودویی

جستجوی عمیق‌شونده تکراری مشابه جستجوی سطح-اول است از این نظر که کل گره‌های یک لایه را قبل از اینکه به لایه‌ی بعدی برود چک می‌کند. به نظر می‌رسد ایجاد یک الگوریتم تکراری مشابه جستجوی یکنواخت که ویژگی بهینه بودن را به ارث برده و همچنین نیاز حافظه‌اش کم باقی بماند، کار با ارزشی باشد. ایده این الگوریتم استفاده از محدودیت هزینه مسیر افزایشی به جای افزایش محدودیت عمق است. الگوریتم حاصل جستجوی طولانی‌شونده تکراری نامیده می‌شود که در تمرین ۳-۱۱ بررسی می‌شود. ولی متأسفانه نتایج نشان می‌دهند که افزایش طول تکراری سربار زیادی را نسبت به الگوریتم جستجو با هزینه‌ی یکنواخت تحمیل می‌کند.

۳-۴-۶- جستجوی دوطرفه[۱]

ایده جستجوی دوطرفه این است که به طور همزمان، از حالت اولیه رو به جلو و از حالت هدف رو به عقب حرکت کنیم. و هنگامی که دو جستجو در وسط به هم رسیدند متوقف شویم (شکل ۳-۱۶). انگیزه ایجاد این الگوریتم این است که $b^{d/2} + b^{d/2}$ بسیار کوچکتر از b^d می‌باشد و یا (در شکل) مساحت دو دایره کوچک کمتر از مساحت یک دایره بزرگ به مرکزیت گره آغازین و ادامه یافته تا هدف می‌باشد.

جستجوی دوطرفه، قبل از بسط هر گره در یکی از جستجوها بررسی می‌کند که آیا گره مورد نظر در حاشیه جستجوی دوم وجود دارد یا خیر، که در صورت وجود، راه‌حل پیدا شده است.

برای مثال، اگر راه‌حل مسئله در عمق $d = ۶$ وجود داشته باشد، و در دو جهت جستجوی سطح-اول روی یک گره در لحظه اجرا شود، آنگاه در بدترین حالت، دو جستجو زمانی به هم برخورد می‌کنند که تمام گره‌ها بعلاوه یک گره در عمق ۳ را بسط داده باشند. برای $b = ۱۰$، در مجموع ۲۲۲۰۰ گره تولید می‌شود (در مقایسه با ۱۱۱۱۱۱۰۰ گره برای جستجوی سطح-اول استاندارد). چک کردن عضویت گره یک درخت در درخت جستجوی دیگر در زمان ثابت و با جدول *hash* قابل انجام است. بنابراین، پیچیدگی زمانی الگوریتم دوطرفه، $O(b^{d/2})$ است. حداقل یکی از درختها باید در حافظه نگهداری شود تا بررسی فوق قابل انجام باشد. بنابراین پیچیدگی حافظه نیز برابر $O(b^{d/2})$ است. این نیاز حافظه بزرگترین ضعف جستجوی دو طرفه است. الگوریتم کامل و بهینه (برای هزینه گام یکنواخت) است اگر هر دو جستجو سطح-اول باشند. ترکیبات دیگر ممکن است کامل بودن، بهینه بودن یا هر دو را از دست بدهند.

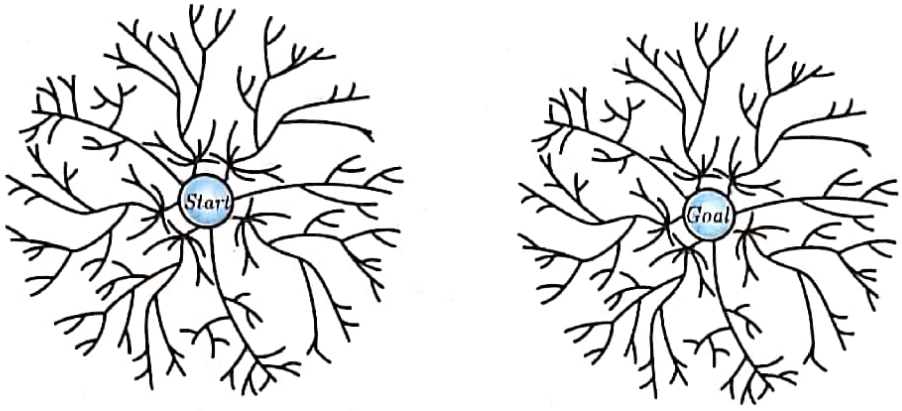
کاهش پیچیدگی زمانی، جستجوی دوطرفه را جذاب می‌کند اما چطور باید از هدف به عقب برگردیم؟ این کار به این سادگی میسر نیست. فرض کنید جد حالت x ، همه حالاتی باشند که حالت x را به عنوان حالت پسین دارند. جستجوی دو طرفه نیازمند محاسبه (به صورت کارا) جد x است. ساده‌ترین حالت وقتی اتفاق می‌افتد که همه کنش‌ها درفضای حالت معکوس‌پذیر باشند. بنابراین جد x برابر است با حالت پسین x حالت‌های دیگر به هوشمندی بیشتری احتیاج دارند.

در نظر داشته باشید که ما در جستجوی به عقب (از هدف) چه چیزی را به عنوان هدف درنظر می‌گیریم؟ برای معمای ۸ و پیدا کردن مسیر در رومانی تنها یک حالت هدف وجود دارد، بنابراین جستجوی به عقب بسیار شبیه جستجوی به جلو است. اگر چندین هدف مشخص وجود داشته باشد (برای مثال دو حالت هدف بدون آلودگی در شکل ۳-۳) آنگاه می‌توانیم یک حالت هدف مجازی بسازیم که اجداد بلافاصله آن، هدف‌های واقعی باشند. در روشی دیگر، می‌توانیم از تولید گره‌های اضافی جلوگیری کنیم. دراین روش مجموعه‌ای از حالت‌های هدف را به عنوان یک حالت درنظر می‌گیریم(و در نتیجه اجداد آن نیز مجموعه‌ای از حالات هستند). سخت‌ترین حالت در جستجوی دوطرفه زمانی اتفاق می‌افتد که تست هدف، توصیفی غیرصریح از هدف را (که احتمالا شامل مجموعه‌ای از هدف‌ها می‌شود) ارائه دهد. برای مثال، همه حالاتی که تست هدف کیش‌ومات را در شطرنج می‌گذرانند. جستجوی به عقب نیاز به ساختن توصیف فشرده‌ای از همه حالاتی که منجر به کیش و مات با

---

[۱]- Bidirectional Search

حرکت m_1 می‌باشند و ...دارد. و این توصیفات باید با حالاتی که توسط جستجوی رو به جلو تولید می‌شوند سنجیده شوند. هیچ راه حل کلی برای انجام این کار به طور بهینه وجود ندارد.



شکل ۳-۱۶ طرحی از یک جستجوی دوطرفه در هنگامی که یک شاخه از گره آغازین با یک شاخه از گره هدف برخورد می‌کند، جواب بدست می‌آید

۳-۴-۷- مقایسه استراتژی‌های جستجو

شکل ۳-۱۷ استراتژی‌های جستجو را بر اساس چهار معیار ارزیابی مقایسه می‌کند.

معیار	دو طرفه (در صورت امکان اعمال)	سطح اول	هزینه یکنواخت	عمیق ـ اول	عمق ـ محدود	عمیق شونده تکراری
کامل بودن	$Yes^{a,d}$	Yes^{a}	$Yes^{a,b}$	No	No	Yes^{a}
زمان	$O(b^{d/2})$	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^{m})$	$O(b^{l})$	$O(b^{d})$
فضا	$O(b^{d/2})$	$O(b^{d+1})$	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
بهینگی	$Yes^{c,d}$	Yes^{c}	Yes	No	No	Yes^{c}

شکل ۳-۱۷ سنجش استراتژی‌های جستجو. b فاکتور انشعاب، d عمق کم‌عمق‌ترین جواب، m ماکزیمم عمق درخت جستجو و l محدودیت عمق می‌باشد. [a] کامل بودن در صورت محدود بودن b، [b] کامل بودن اگر هزینه‌ی هر گام بزرگتر مساوی مقداری مثبت باشد. [c] بهینه بودن در صورت یکسان بودن هزینه‌ی همه‌ی گام‌ها. [d] اگر هر دو جهت از جستجوی سطح- اول استفاده کنند.

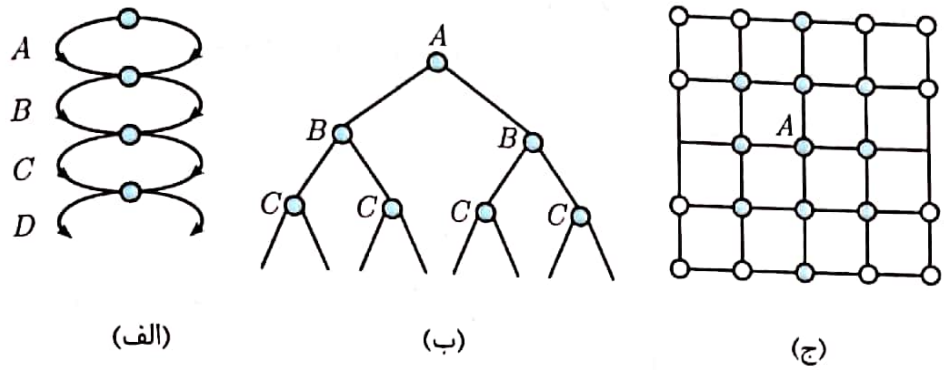
۳-۵- اجتناب از حالات تکراری

تا به حال، همه چیز را در نظر گرفتیم ولی یکی از مهم‌ترین عواقب فرآیند جستجو را در نظر نگرفته‌ایم: امکان اتلاف زمان با بسط گره‌هایی که قبلاً با آنها مواجه شده‌ایم و قبلا در مسیرهای دیگری بسط یافته‌اند. برای بعضی مسایل این امکان هرگز وجود ندارد زیرا هر حالت فقط از یک مسیر قابل دسترسی است. فرموله‌سازی کارای مساله ۸- وزیر(منظور قرار دادن وزیر جدید در خالی‌ترین ستون سمت چپ است) نیز بهمین دلیل کارایی بسیار

زیادی دارد زیرا هر حالت درآن تنها از یک مسیر بدست می‌آید. اگر فرموله سازی مسئله ۸ وزیر را طوری انجام دهیم که وزیرها در هر ستون دلخواه قرار داده شوند ، هر حالت با n وزیر میتواند از !n مسیر مختلف بدست آید.
در بسیاری از مسایل، اجتناب از حالت‌های تکراری امکان‌پذیر نیست. به عنوان مثال، مسایلی که دارای کنشهای معکوس‌پذیرند مثل مسایل پیدا کردن مسیر ویا بلاک‌های پازل متحرک. درخت‌های جستجوی چنین مسایلی بی نهایتند. اما اگر قسمت‌هایی از حالات تکراری را هرس کنیم، می‌توانیم درختی با سایز متناهی بسازیم (تنها قسمتی از درخت را تولید کنیم که گراف فضای حالت را به طور کامل پوشش می‌دهد). با در نظر گرفتن درخت جستجو تا عمق متناهی، می‌توانیم موارد زیادی را پیدا کنیم که اجتناب از حالات تکراری موجب کاهش نمایی هزینه جستجو می‌شوند. در بدترین حالت، فضای حالتی با اندازه‌ی ۱ + d (شکل ۳-۱۸- الف) به درختی با 2^d برگ تبدیل می‌شود (شکل ۳-۱۸-ب).
یک مثال واقعی‌تر مستطیل چهارخانه است که در شکل ۳-۱۸-ج شرح داده شده است. بر روی چهارخانه هر حالت ۴ حالت پسین دارد بنابراین درخت جستجو شامل حالات تکراری است و 4^d برگ دارد. اما تنها حدود $2d^2$ حالت متفاوت در d گام (به ازای هر حالت شده) دارد. برای d = ۲۰ ، تعداد گره‌ها یک تریلیون است در حالیکه حالات مجزا ۸۰۰ تا می‌باشند. حالات تکراری میتوانند یک مسئله قابل حل را غیر قابل حل کنند(البته اگر الگوریتم آنها را کشف نکند). کشف این حالات ازطریق مقایسه حالاتی که قرار است بسط یابند با حالاتی که قبلاً بسط داده شده‌اند امکان‌پذیر است. اگر دو حالت یکسان بودند الگوریتم دو مسیر منتهی به یک حالت را پیدا کرده و می‌تواند یکی از این دو را نادیده بگیرد.
برای الگوریتم جستجوی عمق- اول، تنها گره‌هایی در حافظه هستند که در مسیر ریشه تا گره کنونی قرار دارند. مقایسه این گره‌ها با گره کنونی به الگوریتم اجازه می‌دهد که حلقه‌ها را کشف کرده و بلافاصله حذف کنند.
این کار برای اطمینان از اینکه فضاهای حالت محدود به خاطر وجود حلقه به درختهای جستجوی نامتناهی تبدیل نشوند مناسب است ولی متأسفانه از تولید نمایی مسیرهای غیر حلقه در مسائلی مثل شکل ۳-۱۸ جلوگیری نمی‌کند. تنها راه جلوگیری از این موضوع، نگهداری گره‌های بیشتری در حافظه است. مسئله مصالحه[1] بین زمان و حافظه یک مسئله پایه‌ای است. الگوریتمهایی که تاریخچه‌ی خود را فراموش میکنند ناگزیر به تکرار آن هستند.

اگر یک الگوریتم همه حالاتی را که ملاقات کرده‌است به خاطر بسپارد، مثل این است که مستقیماً برروی گراف فضای حالت جستجو می‌کند. میتوانیم الگوریتم عمومی جستجوی درختی را طوری تغییر دهیم که شامل یک ساختار داده به نام "لیست بسته" بوده و گره‌های بسط یافته را درآن نگهداری کند(گاهی صف گره‌های بسط نیافته را "لیست باز" می‌نامند). اگر گره جاری با یکی از گره‌های لیست بسته یکی باشد، آنگاه آن گره حذف می‌شود(و دیگر بسط داده نمی‌شود). الگوریتم جدید، جستجوی گرافی نامیده می‌شود(شکل ۳-۱۹). در مسائلی با حالات تکراری زیاد، جستجوی گرافی بسیار کاراتر از جستجوی درختی است. زمان و فضای مورد نیاز آن در بدترین حالت، متناسب با اندازه‌ی فضای حالت است و ممکن است بسیار کمتر از $O(b^d)$ باشد.

[1]- Trade-off



(الف) (ب) (ج)

شکل ۳-۱۸ فضاهای حالتی که درخت جستجوی بزرگتر(به صورت نمایی) تولید می‌کنند. الف- یک فضای حالت که دو کنش مختلف از A به B، دو کنش از B به C و غیره دارد. فضای حالت شامل $d + ۱$ حالت است که d بیشینه عمق می‌باشد.ب- جستجوی درخت متناظر با آن که ۲^d شاخه دارد که متناظر با ۲^d مسیر در فضای حالت است. ج- فضای چهارخانه مستطیلی. حالاتی با فاصله دو گام از حالت آغازی (A) به صورت خاکستری نشان داده شده‌اند.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed← an empty set
    fringe ← INSERT(MAKE-NODE (INITIAL-STATE[problem]), fringe
    loop do
        if EMPTY?(fringe) then return failure
        node ← REMOVE-FIRST(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION (node)
        if STATE[node]is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL (EXPAND (node, problem), fringe)
```

شکل ۳-۱۹ الگوریتم عمومی جستجوی گراف. مجموعه بسته می‌تواند با جدول hash پیاده‌سازی شود که کار جستجو برای حالات تکراری را بهینه می‌کند. در این الگوریتم فرض می‌شود که اولین مسیر به حالت S کم‌هزینه‌ترین است (رجوع به متن).

بهینه بودن برای گراف جستجو کمی دشوارتر است. قبلاً گفتیم که وقتی یک حالت تکراری کشف شود الگوریتم دو مسیر منتهی به یک حالت را پیدا کرده‌است. الگوریتم جستجوی گرافی در شکل ۳-۱۹ همواره مسیر جدید کشف شده را حذف می‌کند. واضح است که اگر مسیر جدید کوتاهتر از مسیر قبلی باشد، جستجوی گرافی جواب بهینه را از دست می‌دهد. خوشبختانه می‌توان نشان داد (تمرین ۳-۱۲) با استفاده از الگوریتم جستجو با هزینه یکنواخت و جستجوی سطح- اول با هزینه گام ثابت این اتفاق نمی‌افتد. بنابراین این دو الگوریتم که در جستجوی درختی بهینه هستند در این متد نیز بهینه می‌باشند ولی جستجوی با عمیق‌شونده تکراری از بسط عمق-اول استفاده می‌کند و به سادگی یک مسئله غیربهینه را بدون پیدا کردن مسیر بهینه دنبال می‌کند. بنابراین جستجوی عمیق‌شونده تکراری نیاز به مقایسه مسیر جدید کشف شده با مسیر قدیمی دارد. اگر مسیر جدید بهتر بود باید عمق و هزینه مسیر اجداد آن نود اصلاح شوند. توجه کنید که استفاده از لیست بسته به معنای این است که جستجوی عمق- اول و جستجوی عمیق‌شونده تکراری دیگر به حافظه خطی نیازی ندارند. زیرا جستجوی گرافی همه گره‌ها را در حافظه نگهداری می‌کند. بعضی جستجوها به خاطر محدودیت حافظه قابل اجرا نیستند.

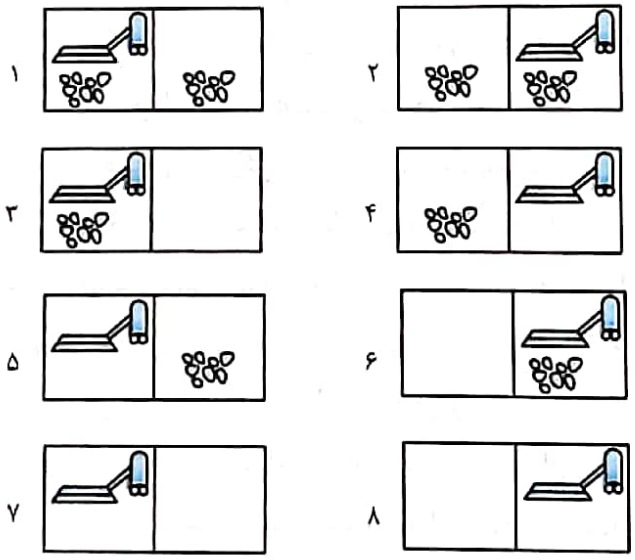
۳-۶- جستجو با اطلاعات ناقص

در بخش‌های قبل فرض کردیم که محیط کاملاً مشاهده‌پذیر و قطعی است و عامل می‌داند کنش‌هایش چه تأثیراتی بر محیط دارند. پس عامل می‌تواند حالت حاصله از اجرای دنباله‌ای از کنش‌ها را محاسبه کند و در نتیجه عامل همواره از حالت جاری خود مطلع است. بنابراین ادراکات عامل پس از اجرای هر کنش اطلاعات جدیدی به او اضافه نمی‌کند. چه اتفاقی می‌افتد اگر دانش کنش‌ها و حالات ناکامل باشد؟ انواع مختلفی از ناکاملی وجود دارد که همه‌ی آن‌ها منجر به ۳ نوع مختلف از مسائل می‌شوند:

۱- مسائل بدون حسگر[۱]: اگر عامل هیچ حسگری نداشته‌باشد آنگاه (تا آنجایی که می‌داند) ممکن است در یکی از چندین حالت اولیه ممکن باشد و هر کنش ممکن است به یکی از حالات پسین مجاز منتهی شود.

۲- مسائل غیرمترقبه[۲] (اقتضایی): اگر محیطی پاره‌مشاهده‌پذیر باشد، یا اگر کنش‌هایش غیرقطعی باشند، ادراکات عامل پس از هر کنش، اطلاعات جدیدی را برای عامل فراهم می‌کنند. هر ادراک ممکن اتفاق غیرمترقبه‌ای را تعریف می‌کند که باید برایش برنامه‌ریزی شده باشد. اگر عدم قطعیت توسط کنش عامل‌های دیگر به وجود بیاید مسئله رقابتی نامیده می‌شود.

۳- مسائل اکتشافی[۳]: اگر حالات و کنش‌های محیط ناشناخته باشند، عامل باید برای کشف آنها تلاش کند. مسائل اکتشافی می‌توانند به عنوان حالت شدید مسائل غیرمترقبه در نظر گرفته شوند.



شکل ۳-۲۰ ۸ حالت ممکن برای جهان جاروبرقی

برای مثال، جهان جاروبرقی را در نظر بگیرید. به یاد داریم که فضای حالت در آن ۸ حالت داشت که در شکل ۳-۲۰ نشان داده شده‌اند. سه کنش وجود دارد(راست، چپ، مکش) و هدف تمیز کردن گرد و غبار است (حالت ۷ و ۸). اگر محیط مشاهده‌پذیر و قطعی و کاملاً شناخته شده باشد مسئله با همه‌ی الگوریتم‌هایی که توصیف شده، قابل حل است. برای مثال اگر حالت آغازی ۵ باشد، آنگاه رشته کنش [راست و مکش] ما را به حالت هدف

۱- Sensorless Problem
۲- Contingency Problem
۳- Exploration Problem

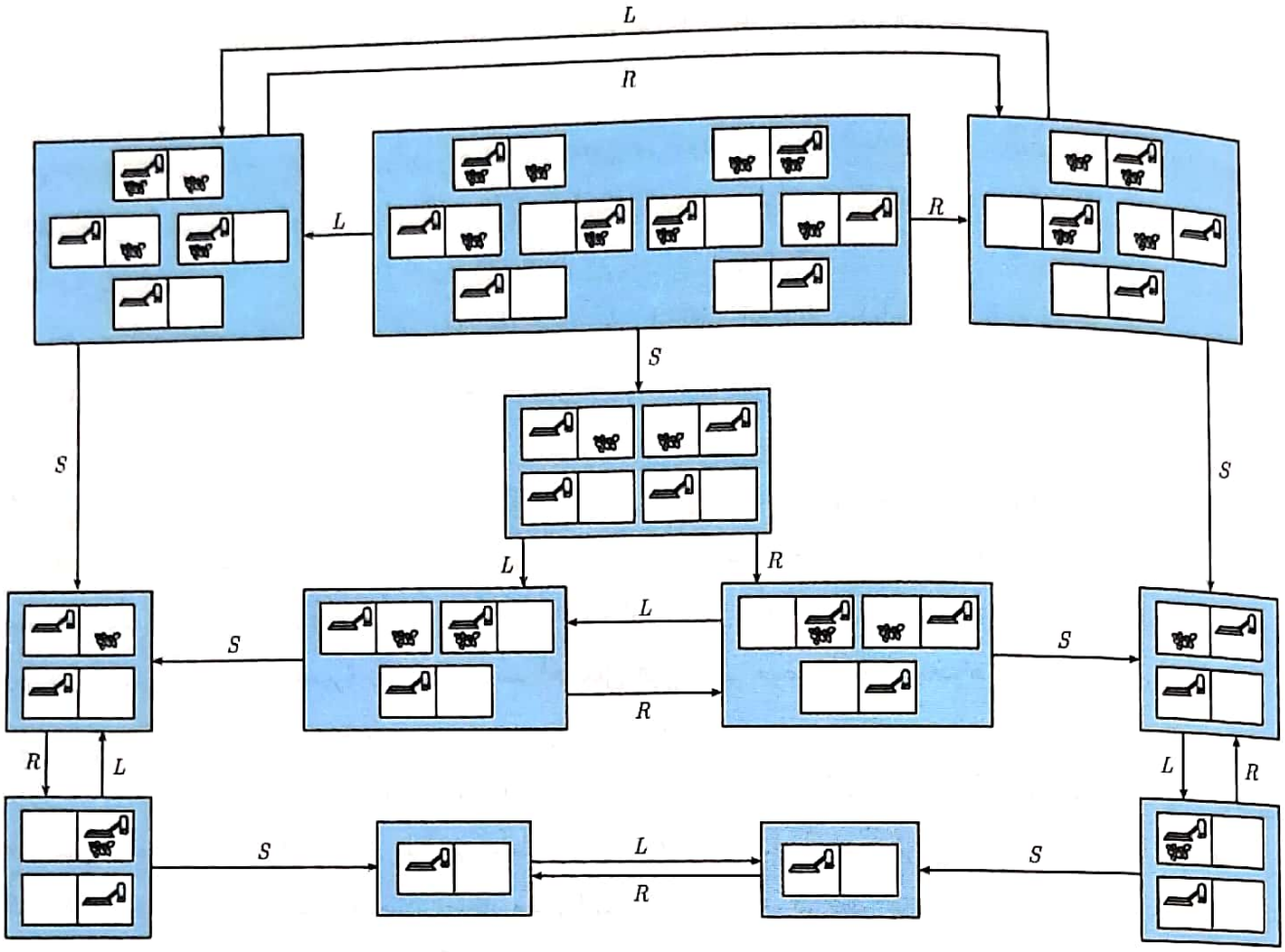
یعنی ۸ می‌رساند. بقیه‌ی این بخش به مسائل بی‌حسگر و غیرمترقبه می‌پردازد. مسائل اکتشافی در فصل ۴ و مسائل رقابتی در فصل ۶ پوشش داده می‌شوند.

۳-۶-۱- مسائل بی‌حسگر

فرض کنید عامل جاروبرقی همه‌ی تأثیرات کنش‌هایش را می‌شناسد ولی هیچ حسگری ندارد. بنابراین تنها می‌داند که حالت آغازی‌اش عضوی از مجموعه‌ی {۱، ۲،...، ۸} است. ممکن است تصور کنید وضعیت عامل ناامید کننده‌است. ولی در واقع این‌طور نیست. زیرا عامل از عملکرد کنش‌هایش مطلع است. برای مثال می‌داند، کنش راست موجب قرار گرفتن عامل در یکی از حالت‌های {۲، ۴، ۶، ۸} و رشته کنش [راست و مکش] همیشه به یکی از حالات {۴، ۸} ختم می‌شوند. سرانجام رشته‌ی [راست، مکش، چپ، مکش] تضمین می‌کند که حالت هدف ۷ مستقل از حالت آغازین به‌دست می‌آید.

درحقیقت عامل محیط را به سمت حالت ۷ سوق می‌دهد (حتی وقتی نمی‌داند از کجا شروع کرده‌است). به‌طور خلاصه زمانی که محیط کاملاً مشاهده‌پذیر نیست، عامل باید پیرامون مجموعه‌ای از حالات به جای یک حالت، استدلال کند. به چنین مجموعه‌ای از حالات حالت عقیده می‌گوییم که نشان دهنده عقیده جاری عامل درباره حالت‌های فیزیکی‌ای است که احتمالا در آن‌ها قرار دارد. در یک محیط کاملاً مشاهده‌پذیر هر حالت عقیده شامل تنها یک حالت فیزیکی است. برای حل مسائل بدون حسگر در فضای حالات عقیده (به جای حالات فیزیکی) جستجو می‌کنیم. حالت ابتدایی یک حالت عقیده است و هر کنش یک حالت عقیده را به حالت عقیده دیگری مرتبط می‌کند. یک مسیر تعدادی از حالات عقیده را به هم وصل می‌کند و جواب، مسیری است که به یک حالت عقیده منتهی می‌شود که همه‌ی اعضایش حالت هدف هستند. شکل ۳-۲۱ فضای حالت-عقیده قابل دسترس را برای جهان جارو برقی قطعی بدون حسگر نشان می‌دهد. تنها ۱۲ حالت عقیده قابل دسترس وجود دارند اما کل فضای حالات عقیده شامل همه‌ی مجموعه حالات فیزیکی یعنی 2^8 = ۲۵۶ حالت عقیده است. به‌طور کلی، اگر فضای حالت فیزیکی S حالت داشته باشد، فضای حالت عقیده 2^S حالت عقیده دارد.

تاکنون بحث ما در مورد مسائل بدون‌حسگر، کنش‌ها را قطعی فرض می‌کند. حال اگر محیط غیرقطعی باشد (یعنی کنش‌ها چند نتیجه ممکن داشته باشند) تحلیل بازهم بدون تغییر باقی می‌ماند. زیرا در غیاب حسگرها عامل هیچ راهی برای ادراک نتیجه اتفاق افتاده ندارد بنابراین نتایج مختلف تنها حالات فیزیکی اضافی‌ای در حالات عقیده پسین هستند. برای مثال فرض کنید محیط از قانون مورفی پیروی می‌کند: کنش مکش گاهی آلودگی روی زمین باقی می‌گذارد اگر محیط قبلاً تمیز بوده باشد. پس اگر مکش به حالت فیزیکی ۴ (شکل ۳-۲۰) اعمال شود، ۲ نتیجه‌ی ۲ و ۴ محتمل هستند. اگر مکش به حالت عقیده اولیه {۱،...،۸} اعمال شود، خروجی برابر با اجتماع نتایج حاصل از اعمال مکش به هر کدام از ۸ عضو حالت آغازین است. بنابراین برای یک مسئله بدون حسگر



شکل ۳-۲۱ بخش قابل دسترس حالات قابل تصور برای جهان جارو برقی قطعی بدون حسگر. هر مربع خاکستری شامل یک حالت قابل تصور است. در هر لحظه عامل در یک حالت قابل تصور است ولی نمی‌داند در کدام حالت فیزیکی است. حالت قابل تصور آغازین (بی اطلاعی کامل) بالاترین مربع می‌باشد. کنش‌ها با فلش‌های عنوان‌دار نشان داده‌شده‌اند. حلقه‌هایی که به خود حالت برمی‌گردند، برای وضوح حذف شده‌اند.

در جهان قانون مورفی کنش مکش حالت عقیده را تغییر نمی‌دهد! در واقع مسئله با این شرایط غیر قابل حل است. (تمرین ۳-۱۶) دلیل این مشکل این است که عامل نمی‌تواند تشخیص دهد خانه کثیف است و بنابراین نمی‌داند که آیا عمل کنش تمیز می‌کند یا کثیفی بیشتری ایجاد می‌کند.

۳-۶-۲- مسائل غیرمترقبه

هنگامی که شرایط برای عامل به‌گونه‌ای است که عامل بعد از عمل کردن در محیط اطلاعات جدیدی در مورد محیط بدست آورد (از طریق حسگرهای خود)، عامل با یک مسئله غیرمترقبه احتمالی روبروست. راه‌حل مسائل احتمالی معمولاً به شکل یک درخت است که هر شاخه‌اش بسته به ادراک (تا آن لحظه) انتخاب می‌شود. برای مثال، فرض کنید عامل جاروبرقی در جهان مورفی دو حسگر برای تشخیص مکان جاری و آلودگی آن داشته باشد ولی حسگری برای تشخیص آلودگی دیگر خانه‌ها نداشته باشد. بنابراین ادراک [چپ و آلوده] بدین معنی‌ست که عامل در یکی از دو حالت {۳،۱} قرار دارد. عامل ممکن است رشته‌ی کنش [مکش و راست و مکش] را فرموله‌سازی کند. مکش اول حالت را به یکی از دو حالت {۷،۵} تغییر می‌دهد و حرکت به راست حالت را به یکی از دو حالت {۸،۶} تغییر می‌دهد. اجرای کنش نهایی مکش در حالت ۶ ما را به حالت ۸

می‌رساند ولی اجرای این کنش در حالت ۸ ما را دوباره به حالت ۶ برمی‌گرداند(توسط قانون مورفی) که به معنی شکست در برنامه است.

با آزمودن فضای حالات عقیده در این مسئله مشخص می‌شود که هیچ رشته‌ی کنش ثابتی برای جواب مسئله وجود ندارد ولی اگر بر رشته‌ی کنش ثابت تأکید نداشته‌باشیم راه‌حلی به شکل زیر وجود دارد:

[مکش، راست، اگر [راست و آلوده] پس مکش]

این کار امکان انتخاب کنش‌ها براساس احتمالات (منظور اتفاقات احتمالی طول اجرا) را به فضای راه‌حل‌ها اضافه می‌کند. بسیاری از مسائل در جهان واقعی و فیزیکی احتمالی هستند زیرا پیش‌بینی دقیق اتفاقات امری غیرا ممکن است.

گاهی مسائل احتمالاتی شامل راه‌حل‌های کاملا ترتیبی می‌شوند. برای مثال یک جهان قانون مورفی کاملا مشاهده‌پذیر را در نظر بگیرید. احتمالات زمانی رخ می‌دهد که عمل مکش در یک محیط تمیز اتفاق بیفتد(زیرا ممکن است آلودگی بر جای بماند و یا ممکن است اصلا این اتفاق رخ ندهد). تا زمانی که عامل این کنش (منظور مکش در محیط تمیز) را انجام ندهد احتمال فوق‌الذکر موضوع بحث نخواهد بود و یک راه‌حل کاملاً ترتیبی از حالت آغازین وجود خواهد داشت (تمرین ۳-۱۶).

الگوریتم مسائل احتمالی بسیار پیچیده‌تر از مسائل جستجوی استاندارد این فصل هستند. مسائل احتمالی همچنین متمایل به نوع دیگری از طراحی عامل هستند که عامل قبل از پیدا کردن نقشه‌ای که جواب را تضمین کند، شروع به عمل در محیط میکند. اینکار مفید است زیرا به جای در نظر گرفتن هر احتمالی که ممکن است در طول اجرا رخ دهد، بهتر است که کنش در محیط شروع شود تا ببینیم چه احتمالاتی واقعاً پیش می‌آیند. بنابراین عامل شروع به حل مسئله میکند و اطلاعات اضافی را نیز به حساب می‌آورد. این نوع هم‌زمانی اجرا و جستجو در مسائل جستجو (فصل ۴) و بازی‌ها (فصل ۶) نیز مفید است.

۳-۷- خلاصه

دراین فصل روشهایی را معرفی کردیم که با توجه به آنها، یک عامل می‌تواند حرکات بعدی خود را در محیطی قطعی، مشاهده‌پذیر، ایستا و کاملاً شناخته شده، انتخاب کند. درچنین مواردی عامل دنباله‌ای از کنشها را برای رسیدن به هدف می‌سازد که به این فرایند جستجو گفته می‌شود.

- قبل از اینکه عامل برای یافتن راه حل شروع به جستجو کند، باید هدف را فرموله‌بندی و سپس، از این هدف برای فرموله‌بندی مسأله استفاده کند.
- یک مسأله از ۴ قسمت تشکیل شده است: حالت اولیه، تعدادی کنش، یک تابع تست هدف نهایی و یک تابع برای محاسبه هزینه مسیر. محیط مسأله نیز توسط فضای حالت قابل نمایش است. راه‌حل، مسیری از حالت اولیه تا حالت نهایی در فضای حالت است.
- الگوریتم عمومی جستجوی درخت برای حل هر مسأله‌ای قابل استفاده است.
- الگوریتم‌های جستجو بر مبنای کامل بودن، بهینه بودن، پیچیدگی زمانی و پیچیدگی فضایی (منظور فضای حافظه است) مقایسه می‌شوند. پیچیدگی به فاکتور انشعاب b در فضای حالت، و عمق کم عمق ترین راه‌حل d وابسته است.

- جستجوی سطح-اول (BFS) ابتدا کم‌عمق‌ترین گره را بسط می‌دهد. این جستجو کامل است و در صورتی که کنش‌ها دارای هزینه واحد باشند، بهینه خواهد بود. پیچیدگی فضایی(فضای حافظه) و زمانی آن $O(b^d)$ است. این پیچیدگی فضایی، آنرا در اکثر موارد ناکارآمد و غیرعملی می‌کند. جستجوی هزینه یکنواخت ابتدا کم هزینه‌ترین گره (کمترین $g(n)$) را بسط می‌دهد. این الگوریتم کامل است و بر خلاف جستجوی سطح-اول حتی در صورتی که کنش‌ها هزینه‌های متفاوت داشته باشند، بهینه است. پیچیدگی زمانی و فضایی آن هم همانند جستجوی سطح- اول است.

- جستجوی عمق- اول (DFS) ابتدا عمیق‌ترین گره در درخت جستجو را بسط می‌دهد. نه کامل است و نه بهینه و پیچیدگی زمانی آن $O(b^m)$ و پیچیدگی فضائی آن $O(bm)$ است. m حداکثر عمق درخت است.

- جستجوی عمق- محدود حدی روی عمقی که الگوریتم جستجوی عمق-اول می‌تواند طی کند قرار می‌دهد.

- جستجوی عمیق‌شونده تکراری (IDS) مرتباً الگوریتم جستجوی با عمق محدود را فراخوانی می‌کند تا اینکه به یک حالت نهایی برسد. این الگوریتم کامل و بهینه است و پیچیدگی زمانی آن $O(b^d)$ و پیچیدگی فضائی آن $O(bd)$ است.

- جستجوی دو طرفه می‌تواند تا حد بسیار زیادی پیچیدگی زمانی را کاهش دهد، اما باید توجه داشت که همیشه قابلیت اجرا ندارد. نیازمندی‌های حافظه این روش ممکن است کاملاً آن را از لحاظ عملی ناممکن کند.

- هنگامی که فضای حالت به جای درخت، گراف باشد بررسی حالات تکراری میتواند مفید باشد. الگوریتم جستجوی درختی حالات تکراری را حذف می‌کند.

- وقتی که محیط پاره‌مشاهده‌پذیر باشد، عامل می‌تواند الگوریتم جستجو را بر روی حالت عقیده یا مجموعه‌ی حالاتی که عامل ممکن است در آن باشد اعمال کند. در بعضی موارد ممکن است یک رشته‌ی پاسخ وجود داشته‌باشد ولی در سایر موارد جواب به صورت احتمالی برای شرایط غیرقابل پیش‌بینی وجود دارد.

۳-۸- تمرین‌ها

۳-۱ اصطلاحات زیر را تعریف کنید: حالت، فضای حالت، درخت جستجو، گره جستجو، هدف، کنش، تابع افزایشی و فاکتور انشعاب.

۳-۲ توضیح دهید چرا باید ابتدا هدف نهایی فرموله‌بندی شود بعد خود مسأله.

۳-۳ فرض کنید تابع LEGAL-ACTION (s) نشان‌دهنده مجموعه‌ای از کنشهاست که در حالت s قانونی هستند و تابع RESULT(a, s) نشان‌دهنده حالتی است که در نتیجه اعمال کنش a بر روی حالت s بدست می‌آید. تابع حالت پسین SUCCESSOR-FN را در قالب این دو تابع تعریف کنید و برعکس.

۳-۴ نشان دهید که حالت‌های ۸-پازل به دو مجموعه منفصل تقسیم می‌شوند به گونه‌ای که هیچ حالتی در یک مجموعه با هیچ تعداد حرکتی قابل تبدیل به حالتی در مجموعه دیگر نیست. رویه‌ای را طراحی کنید که تشخیص دهد هر حالت به کدام دسته تعلق دارد و توضیح دهید که چرا اینکار برای تولید حالات تصادفی مناسب است؟

۵-۳ مسئله‌ی n وزیر را در حالت فرموله‌سازی افزایشی "کارا" در نظر بگیرید. توضیح دهید که چرا اندازه‌ی فضای حالت حداقل $(n!)^{1/3}$ است و بزرگترین n ای که اکتشاف برای آن امکان‌پذیر است را تخمین بزنید.(راهنمایی: با درنظر گرفتن بیشینه تعداد خانه‌هایی(در هرستون دلخواه) که وزیر می‌تواند به آن‌ها حمله کند، حد پایینی برای فاکتور انشعاب درنظر بگیرید).

۶-۳ آیا یک فضای حالت متناهی همیشه به درخت جستجوی متناهی منجر می‌شود؟ در مورد فضای حالت متناهی درختی چه‌طور؟ در مورد فضای حالاتی که به درخت جستجوی متناهی منجر می‌شوند دقیقتر بحث کنید؟

۷-۳ فضای حالت، تست هدف، تابع پسین و تابع هزینه را برای هریک از موارد زیر تعیین کنید. همچنین فرموله‌سازی قابل پیاده‌سازی ارائه کنید.

الف) نقشه‌ای با چهار رنگ طوری رنگ شود که هیچ دو نقطه‌ی مجاوری یکرنگ نباشند.

ب) میمونی با قد ۳ فوت در یک اتاق است. مقداری موز در ارتفاع ۸ متری از سقف آویزان شده است و میمون دوست دارد که موز را به‌دست آورد. اتاق شامل ۲ جعبه به طول ۳ فوت قابل جابجایی و بالا رفتن است.

ج) برنامه‌ای دارید که خروجی‌اش "رکورد ورودی غیرقانونی" است. این پیغام زمانی ظاهر می‌شود که نوع خاصی از فایل به آن داده شود. میدانیم که فرایند هر رکورد مستقل از دیگری است. وظیفه شما کشف رکوردهای غیر قانونی است.

د) ۳ ظرف به اندازه‌های ۱۲، ۸ و ۳ گالون و یک شیر آب دارید. ظرف‌ها میتوانند پر یا خالی شوند و می‌توانند در ظرف دیگری خالی شوند. وظیفه شما پیمانه کردن دقیقا یک گالون است.

۸-۳ فضای حالتی را در نظر بگیرید که حالت شروع عدد یک و تابع پسین برای هر حالت n دو حالت با شماره $2n$ و $2n + 1$ برگرداند.

الف) بخشی از فضای حالت را برای حالات ۱ تا ۱۵ بکشید.

ب) فرض کنید هدف حالت ۱۱ است. ترتیب گره‌های ملاقات شده برای الگوریتم سطح-اول، جستجوی عمق-محدود با محدودیت ۳ و جستجوی عمیق‌شونده تکراری را بنویسید.

ج) آیا جستجوی دو طرفه برای این مثال مناسب است؟ جزئیات کارکرد را توضیح دهید.

د) فاکتور انشعاب در هر جهت جستجوی دوطرفه چیست؟

ه) آیا پاسخ ج فرموله‌سازی‌ای برای حل مسئله برای رسیدن از حالت شروع به حالت هدف بدون جستجو را معرفی می‌کند؟

۹-۳ مسئله کشیش و آدمخوارها به شکل زیر است: سه کشیش و سه آدم خوار در یک طرف رودخانه هستند با یک قایق که یک یا دو نفر را حمل می‌کند. راهی پیدا کنید که همه به سمت دیگر بروند بدون اینکه کشیشها را با تعداد بیشتری آدمخوار تنها بگذارید. این مسئله معروف در هوش موضوع اولین مقاله‌ای بود که فرموله‌سازی را از دید تحلیلی بررسی کرد.

الف) مسئله را به دقت فرموله‌سازی کنید، تنها نکاتی را اشاره کنید که برای حل معتبر مسئله لازم است.

کل فضای حالت مسئله را بکشید.

ب) مسئله را به طور بهینه با یک الگوریتم جستجوی مناسب پیاده‌سازی و حل کنید. آیا چک کردن حالات تکراری فکر خوبی‌ست؟

ج) فکر می‌کنید چرا مردم برای حل این مسئله با مشکل مواجه‌اند؟ با اینکه فضای حالت بسیار ساده است.

۳-۱۰ دو نسخه تابع پسین برای معمای ۸-پازل پیاده‌سازی کنید بگونه‌ای که: همه‌ی حالت‌های پسین را با کپی کردن و ویرایش ساختار داده ۸-پازل تولید کند. ودیگری به گونه‌ای که در هربار فراخوانی با استفاده از تغییر مستقیم حالت والد، یک حالت پسین جدید تولید کند. نسخه عمیق‌شونده تکراری جستجوی عمق-اول را با استفاده از این توابع پیاده‌سازی و کارایی آن‌ها را مقایسه کنید.

۳-۱۱ اشاره کردیم که جستجوی طولانی‌شونده مکرر مشابه الگوریتم با هزینه‌ی یکنواخت تکرارشونده است. ایده، افزایش محدودیت برروی هزینه‌ی مسیر است. اگر گره‌ای تولید شود که هزینه‌اش از محدودیت بیشتر باشد، حذف می‌شود. برای هر تکرار جدید محدودیت به کمترین هزینه‌ی گره حذف شده تغییر می-کند.

الف) نشان دهید که این الگوریتم برای همه‌ی هزینه‌ها بهینه است.

ب) درخت یکنواخت با فاکتور انشعاب b عمق جواب d و هزینه‌ی گام واحد را در نظر بگیرید. چند تعداد تکرار لازم است؟

ج) هزینه‌ی گام را در بازه‌ی [۱ و ۰] در نظر بگیرید. چند تکرار در بدترین حالت لازم است؟

د) الگوریتمی پیاده‌سازی و بر نمونه‌هایی از معمای ۸-پازل و فروشنده دوره‌گرد اعمال کنید. کارایی الگوریتم را با جستجوی با هزینه‌ی یکنواخت مقایسه کنید و در مورد نتایج آن نظر دهید.

۳-۱۲ ثابت کنید که جستجو با هزینه‌ی یکنواخت و جستجوی سطح-اول با هزینه گام ثابت برروی جستجوی گرافی بهینه هستند. فضای حالتی با هزینه‌ی گام متغیر نشان دهید که درآن جستجوی گرافی با استفاده از الگوریتم عمیق‌شونده تکراری راه‌حل غیربهینه برمی‌گرداند.

۳-۱۳ فضای حالتی را توصیف کنید که درآن جستجوی عمیق‌شونده تکراری بسیار بدتر از جستجوی عمق-اول عمل می‌کند.

۳-۱۴ برنامه‌ای بنویسید که دو ورودی URL گرفته و مسیری از لینک‌ها را بین آنها پیدا کند. چه استراتژی جستجویی مناسب است؟ آیا جستجوی دوطرفه فکر خوبی است؟ آیا یک موتور جستجو می‌تواند برای استفاده به عنوان تابع پسین پیاده‌سازی شود؟

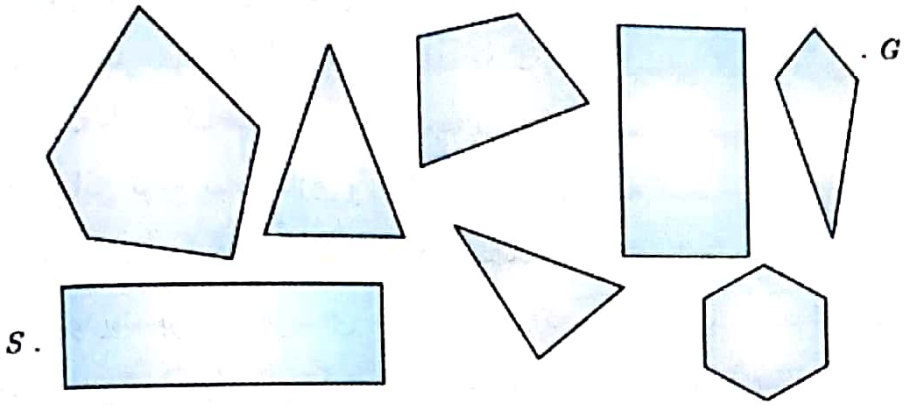
۳-۱۵ مسئله یافتن کوتاهترین مسیر بین دو نقطه را برروی یک صفحه که دارای موانع چندضلعی محدب است درنظر بگیرید(شکل ۳-۲۲ را در نظر بگیرید). این مسئله، شکل ایده‌آل مسئله‌ای است که روبات برای یافتن مسیرش در یک محیط شلوغ استفاده می‌کند.

الف) فرض کنید فضای حالت شامل همه موقعیتهای (x, y) در صفحه باشد. چند حالت وجود دارد؟ چند مسیر به هدف وجود دارد؟

ب) به اختصار توضیح دهید که چرا کوتاهترین مسیر از یک رأس چند ضلعی به دیگری باید شامل مسیر مستقیم باشد که رأسهای چندضلعی را به هم وصل می‌کند؟ حال یک فضای حالت خوب تعریف کنید. اندازه‌ی این فضا چیست؟

ج) تابع ضروری برای پیاده‌سازی الگوریتم جستجو شامل تابع پسین که رأس را به عنوان ورودی گرفته و مجموعه‌ای از رئوس قابل دسترس را برگرداند بنویسید.

د) الگویتمهای این فصل را برای حل مسئله اعمال کرده و در مورد کارایی‌شان نظر بدهید.



شکل ۳- ۲۲ تصویری از موانع چند ضلعی

۳-۱۶ جهانی با ویژگی‌های زیر درنظر بگیرید: فاقد حسگر، جهان جاروبرقی دو- خانه‌ای تحت قانون مورفی. حال، فضای حالت عقیده‌ای که از حالت عقیده آغازین {۱و۲و۳و۴و۵و۶و۷و۸} قابل دسترس هست را بکشید و توضیح دهید چرا مسئله غیرقابل‌حل است. همچنین نشان دهید که اگر جهان کاملا مشاهده‌پذیر باشد آنگاه به ازای هر حالت اولیه دلخواه، دنباله راه‌حل متناظر با آن وجود خواهد داشت.

۳-۱۷ با این فرض که حالت اولیه کاملاً معلوم است، مسأله دنیای خلا که درتمرین‌های فصل ۲ بررسی شد می تواند به عنوان یک مسأله جستجو در نظر گرفته شود.

الف) حالت اولیه، عملگرها، تابع تست هدف نهایی و تابع هزینه مسیر راتعریف کنید.

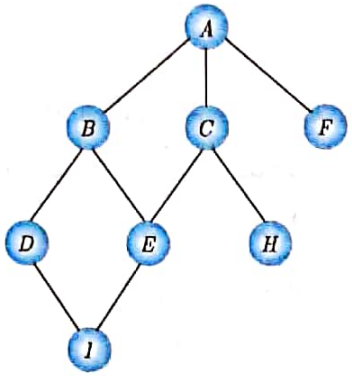
ب) کدام یک از الگوریتم‌هایی که در این فصل تعریف شده‌اند می‌توانند برای این مسأله مناسب باشند؟

پ) یکی از این الگوریتم‌ها را به کار برید و یک دنباله بهینه از اعمال برای حالت ۳×۳ که در آن دو آشغال درخانه شروع و مرکزی وجود دارد پیدا کنید.

ت) یک عامل جستجو برای دنیای خلاء بسازید و کارایی آنرا درمجموعه‌ای از دنیاهای ۳×۳ که احتمال وجود آشغال در هر مربع، ۰/۲ است محاسبه کنید. هزینه جستجو و هزینه مسیر را در محاسبه کارآئی دخالت دهید.

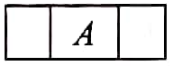
سوالات طبقه‌بندی‌شده فصل سوم

۱- اگر در گراف زیر جستجو در عمق *(Depth First Search)* را از راس *C* شروع کنیم، کدام گره‌ها به ترتیب از چپ به راست رویت *(Visit)* می‌شوند؟ (فرض کنید فرزندان یک گره براساس ترتیب حروف الفبا انتخاب شوند). (کامپیوتر ۸۱)



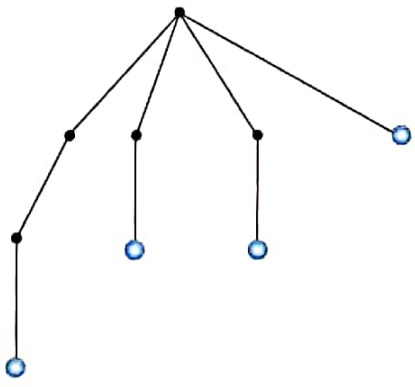
۱) A B C D E F H I
۲) C A B D I E F H
۳) C A E H B F I D
۴) C A B D E H I F

۲- فرض کنید *A* یک جاروبرقی اتوماتیک است. محیط این جاروبرقی، مطابق شکل زیر، از سه خانه کنار هم تشکیل شده است. این جاروبرقی می‌تواند از هریک از این خانه‌ها با انجام یک حرکت به خانه مجاور نقل مکان نماید و زباله‌های موجود در آن خانه را (درصورت وجود) جمع‌آوری کند. باتوجه به این که این جاروبرقی برای جمع‌آوری هر زباله باید در همان خانه‌ای که زباله وجود دارد، قرار بگیرد. فضای حالت این مسئله دارای چند وضعیت منحصر به فرد است؟ (کامپیوتر ۸۳)



۱) ۲۴ ۲) ۹ ۳) ۶۴ ۴) ۸۱

۳- درحین انجام یک روش جستجو، درخت جستجوی حاصل به شکل مقابل رشد یافته است. راس‌هایی که نامزد بسط داده شدن هستند به رنگ سیاه مشخص شده‌اند. این جستجو چه روشی می‌تواند باشد؟ (کامپیوتر ۸۵)



۱) عمق نخست (Depth First)
۲) عرض نخست (Breath First)
۳) جستجوی هزینه یکنواخت (Uniform Cost)
۴) تعمیق تکراری (Iterative Deepening)

۴- ضریب انشعاب یک درخت جستجو ۳ می‌باشد. حل مسئله در آخرین راسی که در عمق ۲ جستجو می‌شود وجود دارد. چه تعداد راس باید بسط داده شوند تا این راس بازدید شود در صورتی که از جستجوی عرض نخست *(Breath First)* استفاده شود؟ (فرض بر این است که حل مسئله بودن یک گره در زمان باز کردن فرزندان آن گره بررسی می‌گردد.) (کامپیوتر ۸۷)

۱) ۱۳ ۲) ۲۷
۳) ۳۲ ۴) ۳۷

۵- کدامیک از عبارت‌های زیر صحیح‌تر است؟ (فناوری اطلاعات ۸۷)

۱) فرموله کردن مسئله همواره باید قبل از فرموله کردن هدف انجام گیرد.

۲) فرموله کردن هدف همواره باید قبل از فرموله کردن مسئله انجام گیرد.

۳) می‌توان فرموله کردن هدف را به اختیار قبل و بعد از فرموله کردن مسئله انجام داد.

۴) فرموله کردن هدف و مسئله اوامر اختیاری هستند، مهم اعمال الگوریتم‌های جستجو است و فرموله کردن این دو همیشه لازم نیست.

۶- روی یک توری $n * n$ که هر خانه به چهار همسایه خود متصل است، خانه میانی را نقطه شروع جستجو و نقطه‌ی $(۰,۰)$ درنظر می‌گیریم. گره هدف در موقعیت (X,Y) است. در این گراف الگوریتم جستجوی A بدون تست تکراری بودن حالات حداکثر $\left(\left(۴^{X+Y+۱}-۱\right)/۳\right)-۱$ گره و الگوریتم جستجوی B با تست تکراری بودن حالات، حداکثر $۴(X+Y)(X+Y+۱)-۱$ گره را قبل از یافتن جواب بسط می‌دهند. کدام یک از گزینه‌های زیر در مورد این دو الگوریتم صحیح است؟ (کامپیوتر ۹۰)

۱) A الگوریتم اول عمق (Depth first) و B الگوریتم اول پهنا (Breadth first) است.

۲) A و B هر دو الگوریتم اول عمق (Depth first) هستند.

۳) A الگوریتم اول پهنا (Breadth first) و B الگوریتم اول عمق (Depth first) است.

۴) A و B هر دو الگوریتم اول پهنا (Breadth first) هستند.

۷- فرض کنید برای مسئله‌ای با جستجوی اول پهنا $(breadth-first)$ و تست هدف در لحظه‌ی تولید، نیاز به بسط دادن $(expand)$ ۳۲ گره باشد. اگرفاکتورانشعاب $(branching-factor)$ درخت جستجو ثابت باشد و عمق درخت برابر ۵ و عمق هدف $(goal)$ برابر ۴ باشد، کدام یک از گزینه‌ها مقدار فاکتور انشعاب (b) را نشان می‌دهد؟ (فرض کنید ریشه‌ی درخت در عمق صفر (۰) واقع شده است.) (کامپیوتر ۹۱)

۱) $b=۲$ ۲) $b>۵$ ۳) $۲<b<۳$ ۴) $۳\leq b\leq ۵$

پاسخنامه تشریحی فصل سوم

۱) گزینه ۲ درست است.

الگوریتم به صورت گام به گام اینگونه عمل خواهد کرد:

سطح یک: ابتدا گره C (گره ریشه) رویت و فرزندان آن بسط داده خواهند شد.

سطح دو: گزینه‌ها برای رویت شدن: A، E، H. که با توجه به ترتیب حروف الفبا، A انتخاب می‌شود.

سطح سه: گزینه‌ها برای رویت شدن: F،B. دقت کنید از آنجا که C یک بار رویت شده، دیگر انتخاب نمی‌شود. بنابراین گره B انتخاب می‌شود.

سطح چهار: گزینه‌ها D، E. گره D انتخاب می‌شود.

سطح پنج: گزینه I. انتخاب می‌شود. (پاسخ درست گزینه ۲ خواهد بود).

الگوریتم به همین ترتیب ادامه پیدا کرده و به ترتیب گره‌های E، F، H بسط داده می‌شوند.

۲) گزینه ۱ درست است.

برای پیدا کردن فضای حالت یک مسئله باید تمامی حالات ممکن برای آن مسئله را در نظر بگیریم. بنابراین:

هرکدام از خانه‌ها ممکن است تمیز یا کثیف باشند: تعداد حالات = $۲^۳ = ۸$

جاروبرقی A در هر لحظه می‌تواند در یکی از خانه‌ها قرار بگیرد. بنابراین تعداد حالات = ۳

تعداد کل حالات $= ۸ \times ۳ = ۲۴$

۳) گزینه ۳ درست است.

زیرا در جستجوی هزینه‌های یکنواخت، عمق گره‌ها مورد نظر نیست، بلکه هزینه گره‌ها در آن مورد توجه قرار می‌گیرد. بنابراین این گزینه قابل قبول خواهد بود.

گزینه ۱ نادرست است. زیرا در جستجوی عمق اول تنها یک گره نامزد برای بسط دادن می‌شود، که آن عمیق‌ترین گره در درخت جستجو است. بنابراین، با توجه به اینکه در شکل، چهار گره نامزد برای بسط دادن هستند، این گزینه رد خواهد شد.

گزینه۲ نادرست است. با توجه به شکل می‌توان دریافت که نودهای بسط داده شده در یک سطح(عمق) قرار ندارند، بنابراین جستجوی مورد نظر نمی‌تواند از نوع عرض نخست باشد.

گزینه ۴ نادرست است. با توجه به اینکه در جستجوی تعمیق تکراری، در هر بار تکرار عمق یک واحد افزایش می‌یابد. بنابراین با توجه به اختلاف عمق گره‌ها در شکل این گزینه نیز رد می‌شود.

۴) گزینه ۴ درست است.

باتوجه به اینکه تست هدف بودن یک گره، زمان بسط گره انجام می‌شود بنابراین برای پیدا کردن هدف در سطح یک، ۱ گره (ریشه) در سطح دو، ۳ گره، و در سطح سوم ۹ گره بسط می‌یابند. اما در این سطح هنوز هدف را پیدا نکرده ایم. بنابراین ۲۴ گره دیگر را نیز بسط می‌دهیم تا بالاخره به گره هدف برسیم. در این لحظه، دیگر گره‌های فرزند گره هدف را بسط نمی‌دهیم زیرا هدف را پیدا کرده ایم.

بنابراین تعداد گره‌های بسط داده شده برابر است با: ((۳-۲۷)+۹+۳+۱)

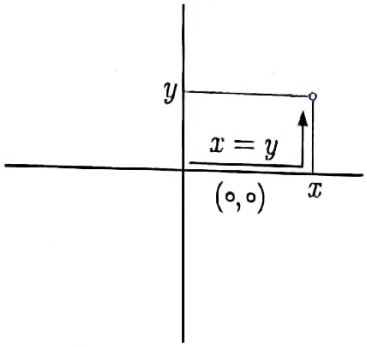
۵) گزینه ۲ درست است.

اولین قدم در حل مسئله فرموله‌سازی هدف براساس موقعیت کنونی و معیار کارایی عامل است. بعد از فرموله‌سازی هدف، عامل می‌تواند در مورد انتخاب فاکتورهای دیگری که در مطلوبیت رسیدن به هدف تاثیر گذارند، تصمیم‌گیری کند.

درحقیقت، قبل از اینکه عامل برای یافتن راه حل شروع به جستجو کند، باید هدف را فرموله‌بندی و سپس، از این هدف برای فرموله‌بندی مسأله استفاده کند.

۶) گزینه ۴ درست است.

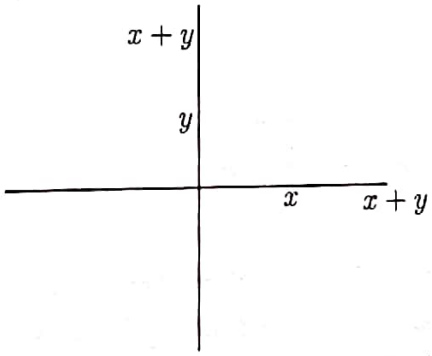
فاصله گره هدف تا گره شروع $X+Y$ یال است. مسأله‌ای که فاکتور انشعاب ۴ باشد و به صورت درختی جستجو شود، اگر با BFS جستجو شود، تعداد گره‌های بسط داده شده برابر است با:



$$h = ۱ + ۴ + ۴^۲ + \cdots + ۴^{X+Y} = \frac{۴^{X+Y+۱} - ۱}{۴ - ۱}$$

بدون احتساب گره نهایی، دقیقاً برابر با تعداد مطرح شده در سؤال است، پس جستجوی A حتماً BFS است.

اگر همین جستجو (BFS) را روی گراف در نظر بگیرید. آنگاه فضای جستجو به شکل زیر است:



مساحت لوزی عملاً تعداد گره‌های بسط داده شده را نشان می‌دهد که برابر است با:

$$۲ \times (X+Y) \cdot (X+Y+۱)$$

بدون احتساب گره نهایی دقیقاً برابر با تعداد مطرح شده در سوال خواهد شد. پس B هم BFS است.

۷) گزینه ۴ درست است.

بایستی مقدار b طوری باشد که اگر عمق یک و دو و سه را بسط بدهیم به هدف نرسیم و حتما در عمق چهار به هدف برسیم. سوال یک نکته دقیق دارد و آن مفهوم بسط دادن است. در واقع سوال گفته است که ۳۲ گره بسط میدهیم و نه تولید میکنیم. یعنی اگر ۳۲ گره بسط دهیم به ازای هر گره b فرزند تولید میکنیم. بنابراین در واقع سوال این است که $۳۲b$ گره تولید شده است.

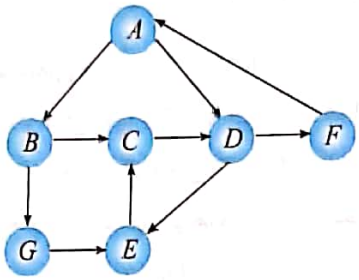
در اینصورت بایستی گره هدف در جایی باشد که در عمق چهار باشد یعنی حتما در سه سطح اول $۳۲b$ گره نتوانیم تولید کنیم.

و حتما پس از بسط سطح چهارم نیز نباشد. یعنی $b + b^۲ + b^۳ < ۳۲b$

مقادیر مجاز b فقط می‌تواند ۳و ۴و ۵ باشد. $b + b^۲ + b^۳ + b^۴ > ۳۲b$

تست‌های تألیفی

۱- اگر در گراف جستجو در عمق (dfs) از راس B شروع کنیم، کدام گره ها به ترتیب از چپ به راست رویت می شوند؟ (فرض شود ترتیب فرزندان یک گره براساس ترتیب حروف الفباست).

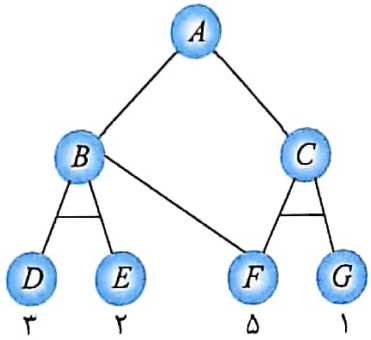


۱) ABCDEFG
۲) BCDEFAG
۳) BCDEAFG
۴) GFEDCBA

۲- حداکثر تعداد گره های فضای حالت 8-puzzle چقدر است؟

۱) !۸گره ۲) ۸۹ ۳) !۹ گره ۴) ۲ / !۹ گره

۳- اگر درخت AND-OR در شکل روبرو بیانگر پیشنهاد انجام یک فعالیت باشد، کدام فعالیت‌ها جهت برآورده شدن A کافی است؟ (هزینه هر یال یک واحد است)، و هزینه هر فعالیت نهایی در زیر آن مشخص است.



۱) F
۲) F,G با هم
۳) D,E با هم
۴) گزینه الف و ج

۴- فرض کنید که می خواهیم روی یک درخت جستجو با فاکتور انشعاب ۷، الگوریتم جستجوی BFS انجام دهیم و می دانیم که جواب در فاصله ۶ از گره‌ی اولیه قرار دارد اگر گسترش هر گره ۱ ثانیه طول بکشد، حداکثر زمانی که ما باید منتظر شویم تا الگوریتم به جواب برسد چقدر است؟

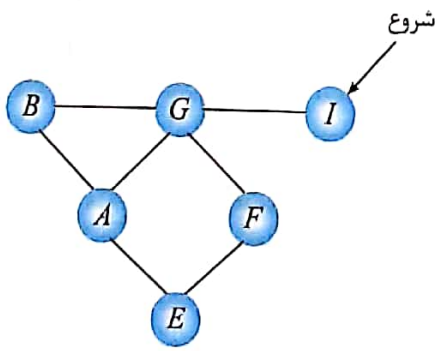
۱) ۸ ثانیه ۲) ۱+۷+۷۲+۷۳+۷۴+۷۵+۷۶+۷۷

۳) $۷^۶$ ثانیه ۴) ۱+۷+۷۲+۷۳+۷۴+۷۵+۷۶

۵- تفاوت الگوریتم dfs با جستجوی Backtracking در چیست؟

۱) میزان مصرف حافظه dfs کمتر است.
۲) هزینه اجرای dfs کمتر است.
۳) میزان مصرف حافظه dfs بیشتر است.
۴) هزینه اجرای dfs بیشتر است.

۶- در گراف زیر با استفاده از جستجوی درختی به ترتیب از چپ به راست چه گره‌هایی را مشاهده می‌کند؟ گره‌ها به ترتیب حروف الفبا دیده می‌شوند.



۱) I.G.A.B.F.B.E
۲) I.G.A.F.E
۳) I.G.A.E
۴) I.G.A.B.F.E

۷- ضریب انشعاب یک درخت جستجو ۳ است. دو جواب مسأله وجود دارد که یکی در آخرین رأسی است که در عمق ۴ قرار دارد، و دیگری در اولین راسی در عمق ۵ قرار دارد. در صورتی که از جستجوی عمیق‌شونده‌ی تکراری (IDS) استفاده کنیم، چه تعداد رأس باید بسط داده شوند تا اولین جواب بدست آید؟

۱) ۵۹ ۲) ۵۸ ۳) ۴۱ ۴) ۵۷

۸- می‌خواهیم در محیطی شطرنجی $n \times n$ جستجوی دوطرفه انجام دهیم. مبدأ جستجو نقطه‌ی (۰،۰) و هدف در نقطه‌ی (n, n) قرار دارد. در صورتیکه که هر دو طرف از BFS استفاده کنند (شروع حرکت با مبدأ) و گره‌های سمت چپ بر گره‌های پایین همچنین گره‌های بالا به گره‌ها سمت راست در بسط داده شدن اولویت داشته باشند. دو طرف در چه نقطه‌ای از صفحه به یکدیگر خواهند رسید؟ (شماره گذاری‌ها طبق جدول مختصات دکارت فرض شود)

۱) (n/۲, n/۲) ۲) (n,n) ۳) (ϕ,n) ۴) (n,ϕ)

۹- ضریب انشعاب یک درخت جستجو ۳ است. جواب مسأله در اوّلین رأسی است که در عمق ۵ قرار دارد. در صورتی که از جستجوی عمیق‌شونده‌ی تکراری (IDS) استفاده کنیم، چه تعداد رأس باید بسط داده شوند تا جواب بدست آید؟

۱) ۵۹ ۲) ۵۸ ۳) ۴۱ ۴) ۶۲

پاسخ تشریحی تست‌های تألیفی

۱) گزینه ۲ درست است.

با شروع از گروه B و به ترتیب از حروف کوچک به بزرگ جواب ب بدست می‌آید.

۲) گزینه ۴ درست است.

برای عدد ۱ از معمای هشت ، ۹ حالت وجود دارد. عدد ۲، ۸ حالت و ... که بدین ترتیب !۹ حالت ایجاد میگردد. ولی میتوان ثابت کرد که کل مسئله معمای هشت به دو قسمت که به هم مرتبط نیستند قابل تقسیم است. درواقع فضای حالت ۹!/۲ خواهد بود. برای اثبات این موضوع یک معیار با نام مقدار وارونگی یک حالت را بصورت زیر تعریف میکنیم:
مقدار وارونگی یک حالت برابر است با حاصلجمع وارونگی برای همه‌ی اعداد ۱..۸ معمای هشت. که مقدار وارونگی یک عدد در یک حالت برابر است با تعداد اعداد کوچکتر از خود که در ترتیب شکل زیر بعد از خود ظاهر شده باشند. میتوان براحتی ثابت کرد که هر حالتی که مقدار وارونگی آن زوج باشد با هیچیک از حرکتهای اعداد آن فرد نمیشود و بالعکس . بنابراین همه‌ی حالتهای زوج با هم و همه‌ی حالت های فرد با هم تشکیل دو مجموعه ی مجزا را میدهند . بنابراین فضای حالت نصف میشود.

۱	۲	۳
۴	۵	۶
۷	۸	۹

۳) گزینه ۴ درست است.

A دارای دو یال or است بنابراین B یا C کافی است. برای B ،
یا F با هزینه ۵ انتخاب شود یا D و E با هزینه جمعا ۴ انتخاب میگردد. با در نظر گرفتن هزینه هر یال برابر یک، بنابراین اگر F انتخاب شود جمعا هزینه ۷ خواهد بود. اگر هم D , E با هم انتخاب شوند، جمعا هزینه ۷ خواهد بود که برابر است و هر دو از حالت دیگر که F, G با هم است کمتر است. بنابراین گزینه د درست است.

۴) گزینه ۴ درست است.

اگر جواب در فاصله ۶ باشد، بایستی کلیه ی گره های سطح ۱ و ۲و ۳و ۴و ۵و ۶و ۷ تولید شوند و این به معنای آن است که سطحهای ۱..۶ گسترش یابند. (گسترش هر گره به معنای تولید گره های فرزندان آن گره است).

۵) در الگوریتم backtracking ، مقدار حافظه مورد نیاز O(d) است که از dfs کمتر است.

۶) گزینه ۱ درست است.

اگر به ترتیب FIFO گره‌ها در صف قرار داده شوند، پس از مشاهده I، G ، پس از G، ABF ، پس از A ، BE و ... ترتیب گزینه ۱ ترتیب مذکور است که البته الگوریتم در این حالت به اتمام نمی‌رسد و ادامه خواهد یافت .

۷) گزینه ۲ درست است.

IDS کم عمق ترین جواب را خروجی می‌دهد بنابراین گره ای که در آخرین راس عمق ۴ است، را خروجی می‌دهد. بنابراین تعداد تکرارها برابر با ۴ است و تعداد کل گره ها

سطح ۰ (به تعداد ۱ گره) : ۴ بار

سطح ۱ (به تعداد ۳گره) : ۳بار

سطح ۲ (به تعداد ۹ گره) : ۲بار

سطح ۳ (به تعداد ۲۷ گره) : ۱بار جمعا برابر است با ۱*۴ + ۳*۳ + ۹*۲ + ۲۷*۱ = ۵۸

۸) گزینه ۳ درست است.

بایستی در قطر اصلی مربع تداخل اتفاق بیافتد. ولی چون برای جستجوی forward، اولویت حرکت به بالا بیشتر از راست و برای جستجو backward، اولویت حرکت به چپ بیش از پائین است، بنابراین اولین محل تداخل اولین گره از قطر اصلی در مختصات (n , ۰) است.

۹) گزینه ۴ درست است.

بایستی یک dfs کامل با عمق ۴ تکمیل گردد و سپس یک شاخه فقط به عمق ۵ طی شود پس کل گره ها برابر است با dfs = به عمق یک + dfs به عمق دو + dfs به عمق سه + dfs به عمق چهار + یک مسیر از dfs به عمق پنج که مستلزم بسط ۴ گره است. تعداد کل گره‌ها برابر است با:

$$۴\times۱\times+۳\times۳+۳\times۳^{۲}+۳\times۳^{۳}+۴=۶۲$$

فصل چهارم

جستجوهای آگاهانه

در فصل ۳ اشاره شد که با استفاده از استراتژی‌های جستجوی ناآگاهانه و به کمک تولید حالت‌های جدید برای مسئله و مقایسه آنها با هدف مورد نظر می‌توان مسائل را حل کرد. اما متأسفانه این استراتژی‌ها در اغلب موارد بسیار ناکارآمد هستند. در این بخش خواهیم دید که چگونه استراتژی جستجوی آگاهانه (جستجویی که در آن دانش و اطلاعاتی درباره مسئله داریم) می‌تواند مسائل را به صورت کاراتری حل نماید. بخش ۴-۱ الگوریتم‌های فصل ۳ را در غالب جستجوی آگاهانه بررسی خواهد کرد و بخش ۴-۲ نشان می‌دهد که چگونه میتوانیم اطلاعات و دانش ضروری را برای حل یک مسئله خاص بیابیم. در بخش‌های ۴-۳ و ۴-۴ الگوریتم‌هایی مورد بحث قرار می‌گیرد که در آن‌ها تنها از جستجوی محلی در فضای حالت استفاده می‌شود. در این الگوریتم‌ها به جای کاوش سیستماتیک هدف از یک حالت اولیه، بررسی و اصلاح یک یا چند حالت جاری مسئله مورد توجه قرار می‌گیرد. این نوع الگوریتم‌ها برای مسائلی مناسب هستند که در آن‌ها هزینه مسیر اهمیتی ندارد بلکه تنها رسیدن به پاسخ مسئله حائز اهمیت است. در اینجا می توان به روش‌هایی که از فیزیک آماری (الگوریتم "شبیه سازی ذوب فلزات[۱]") و زیست تکاملی (الگوریتم" ژنتیک[۲]") الهام گرفته شده است اشاره نمود که از خانواده الگوریتم‌های جستجوی محلی به حساب می‌آیند. سرانجام، بخش ۴-۵ "جستجوی برخط[۳]" را بررسی می‌کند که در آن‌ها عامل با فضای حالتی روبرو است که کاملا ناشناخته است.

۴-۱- استراتژی‌های جستجوی آگاهانه (ابتکاری)

در این بخش خواهیم دید که چگونه استراتژی جستجوی آگاهانه (جستجویی که در آن دانش و اطلاعاتی علاوه بر تعریف خود مسئله داریم) مسائل را بسیار کاراتر از استراتژی جستجوی ناآگاهانه حل می‌کند.

نگرش کلی ما جستجوی بهترین- اولین[۴] است. جستجوی بهترین- اولین یک حالت خاص از جستجوی درخت یا جستجوی گراف است که در آن یک گره بر اساس تابع ارزیابی، $f(n)$ بسط داده می شود. از آنجا که تابع ارزیابی فاصله از هدف را اندازه گیری می‌کند به طور معمول، گره‌ای که دارای کمترین مقدار ارزیابی است بسط داده می‌شود. الگوریتم جستجوی بهترین- اولین را می‌توان همچون دیگر الگوریتم‌های جستجو با استفاده از حاشیه (لبه) (ساختمان داده‌ای است که عناصر موجود در صف را به ترتیب صعودی مقادیر تابع $f(n)$ در خود نگهداری می‌کند) پیاده‌سازی کرد. (صف اولویت)

۱- Simulated annealing
۲- Genetic
۳- Online search
۴- Best-first search

انتخاب نام "جستجوی بهترین- اولین" برای این الگوریتم قابل پذیرش اما نادقیق است زیرا اگر ما بتوانیم در جستجوی میان گره‌ها همواره اولین و بهترین گره را بسط دهیم آنگاه الگوریتم ما دیگر جستجو نخواهد بود بلکه حرکت مستقیم به سوی هدف است در حالی که ما در این الگوریتم گره‌ای را بسط می‌دهیم که براساس تابع ارزیابی به نظر می آید که بهترین گره است. اگر تابع ارزیابی ایده آل باشد آنگاه گره بسط داده شده واقعا بهترین گره خواهد بود اما در واقعیت گاهی اوقات تابع ارزیابی بسیار دور از واقعیت است و ممکن است مسیر جستجو را به گمراهی بکشاند. با این وجود ما از نام "جستجوی بهترین- اولین" برای این الگوریتم استفاده می‌کنیم زیرا نام "جستجوی ظاهرا بهترین- اولین" خوشایند نیست.

الگوریتم‌های بسیاری با توابع ارزیابی متفاوت از خانواده جستجوی بهترین-اولین وجود دارند. بخش کلیدی همه‌ی این الگوریتم‌ها تابعی به نام تابع ابتکاری[1] است که با h(n) نشان داده می‌شود :

h (n) = تقریبی است از هزینه‌ی کم هزینه‌ترین مسیر از گره n تا هدف.

به عنوان مثال، در کشور رومانی می توان هزینه کم هزینه‌ترین مسیر از شهر آراد تا بخارست را از طریق اندازه‌گیری خط مستقیم میان آراد و بخارست تقریب زد.

استفاده از تابع ابتکاری متداول‌ترین روش برای افزودن اطلاعات اضافی مرتبط با مسئله به الگوریتم جستجو است که در مورد آن‌ها در بخش ۴-۲ بیش‌تر بحث خواهیم نمود. فعلا، ما آن‌ها را به عنوان یک تابع اختیاری برای مسئله در نظر می گیریم با این فرض که: اگر گره n هدف باشد، آنگاه h(n)=۰ خواهد بود. در ادامه این بخش در مورد ۲ راه استفاده از اطلاعات توابع ابتکاری به عنوان راهنما در جستجوها اشاره خواهیم نمود.

۴-۱-۱- جستجوی بهترین-اولین حریصانه

جستجوی بهترین-اولین حریصانه[2] در مسئله‌هایی که احتمال رسیدن سریع به جواب وجود دارد تلاش می‌کند که نزدیک‌ترین گره به هدف را بسط دهد بنابراین گره‌ها را با استفاده از تابع ابتکاری ارزیابی می کند: f(n) = h(n)

حال می‌خواهیم نشان دهیم که چگونه این الگوریتم با استفاده از "طول خط مستقیم[3]" به عنوان تابع ابتکاری که با h_{SLD} آن را نمایش می دهیم در مسئله مسیریابی کشور رومانی عمل می کند. اگر هدف رفتن به بخارست باشد، آنگاه ما به اندازه خط مستقیم میان شهرها تا بخارست نیازمندیم که در شکل شماره۴-۱ نشان داده شده است. به عنوان مثال، $h_{sld}(In(Arad)) = ۳۶۶$. دقت داشته باشید که مقادیر h_{sld} از روی صورت مسئله قابل محاسبه و اندازه‌گیری نیست. علاوه بر این تشخیص این مسئله که "h_{sld} وابسته به فاصله واقعی راه‌هاست، بنابراین تابع ابتکاری مفیدی محسوب می شود" نیازمند کمی زمان و تجربه است.

۱- Heuristic function
۲- Greedy best-first search
۳- Straight line distance (SLD)

Arad	۳٦٦	*Mehadia*	۲٤۱
Bucharest	۰	*Neamt*	۲۳٤
Craiova	۱٦۰	*Oradea*	۳۸۰
Dobreta	۲٤۲	*Pitesti*	۱۰۰
Eforie	۱٦۱	*Rimnicu Vilcea*	۱۹۳
Fagaras	۱۷٦	*Sibiu*	۲۵۳
Giurgiu	۷۷	*Timisoara*	۳۲۹
Hirsova	۱۵۱	*Urziceni*	۸۰
Iasi	۲۲٦	*Vaslui*	۱۹۹
Lugoj	۲٤٤	*Zerind*	۳۷٤

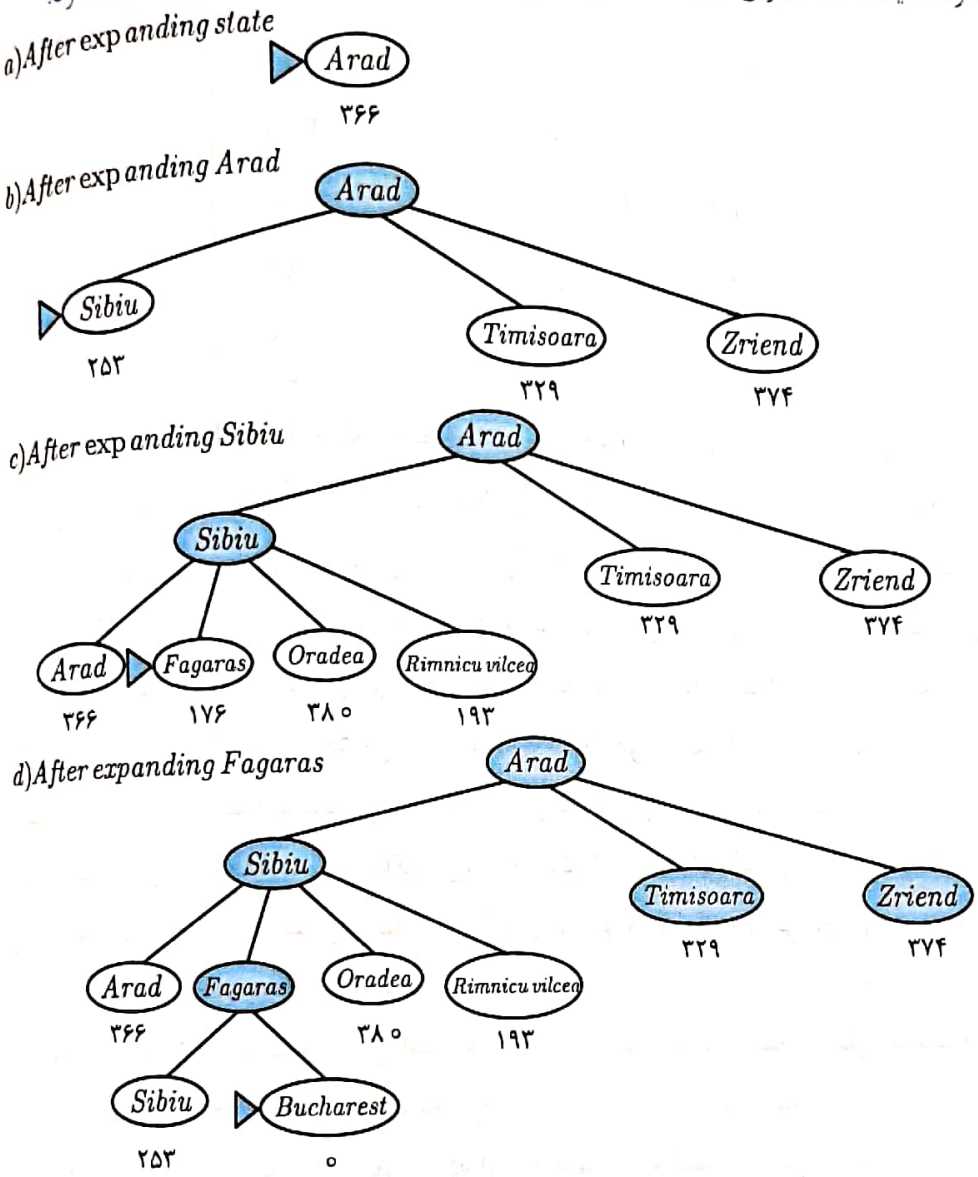
شکل ۴-۱ مقادیر hSLD - طول خط مستقیم تا بخارست.شکل شماره ۴-۲ روند اجرای الگوریتم "جستجوی بهترین – اولین حریصانه" که از تابع ابتکاری h_{sld} استفاده می کند را برای پیدا کردن مسیری بین آراد و بخارست نشان می‌دهد. در این شکل بعد از آراد گره سیبیو به عنوان اولین گره بسط داده شده است، زیرا این گره از ۲ گره زریند و تیمیسوارا به هدف نزدیک‌تر است. گره بعدی، گره فاگاراس است، زیرا نزدیک ترین گره به هدف است. پس از آن فاگاراس گره بخارست را که همان هدف است نتیجه می‌دهد. در این مسئله بخصوص، الگوریتم جستجوی اولین – بهترین با استفاده از h_{sld} جواب مسئله را بدون بسط حتی یک گره اضافی (که در مسیر حل مسئله نباشد) بدست آورد، بنابراین هزینه جستجو مینیمم می‌باشد. هر چند، این جواب بهینه نیست زیرا مسیر شامل شهرهای سیبیو، فاگاراس، بخارست ۳۲ کیلومتر طولانی‌تر از مسیر گذرنده از شهرهای ریمنیسو ویلسا، پیتستی است. به همین دلیل این الگوریتم حریصانه نامیده می‌شود زیرا در هر مرحله سعی می کند هرچه قدر که می‌تواند به هدف نزدیکتر شود.

اما بسط گره با h(n) مینیمم ممکن است ما را دچار اشتباه کند، به عنوان مثال به مسئله‌ی رفتن از شهر آیسی[۱] به شهر فاگاراس توجه کنید، با توجه به تابع ابتکاری، باید اول گره نیمت[۲] را بسط دهیم زیرا این گره نزدیک-ترین گره به فاگاراس است، در صورتی که این گره بن‌بست است و راهی به سوی فاگاراس ندارد. برای حل این مسئله ابتدا باید به شهر واسلوی[۳] رفت (راهی که براساس تابع ابتکاری از هدف دورتر است) و پس از آن به اورزیسنی[۴] و بخارست و فاگاراس. در این مسئله تابع ابتکاری چند گره غیرضروری را بسط می دهد. از این گذشته، اگر ما وجود حالت‌های تکراری را چک نکنیم جواب مسئله هیچگاه پیدا نمی‌شود، زیرا جستجوی ما بین دو شهر نیمت و آیسی نوسان می‌کند. جستجوی بهترین ـ اولین از این جهت که همواره در حل مسئله تنها یک مسیر را دنبال می‌کند و در صورت رسیدن به بن بست به عقب برگشت می‌کند بسیار شبیه جستجوی عمق -

---

۱-Iasi
۲-Neamt
۳- Vaslui
۴-Urziceni

اول است به همین دلیل همچون جستجوی عمق - اول اشکالاتی نظیر بهینه نبودن و کامل نبودن را داراست **(زیرا ممکن است الگوریتم یک مسیر بی‌پایان را شروع کرده و هیچگاه به عقب برگشت نکند تا شانس‌های دیگر را امتحان کند)**. در بدترین حالت، پیچیدگی الگوریتم از نظر زمانی و فضای حافظه برابر $O(b^m)$ خواهد بود که در آن m ماکزیمم عمق فضای جستجو است اگرچه در صورت استفاده از یک تابع ابتکاری مناسب این پیچیدگی به میزان زیادی کاهش خواهد یافت که میزان این کاهش بستگی به نوع مسئله و کیفیت تابع ابتکاری دارد.



شکل ۴-۲ مراحل بهترین - اولین جستجوی حریصانه برای بخارست با استفاده ابتکار طول خط مستقیم hSLD گره ها با مقادیر تابع h برچسب‌گذاری شده‌اند.

۴-۱-۲- جستجوی A* : مینیمم کردن هزینه کلی تقریبی جواب مسئله:

جستجوی A* (A استار) مشهورترین نوع از انواع الگوریتم‌های جستجوی بهترین- اولین است. این الگوریتم گره‌ها را توسط ۲ تابع g(n) (هزینه رسیدن از حالت اولیه به گره n) و h(n) (هزینه رسیدن از گره n به هدف) ارزیابی می‌کند:

$$f(n) = g(n) + h(n)$$

از آنجایی g(n) هزینه رسیدن از گره اولیه تا گره n را محاسبه و h(n) هزینه کم هزینه‌ترین مسیر برای رفتن از گره n به هدف را تقریب می‌زند، داریم:

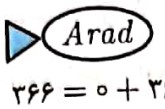
f (n) = تخمین و تقریبی از هزینه کم هزینه‌ترین راه حل برای رسیدن به هدف از طریق گره n.

بنابراین، برای اینکه کم‌هزینه‌ترین جواب را بیابیم منطقی است که ابتدا گره‌ای با کم‌ترین مقدار g(n) + h(n) را بسط دهیم. نتیجه اینکه این استراتژی نه تنها منطقی به نظر می‌رسد بلکه با گذاشتن شروطی برای تابع ابتکاری h(n) این الگوریتم هم بهینه و هم کامل خواهد بود.

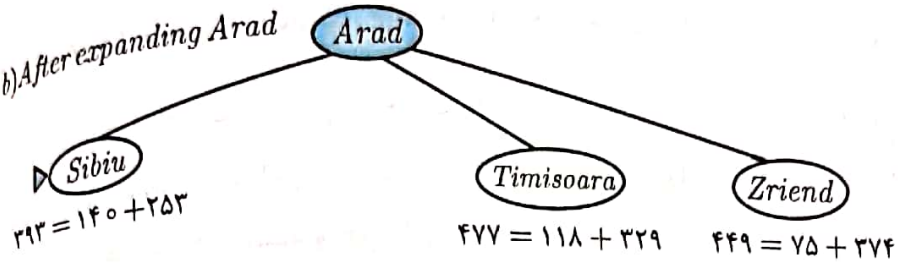
جستجوی A* بهینه است در صورتی که در ساختار داده درخت به کار گرفته شود و h(n) یک تابع ابتکاری قابل پذیرش۱ باشد (بدین معنی که h(n) هیچگاه هزینه رسیدن به هدف را زیادتر از مقدار واقعی تقریب نزند). توابع ابتکاری قابل پذیرش به ذات خوش‌بین هستند، زیرا آنها هزینه حل مسئله را کم‌تر از مقدار واقعی تقریب می زنند. از آن جایی که g(n) هزینه واقعی رسیدن به گره n است براحتی می‌توان نتیجه گرفت که f(n) هیچگاه هزینه رسیدن به هدف از طریق گره n را بیش‌تر از هزینه واقعی حل مسئله تقریب نمی‌زند.

یک مثال واضح از تابع ابتکاری قابل پذیرش، تابع ابتکاری «طول خط مستقیم» h_{SLD} است که از آن برای رفتن به بخارست استفاده کردیم. تابع ابتکاری طول خط مستقیم، قابل پذیرش است زیرا کوتاه‌ترین مسیر بین دو نقطه همان خط مستقیم میان آن دو نقطه است بنابراین هیچگاه خط مستقیم زیادتر از مقدار واقعی تقریب نمی‌زند. در شکل ۴-۳ روند اجرای الگوریتم جستجوی درختی A* برای رفتن به بخارست نشان داده شده است. مقادیر تابع g بر اساس شکل ۳-۲ و مقادیر تابع h_{SLD} در شکل ۴-۱ داده شده است. دقت داشته باشید که گره بخارست اولین بار در مرحله e در حاشیه ظاهر اما بسط داده نشد زیرا مقدار تابع f آن (۴۵۰) بیشتر از پیتستی (۴۱۷) بوده است. به عبارت دیگر، ممکن است جواب دیگری برای مسئله از مسیر گره پیتستی وجود داشته باشد که هزینه آن نزدیک به ۴۱۷ باشد، بنابراین الگوریتم هزینه ۴۵۰ را انتخاب نمی‌کند. با دقت در این مثال می‌توان به طور کلی اثبات کرد که جستجو A* در ساختار داده درخت بهینه است به این شرط که h(n) قابل پذیرش باشد.

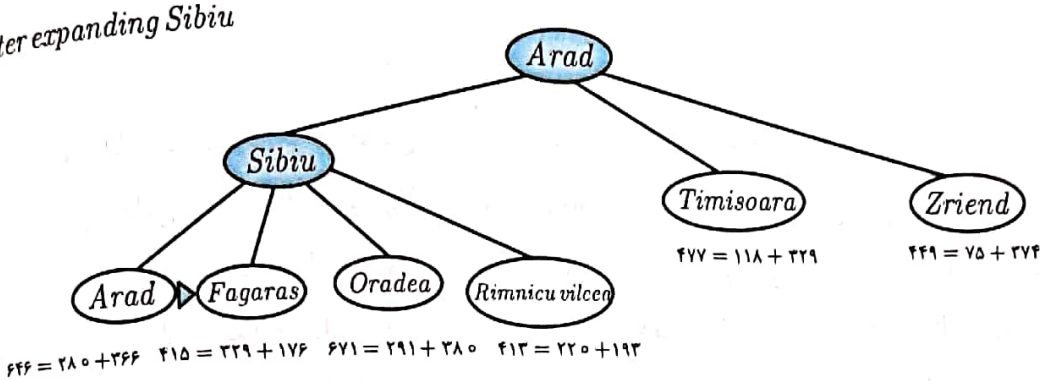
a)The int ital state



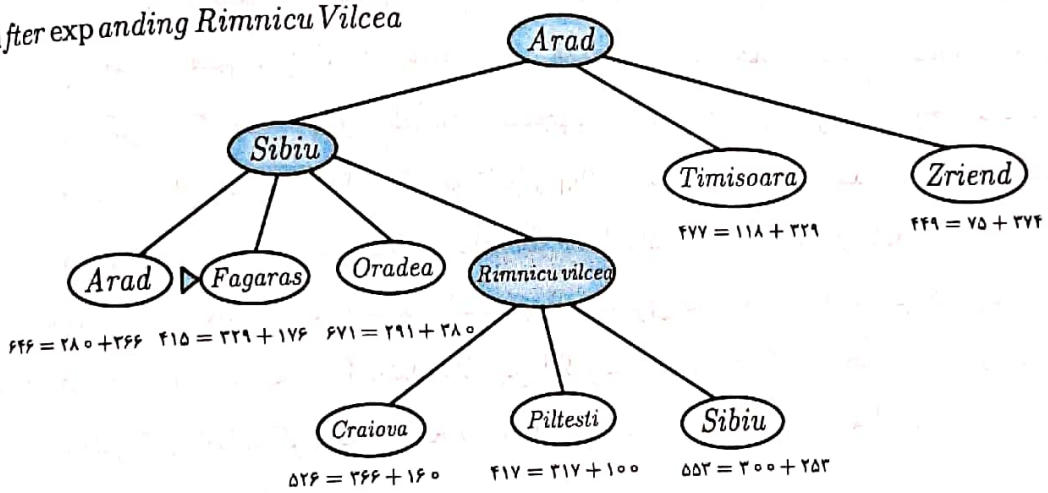
b)After expanding Arad



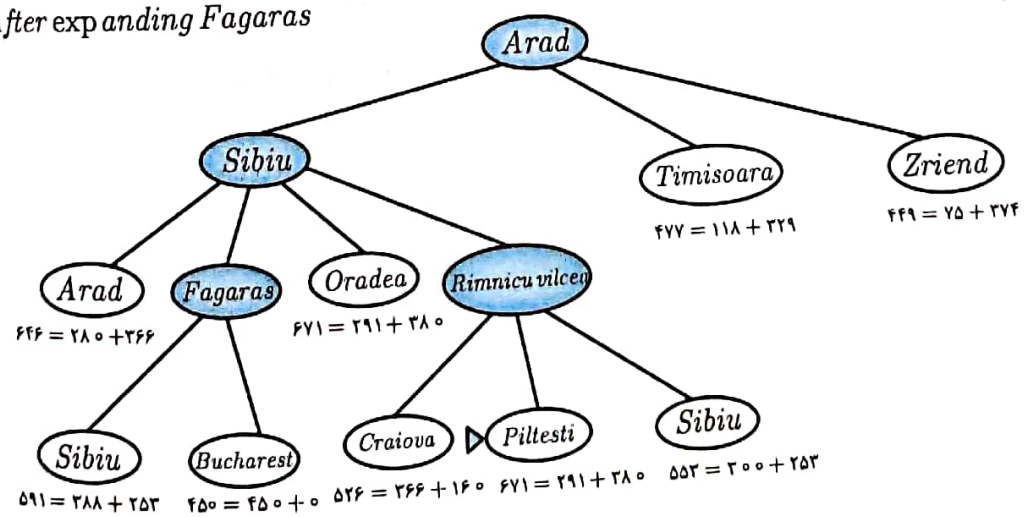
c)After expanding Sibiu

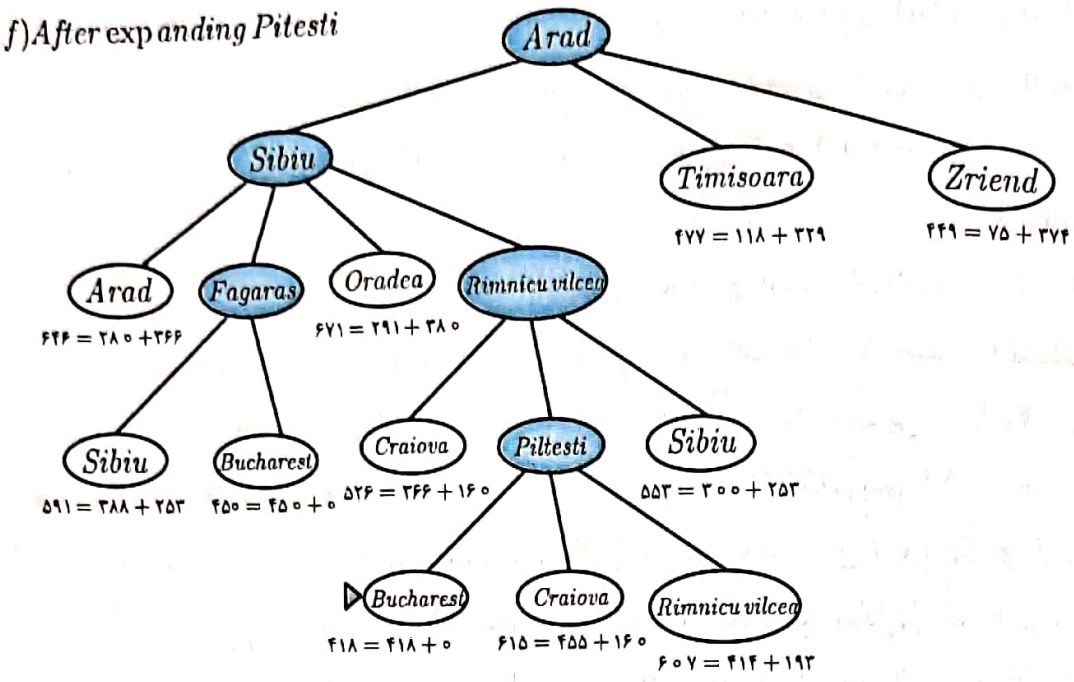


d)After exp anding Rimnicu Vilcea



e)After exp anding Fagaras





شکل ۴-۳ مراحل جستجوی A* برای بخارست. گره‌ها با مقادیر f=g+h برچسب‌گذاری شده‌اند. مقادیر h همان طول خط مستقیم تا بخارست است که از شکل ۴-۱ گرفته شده است.

فرض کنید، گره G۲ که یک هدف (جواب) غیر بهینه برای مسئله است در حاشیه ظاهر شود و فرض کنید جواب بهینه مسئله نیز C* باشد، آنگاه، چون G۲ غیربهینه است و h(G۲) = ۰ (این عبارت به ازای همه‌ی هدف‌های مسئله درست است) داریم:

$$f(G۲) = g(G۲) + h(G۲) = g(G۲) > C^*$$

حال به گره‌ای مانند گره n که در حاشیه است و در ضمن در مسیر جواب بهینه(در صورتی که مسئله جواب داشته باشد همواره چنین گره‌ای وجود دارد) قرار دارد توجه کنید (به عنوان مثال پیتستی در مثال پاراگراف قبلی) اگر h(n) هزینه بقیه مسیر رسیدن به هدف را بیش‌تر از مقدار واقعی تخمین نزند، آنگاه داریم:

$$f(n) = g(n) + h(n) \leq C^*$$

بنابراین، نشان دادیم که $f(n) \leq C^* < f(G۲)$، پس G۲ بسط داده نمی‌شود و A* جواب بهینه مسئله را بدست می‌آورد.

اگر ما به جای الگوریتم جستجو در درخت از الگوریتم جستجو در گراف شکل ۳-۱۹ استفاده کنیم آنگاه این اثبات درست نخواهد بود. زیرا در این صورت الگوریتم جستجوی گراف ممکن است جواب غیربهینه بدهد و جواب بهینه مسئله را به این دلیل که یک حالت تکراری در جستجو است حذف کند. (تمرین شماره ۴-۴).

برای حل این مشکل ۲ راه وجود دارد، راه حل اول این است که الگوریتم جستجوی گراف از بین ۲ مسیری که به یک گره ختم می شوند، مسیر با هزینه بیش‌تر را حذف بکند (به بخش ۳-۵ مراجعه کنید) اگرچه نگه داشتن اطلاعات اضافی آن هزینه‌بر است اما بهینه بودن الگوریتم را تضمین می کند. راه حل دوم این است که تضمین کنیم همواره مسیر بهینه مسئله در صورت ظهور حالت‌های تکراری، همان اولین حالت اتفاق افتاده باشد – همچون «جستجوی هزینه یکنواخت». اگر ما محدودیت جدیدی را روی h(n) اعمال کنیم این ویژگی برقرار

می‌شود که به آن خاصیت سازگاری[۱] (و یا یکنوایی[۲]) گویند. تابع ابتکاری h(n) سازگار است اگر به ازای هر گره n و n' (گره پسین n) که با کنش a تولید شده است، هزینه تخمینی رفتن از گره n به هدف بیش‌تر از حاصل جمع هزینه تخمینی رفتن از گره n' به هدف و هزینه رفتن از n به n' نباشد:

$$h(n) \leq C(n,a,n') + h(n')$$

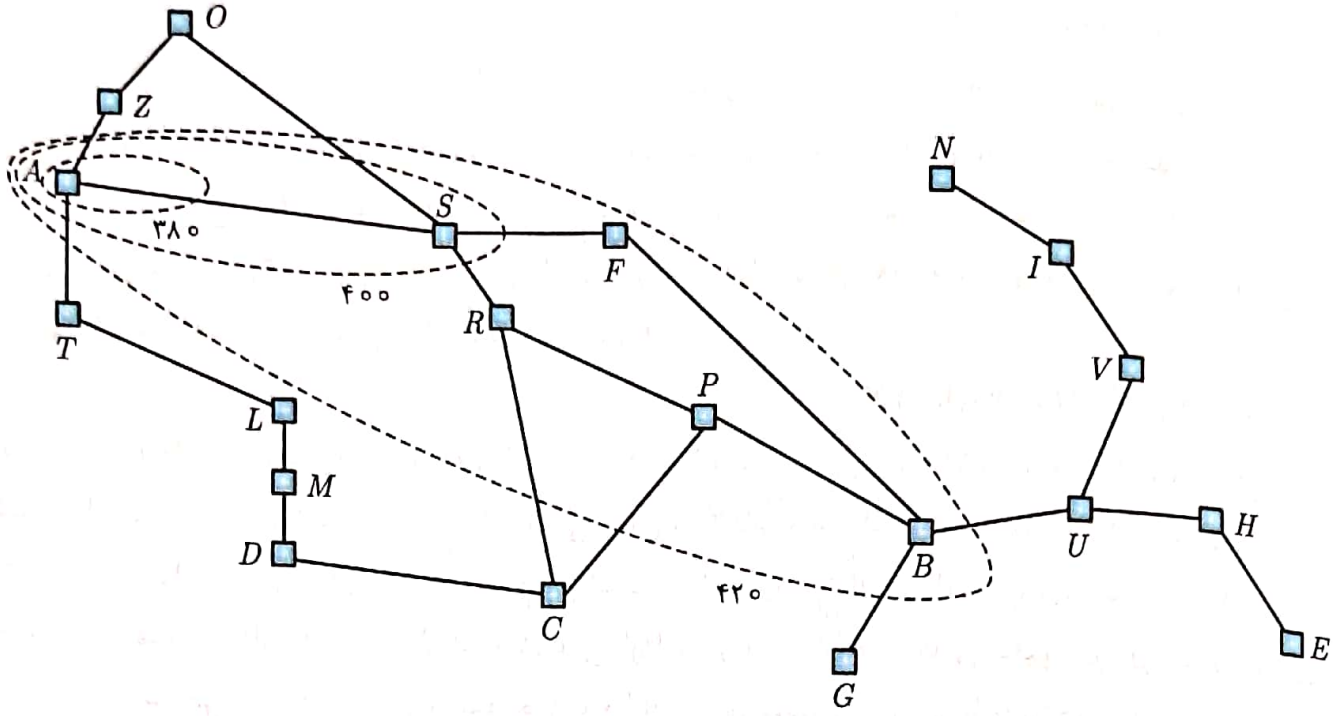
این نامساوی نوعی نامساوی مثلث است (به این معنی که هیچ کدام از اضلاع مثلث از حاصل جمع دو ضلع دیگر بزرگ‌تر نیست) حال آنکه ۳ گره n و n' و نزدیک‌ترین گره هدف به n یک مثلث را تشکیل می‌دهند. براحتی می توان نشان داد که هر تابع ابتکاری سازگار، قابل پذیرش نیز است (تمرین شماره۴-۷). در هر حال مهم‌ترین نتیجه‌ای که از سازگاری تابع ابتکاری می‌توان گرفت این است که الگوریتم A* در جستجوی گراف بهینه است. اگرچه شرط سازگاری بسیار محدود کننده‌تر از قابل پذیرش بودن برای تابع ابتکاری است اما به ندرت توابعی را می‌توان یافت که قابل پذیرش باشند اما سازگار نباشند. همه‌ی توابع ابتکاری که ما در این فصل از آن‌ها صحبت می‌کنیم سازگار نیز هستند. به عنوان مثال به h_{SLD} توجه کنید، می‌دانیم که نامساوی مثلث برای خطوط مستقیم برقرار است و چون خط مستقیم بین n و n' از C(n,a,n') بزرگتر نیست، بنابراین h_{SLD} تابعی سازگار است. نتیجه‌ی مهم دیگری که می‌توان گرفت این است که: اگر h(n) تابعی سازگار باشد، آنگاه مقادیر تابع f(n) در همه‌ی مسیرها غیر نزولی خواهند بود که از تعریف سازگاری به راحتی می‌توان آن را اثبات نمود: فرض کنید n' گره پسین n باشد، آنگاه به ازای مقادیری از a داریم g(n') = g(n) + C(n,a,n'). از طرفی داریم:

$$f(n') = g(n') + h(n') = g(n) + C(n,a,n') + h(n') \geq g(n) + h(n) = f(n)$$

بنابراین می‌توان نتیجه گرفت که دنباله گره‌های بسط داده شده توسط الگوریتم A* در جستجوی گراف به ترتیب غیرنزولی مقادیر تابع f(n) قرار دارند. لذا اولین گره هدفی که بسط داده می‌شود، پاسخ بهینه‌ی مسئله است، و بعد از آن هر گره‌ای که بسط داده شود حداقل مقداری برابر گره هدف بهینه خواهد داشت.

این واقعیت که مقادیر تابع f در تمامی طول مسیر غیر نزولی هستند به این معنی است که ما می‌توانیم در فضای حالت خطوط سطح تراز[۳] را رسم نمائیم دقیقا مشابه خطوط تراز در نقشه برداری. این موضوع در شکل ۴-۴ نمایش داده شده است. همه‌ی گره‌های داخل خط تراز با برچسب ۴۰۰ دارای f(n) کوچکتر یا مساوی ۴۰۰ می‌باشند، این موضوع در مورد خطوط تراز دیگر نیز صادق است. از آنجایی که الگوریتم A* گره‌هایی با کوچک‌ترین مقدار تابع f(n) موجود در حاشیه را بسط می‌دهد، مشاهده می‌کنید که جستجوی A* از گره اولیه خارج شده و گره‌ها را به صورت حلقه‌هایی هم مرکز براساس افزایش مقادیر تابع f بسط می‌دهد. در الگوریتم جستجوی هزینه‌ی یکنواخت (جستجوی A* با h(n) = ۰) این حلقه‌ها به صورت دوایری حول حالت اولیه (گره اولیه) خواهند بود. در توابع ابتکاری دقیق، این حلقه‌ها حول هدف گسترش می‌یابند تا اینکه اطراف مسیر بهینه باریک‌تر شده و بر روی آن متمرکز می‌شوند. اگر C* هزینه جواب بهینه مسئله باشد آنگاه، داریم:
A* همه‌ی گره‌هایی با ویژگی f (n) < C* را بسط می‌دهد.

۱- Consistency
۲- Monotonicity
۳- Contour



شکل ۴-۴ نقشه کشور رومانی که خطوط سطح تراز در f = ۳۸۰ و f = ۴۲۰ را نشان می‌دهد که در آن آراد به عنوان حالت اولیه است. گره‌های درون خطوط سطح تراز دارای مقادیر f کوچکتر یا مساوی مقدار تراز هستند.

A* ممکن است بعضی از گره‌های روی «حلقه هدف» (جایی که f (n) = C*) را قبل از انتخاب گره هدف بسط دهد.

به طور شهودی، واضح است که اولین جواب، همان جواب بهینه‌ی مسئله است، زیرا تمامی گره‌های هدف دیگر که در حلقه‌های بعدی قرار دارند دارای مقدار f(n) بیش‌تری هستند، بنابراین مقدار g(n) بالاتری دارند (زیرا در تمامی گره‌های هدف h(n) = ۰ است). همچنین به طور شهودی واضح است که الگوریتم جستجوی A* کامل است زیرا همینطور که ما حلقه‌ها را براساس افزایش مقادیر f(n) اضافه می‌کنیم، سرانجام به حلقه‌ای می‌رسیم که در آن مقدار f(n) برابر هزینه‌ی مسیر رسیدن به حالت هدف است.

توجه داشته باشید که A* هیچ گره‌ای با ویژگی f(n) > C* را بسط نخواهد داد (برای مثال، تیمیسوارا[1] اگرچه در شکل۴-۳ فرزند ریشه است، هیچگاه بسط داده نشده‌است). (در این حالت) می‌گوییم زیردرخت مجاور تیمیسوارا هرس شده است (از آنجا که h_{SLD} قابل پذیرش است) الگوریتم می‌تواند با اطمینان این قسمت از درخت را نادیده گرفته و همچنان بهینه باقی بماند.

موضوع هرس کردن (حذف احتمال وقوع بعضی حالات بدون اینکه آنها را امتحان کرده باشیم) در بسیاری از زمینه‌های هوش مصنوعی بسیار حائز اهمیت است.

در مورد الگوریتم‌های بهینه از این نوع (الگوریتم‌هایی که در آنها جستجو از ریشه آغاز و بسط می‌یابد) الگوریتم A* به ازای هر تابع ابتکاری دلخواه به طور بهینه‌ای کارا است، بدین معنی که الگوریتم بهینه‌ی دیگری وجود ندارد که در آن تعداد گره‌های بسط داده شده کمتر از الگوریتم A* باشد. (به جز احتمالاً زمان‌هایی که انتخاب بین گره‌هایی با f(n) = C* اتفاق می‌افتد). دلیل این امر این است که در الگوریتم‌هایی که تمامی گره‌های با ویژگی f(n) < C* بسط داده نمی‌شود، این خطر وجود دارد که جواب بهینه‌ی مسئله را از دست بدهند.

[1]- Timisoara

الگوریتم جستجوی A* کامل، بهینه، و در میان الگوریتم‌هایی از این نوع به طور بهینه‌ای کارا است، بنابراین بیش از حد انتظار رضایت‌بخش است. متأسفانه، این بدین معنی نیست که A* تمامی نیازهای ما برای جستجو را حل می‌کند. مشکل اینجاست که در بسیاری از مسائل، نسبت تعداد گره‌های درون حلقه هدف به اندازه جواب مسئله، نمایی است. اگرچه اثبات این بحث در چهارچوب این کتاب نیست، اثبات شده‌است که این رشد نمایی اتفاق می‌افتد، مگر اینکه رشد خطا در تابع ابتکاری از لگاریتم هزینه‌ی مسیر واقعی سریعتر نباشد. در ریاضی، شرط رشد کمتر از نمایی را اینگونه نشان می‌دهند:

$$|h(n) - h^*(n)| \leq O(\log(h^*(n)))$$

که در آن $h^*(n)$ هزینه‌ی واقعی رفتن از گره n به هدف است. در عمل برای اکثر توابع ابتکاری، حداقل خطا، نسبتی از هزینه‌ی مسیر است و رشد نمایی ناشی از آن سرانجام بر هر کامپیوتری غلبه می‌کند. به همین دلیل اغلب، تلاش برای یافتن جواب بهینه، غیر عملی است. می‌توان با استفاده از الگوریتم‌های مختلف A* بسرعت جواب غیربهینه مسئله را بدست آورد، ویا گاهی اوقات می‌توان توابع ابتکاری دقیق‌تری (و نه دقیقاً قابل پذیرشی) را طراحی کرد. در هر حال، استفاده از تابع ابتکاری مناسب، در مقایسه با جستجوی ناآگاهانه، همچنان مقدار زیادی صرفه‌جویی می‌کند. در بخش ۴-۲ به موضوع طراحی تابع ابتکاری مناسب خواهیم پرداخت. با این وجود مهم‌ترین عیب A* زمان محاسباتی آن نیست بلکه از آنجایی که این الگوریتم تمامی گره‌های تولید شده را در حافظه نگه می‌دارد (همچون تمامی الگوریتم‌های جستجو در گراف)، معمولاً قبل از اینکه دچار کمبود زمان بشود دچار کمبود حافظه می‌شود. به همین دلیل A* برای مسئله‌هایی با مقیاس بزرگ عملی نیست. الگوریتم‌های جدید گسترش یافته مشکل کمبود فضا را بدون نقض بهینه بودن و کامل بودن، به ازای مدت زمان اجرای کوتاهی، حل کرده‌اند که در مباحث بعدی بحث خواهد شد.

۴-۱-۳- جستجوی ابتکاری در حافظه کران‌دار

ساده‌ترین راه برای کاهش نیاز به حافظه در A* استفاده از ایده‌ی الگوریتم عمیق‌شونده‌ی تکراری[1] در جستجوی تابع ابتکاری است که نتیجه‌ی آن الگوریتم عمیق‌شونده تکراری A* (IDA*) است. مهم ترین تفاوت IDA* با الگوریتم عمیق شونده تکراری استاندارد این است که در آن به جای برش بر اساس عمق، برش‌ها براساس مقادیر $f(n) = (g(n) + h(n))$ انجام می‌شوند. در هر بار تکرار، مقدار برش، برابر کوچک‌ترین مقدار f گره‌ای است که از مقدار برش قبلی تجاوز کرده‌است. IDA* در بسیاری از مسائل که در آن‌ها هزینه‌ی گام‌ها برابر واحد است عملی است و از تحمیل سربار بسیار زیاد مربوط به نگه‌داشتن صف مرتب شده‌ای از گره‌ها خودداری می‌کند. متأسفانه، مشکل این الگوریتم همانند جستجوی هزینه‌ی یکنواخت تکراری (تمرین۳-۱۱) مقادیر حقیقی هزینه‌هاست. در این بخش به طور خلاصه، دو الگوریتم با حافظه‌ی کران‌دار به نام‌های RBFS و MA* بررسی خواهد شد.

[1]- Iterative Deepening

```
function Recursive-Best-First-Search(problem) returns a solution, or failure
    RBFS(problem, Make-Node(Initial-State [problem]), ∞)

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f-cost limit
    if Goal-Test[problem](State[node]) then return node
    successor ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) +h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f[best] > f-limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best]← RBFS(problem, best, min(f-limit, alternative))
        if result ≠ failure then return result
```

شکل ۴-۵ الگوریتم برای جستجوی بهترین- اولین بازگشتی.

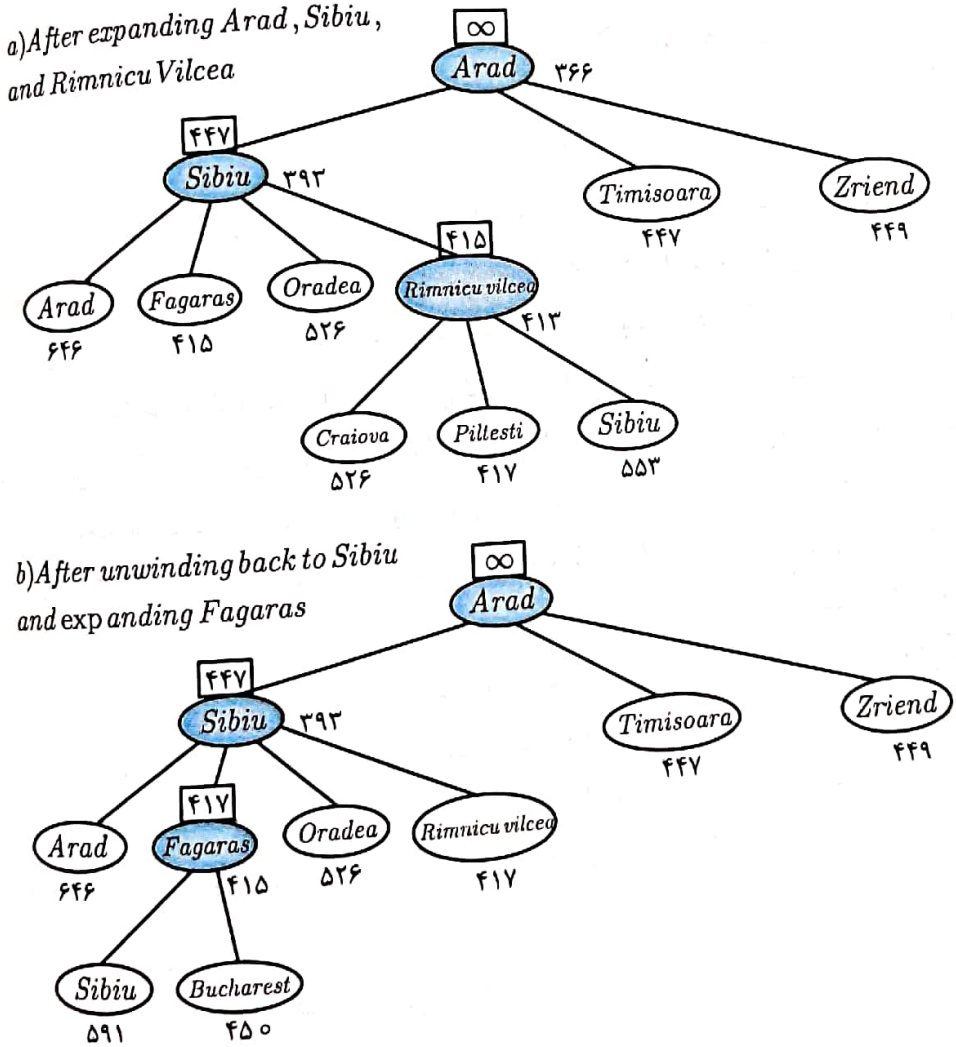
جستجوی اولین-بهترین بازگشتی[1] (RBFS) یک الگوریتم بازگشتی ساده است که تلاش می‌کند الگوریتم جستجوی بهترین- اولین استاندارد را در فضای حافظه خطی شبیه سازی کند. این الگوریتم در شکل ۴-۵ نشان داده شده‌است. ساختار آن بسیار شبیه به جستجوی عمق ـ اول بازگشتی است اما با این تفاوت که راه خود را به صورت بی‌پایان در مسیر جاری ادامه نمی‌دهد، بلکه به دنبال مقدار f بهترین مسیر ثانویه موجود در اجداد گره جاری می‌رود. اگر گره جاری از این حد(مقدار مسیر ثانویه) تجاوز کند، آنگاه الگوریتم به مسیر ثانویه برگشت می‌کند. هنگامیکه الگوریتم بازگشت می‌کند، در طول مسیر، بهترین مقدار f فرزندان هر گره را با مقدار f آن گره جایگزین می‌کند. به این ترتیب، RBFS مقدار f بهترین برگ را در زیر درخت فراموش شده به خاطر می‌سپارد. و در نتیجه می‌تواند در آینده در مورد با ارزش‌ترین زیر درختی که می‌خواهد دوباره بسط دهد تصمیم‌گیری نماید. شکل شماره ۴-۶ نشان می‌دهد که چگونه RBFS به بخارست می‌رسد. RBFS نسبت به IDA* کاراتر است، اما همچنان دارای مشکل تعدد بسط تکراری گره‌هاست. در مثال شکل ۴-۶ RBFS ابتدا مسیر را از طریق گره ریمنیسو ویلسا دنبال می‌کند سپس تغییر عقیده می‌دهد، و فاگاراس را امتحان می‌کند، اما دوباره نظرش عوض می‌شود و برگشت می‌کند. این تغییر عقیده‌ها به این دلیل روی می‌دهد که هر بار که بهترین مسیر جاری بسط داده می‌شود، این احتمال وجود دارد که مقدار f آن افزایش یابد (زیرا h معمولا برای گره‌های نزدیک به هدف کمتر خوش بین است) . وقتی چنین اتفاقی می‌افتد (مخصوصا در فضاهای حالت بزرگ) بهترین مسیر ثانویه جایگزین بهترین مسیر می‌شود، بنابراین الگوریتم جستجو بازگشت می‌کند تا آن را ادامه دهد. هر کدام از این تغییر نظرها مشابه یک تکرار در IDA* است و نیاز به گسترش دوباره گره‌های فراموش شده برای ساختن دوباره بهترین مسیر و سپس بسط یک گره دیگر دارد.

RBFS نیز مشابه A* در صورتی که h(n) قابل پذیرش باشد الگوریتمی بهینه است. پیچیدگی فضای حافظه آن O(bd) است، اما محاسبه پیچیدگی زمانی آن کمی مشکل است: هم به دقت تابع ابتکاری و هم به تعداد تغییر

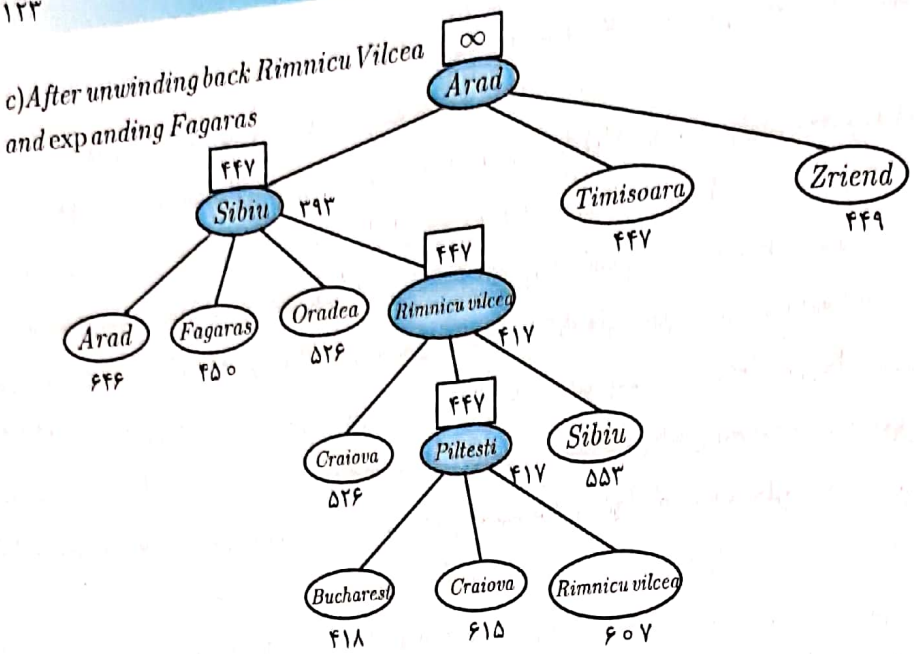
[1]- Recursive Best-First Search

بهترین مسیر در طی بسط گره‌ها وابسته است. در مورد IDA* و RBFS احتمال افزایش نمایی پیچیدگی مرتبط با جستجو بر روی گراف (به بخش ۳-۵ مراجعه کنید) وجود دارد زیرا در این الگوریتم‌ها امکان بررسی حالت‌های تکراری (به جز در مسیر جاری) وجود ندارد. بنابراین ممکن است حالت‌های تکراری را چندین بار جستجو کنند. اشکال IDA* و RBFS استفاده بسیار کم آن‌ها از حافظه است به طوری که اگر مقدار حافظه بیش‌تری موجود باشد، آن‌ها راهی برای استفاده از آن ندارند. دربین تکرارها، IDA* تنها یک عدد نگه می‌دارد: کران بالای مقدار f جاری.RBFS اطلاعات بیش‌تری نگه می‌دارد و تنها O (bd) حافظه مصرف می‌کند.

منطقی به نظر می‌رسد که از همه‌ی فضای حافظه موجود استفاده کنیم. دو الگوریتمی که این کار را انجام می‌دهند عبارتند از MA* (A* با حافظه محدود[1]) و SMA* (شکل ساده[2]MA*). ما الگوریتم SMA* را که بسیار ساده‌تراست توضیح خواهیم داد.SMA* همچون الگوریتمA*، بهترین برگ را بسط می‌دهد تا زمانی که حافظه پر شود. در این لحظه، دیگر نمی‌تواند گره جدیدی را به درخت جستجواضافه کند مگر اینکه یک گره از گره‌های قدیمی را دور بیاندازد.



[1]-Memory Bounded A*
[2]-Simplified MA*



شکل ۴-۶ مراحل جستجوی RBFS برای کوتاه‌ترین مسیر تا بخارست. مقدار محدود f برای هر یک از فراخوان‌های بازگشتی، بالای گره جاری نشان داده شده است. (a) مسیر گذرنده از ریمنیسو ویلسا تا زمانی که مقدار بهترین گره جاری (پیتستی) از مقدار بهترین مسیر ثانویه (فاگاراس) بدتر باشد، ادامه می‌یابد. (b) الگوریتم بازگشت کرده و مقدار بهترین برگ فراموش شده(۴۱۷) را به ریمنیسو ویلسا برمی‌گرداند. آنگاه فاگاراس بسط داده می‌شود و مقدار بهترین برگ را(۴۵۰) آشکار می‌کند.(c) الگوریتم بازگشت می‌کند و بهترین مقدار برگ زیردرخت فراموش شده (۴۵۰) به سمت فاگاراس برگشت داده می‌شود، آنگاه ریمنیسو ویلسا بسط داده می‌شود (این دفعه، به این دلیل که بهترین مسیر ثانویه (از مسیر گره تیمیسوارا) دارای حداقل هزینه ۴۴۷ است) بسط تا Bucharest ادامه پیدا می‌کند.

*SMA همواره بدترین برگ را دور می‌اندازد (گره‌ای که بیش‌ترین مقدار f را داراست). همچون RBFS، *SMA مقدار گره فراموش شده را به والد آن بر می‌گرداند. به این ترتیب، جد هر زیر درخت فراموش شده از کیفیت بهترین مسیر موجود در زیر درخت خود اطلاع دارد. بوسیله این اطلاعات، *SMA زمانی که دریافت که تمامی مسیرهای دیگر از مسیر فراموش شده بدتر هستند، دوباره آن را ایجاد خواهد کرد. به عبارت دیگر، اگر تمامی نوادگان گره n فراموش شوند، آنگاه ما نمی‌دانیم چگونه و از چه مسیری از گره n حرکت کنیم، اما ایده‌ای از این که با چه هزینه‌ای می‌توانیم از گره n به گره‌های مختلف برویم، داریم.

اگرچه توضیح کل این الگوریتم در این کتاب مشکل است اما اشارات زیرکانه‌ای به آن می‌کنیم. گفتیم که *SMA بهترین برگ را بسط داده و بدترین برگ را حذف می‌کند. اما اگر همه‌ی گره‌ها دارای مقدار مساوی بودند چه؟ آنگاه ممکن است الگوریتم گره یکسانی را برای بسط و حذف انتخاب کند.*SMA این مشکل را بوسیله بسط بهترین و جدیدترین برگ و حذف قدیمی ترین و بدترین برگ حل کرده‌است. این دو حالت تنها زمانی گره یکسانی را نتیجه می‌دهند که تنها یک برگ وجود داشته باشد، در این حالت درخت جستجوی جاری یک مسیر واحد از ریشه تا برگ است که تمام حافظه را اشغال می‌کند. اگر برگ مورد نظر هدف نباشد، آنگاه حتی اگر در مسیر جواب بهینه مسئله باشد، جواب مورد نظر با فضای حافظه موجود قابل دسترس نخواهد بود. بنابراین، گره مورد نظر می‌تواند کنارگذاشته شود انگار که گره پسینی نداشته است.*SMA کامل است اگر جواب قابل دسترسی وجود داشته باشد -بدین معنی که d (عمق سطحی‌ترین هدف) کمتر از سایز حافظه (در واحد گره) باشد- و بهینه است اگر جواب بهینه قابل دسترسی وجود داشته باشد، در غیر این صورت، بهترین جواب قابل دسترس را بر می‌گرداند. در عمل *SMA بهترین الگوریتم عمومی برای پیدا کردن جواب بهینه

است مخصوصاً زمانی که فضای حالت گراف باشد، هزینه‌ی گام‌ها یکسان نباشد، و تولید گره در مقایسه با سربار اضافی نگهداری لیست باز و بسته پرهزینه باشد.

در مسائل بسیار پیچیده گاهی اوقات اتفاق می‌افتد که *SMA مدام بین یک مجموعه از چند مسیر کاندید رفت و آمد می‌کند زیرا تنها مقدار کوچکی از آن مجموعه در حافظه جا می‌گیرد. (این مشکل شبیه پدیده شکست[۱] در سیستم‌های صفحه‌بندی دیسک[۲] است). بنابراین زمان اضافی برای بسط مجدد گره‌های تکراری موجب می‌شود که مسئله‌ای که در عمل به وسیله‌ی الگوریتم *A قابل حل بوده‌است (به ازای حافظه‌ی بی‌کران) برای *SMA غیر قابل حل باشد. این بدین معنی است که محدودیت‌های حافظه می‌تواند مسئله‌ها را از نظر زمان محاسباتی غیر قابل حل بسازد. اگرچه تئوری خاصی در مورد رابطه‌ی مصالحه[۳] بین زمان و حافظه وجود ندارد، اما به نظر می‌آید که این مشکلی اجتناب ناپذیر است. شاید تنها راه صرف نظر کردن از خاصیت بهینگی است.

۴-۱-۴- یادگیری برای جستجوی بهتر

تا اینجا استراتژی‌های ثابت زیادی را معرفی کردیم - عمق- اول، بهترین- اولین حریصانه و غیره- که توسط دانشمندان کامپیوتر طراحی شده‌است. آیا یک عامل می‌تواند یاد بگیرد که چگونه بهتر جستجو کند؟ پاسخ مثبت است و روش آن متکی به مفهوم مهمی به نام فضای حالت ماوراء[۴] است. حالت‌ها در فضای حالت ماوراء، حالت درونی (محاسباتی) برنامه‌ای را که درون فضای حالت اشیاء[۵] (مثل کشور رومانی) جستجو می‌کند ضبط می‌کنند. به عنوان مثال، حالت درونی الگوریتم *A شامل درخت جستجوی جاری است. هر کنش در فضای حالت ماوراء یک گام محاسباتی است که حالت درونی را تغییر می‌دهد، برای مثال هر گام محاسباتی در *A یک برگ را بسط می‌دهد و گره‌های پسین آن را به درخت اضافه می‌کند. بنابراین شکل ۴-۳ که بزرگ شدن دنباله‌ای از جستجوی‌های درخت را نشان می‌دهد می‌تواند به عنوان یک رسام مسیر در فضای حالت ماوراء باشد که در آن هر حالت در مسیر یک جستجوی فضای حالت اشیاء است.

مسیر شکل ۴-۳ پنج مرحله دارد که شامل یک مرحله (بسط فاگاراس) غیر مفید است. برای مسئله‌های پیچیده‌تر، بسیاری از این گام‌های اشتباه وجود دارد، و الگوریتم یادگیری ماورائی[۶] می‌تواند از این تجربیات یاد گرفته و زیر درخت‌های غیر قابل اطمینان را جستجو نکند. تکنیک‌های این نوع یادگیری در فصل [۲۱] توضیح داده شده‌است. هدف یادگیری به حداقل رساندن هزینه‌ی کل حل مسئله، با توجه به رابطه مصالحه بین هزینه‌ی محاسباتی و هزینه‌ی مسیر است.

۱- Thrashing
۲- Disk Paging Systems
۳- Tradeoff
۴- Meta-Level State Space
۵- Object-Level State Space
۶- Meta-level learning

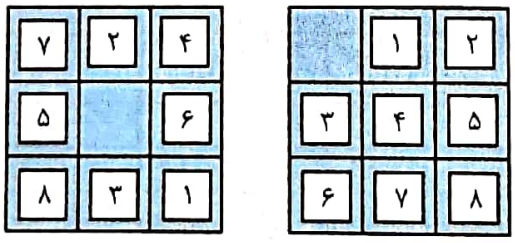
۴-۲- توابع ابتکاری

در این بخش، درباره‌ی توابع ابتکاری برای مسئله ۸-پازل بحث خواهیم کرد تا ماهیت توابع ابتکاری را به طور کلی روشن کنیم.

۸- پازل یکی از اولین مسائل جستجو با توابع ابتکاری بوده‌است. همانطور که در بخش ۳-۲ اشاره شد هدف پازل این است که کاشی‌ها را به گونه‌ای (افقی و عمودی) درون فضای خالی حرکت دهیم که وضعیت پازل مانند وضعیت هدف بشود(شکل ۴-۷).

هزینه‌ی جواب به طور متوسط به ازای یک وضعیت تصادفی ۸-پازل تقریباً ۲۲ گام است. فاکتور انشعاب[1] تقریبا ۳ است(هنگامی که کاشی خالی وسط است ۴ انتخاب برای حرکت وجود دارد، هنگامی که گوشه است ۲ تا و هنگامی که در امتداد ضلع قرار دارد ۳ تا). این بدین معنی است، یک جستجوی کامل تا عمق ۲۲، $3^{22} \sim 3.1 \times 10^{10}$ حالت را می‌یابد. با پیدا کردن حالت‌های تکراری می‌توانیم تقریبا ۱۷۰۰۰۰ حالت را فاکتور گرفته و آن را تقلیل دهیم زیرا تنها ۹!/۲ = ۱۸۱.۴۴۰ حالت متمایز قابل دسترسی وجود دارد (به تمرین ۳-۴ مراجعه کنید). اگرچه این عدد قابل کنترل است اما این عدد برای ۱۵- پازل تقریبا 10^{13} است بنابراین عمل بعدی پیدا کردن یک تابع ابتکاری مناسب است. اگر می‌خواهیم کوتاه‌ترین جواب را با A* بیابیم، به تابع ابتکاری‌ای نیازمندیم که هیچگاه تعداد گام‌های رسیدن به هدف را بیش‌تر از مقدار واقعی تقریب نزند. برای پیدا کردن تابع ابتکاری برای ۱۵-پازل تاریخچه طولانی وجود دارد، که در این جا به دو تا از مشهورترین آنها اشاره خواهیم نمود:

h_1 = تعداد کاشی‌هایی که در جای صحیح خود قرار ندارند. در شکل ۴-۷ همه ۸ کاشی در جای نادرست قرار دارند بنابراین در حالت اولیه داریم $h_1 = 8$. h_1 قابل پذیرش است زیرا واضح است که هر کاشی که در جای اشتباه قرار دارد حداقل یک بار باید جا به جا شود.



شکل ۴-۷ یک نمونه متداول از بازی ۸- پازل. راه‌حل آن ۲۶ مرحله طول می‌کشد.

h_2 = حاصل جمع فاصله کاشی‌ها از مکان مناسب هر یک از آنها. از آنجا که کاشی‌ها مورب حرکت نمی‌کنند، فاصله‌ای که محاسبه می کنیم براساس فاصله افقی و عمودی است که گاهی اوقات فاصله بلاک شهر[2] و فاصله Manhattan نامیده می‌شود. h_2 نیز قابل پذیرش است زیرا تنها کاری که یک جابجایی می‌تواند بکند این است که ما را یک گام به هدف نزدیک‌تر کند. کاشی‌های ۱ تا ۸ در حالت اولیه دارای فاصله Manhattan زیر هستند.

$$h_2 = ۳ + ۱ + ۲ + ۲ + ۲ + ۳ + ۳ + ۲ = ۱۸$$

---

[1]- Branching factor

[2]- City Block Distance

همانطور که انتظار می‌رفت هیچ‌کدام از این ۲ تابع ابتکاری بیش‌تر از مقدار واقعی که ۲۶ است تقریب نزدند.

۴-۲-۱- اثر دقت تابع ابتکاری بر کارایی

فاکتور انشعاب موثر[1]، *b یک راه برای ارزیابی کیفیت تابع ابتکاری است. اگر تعداد کل گره‌های بسط داده شده توسط *A برای یک مسئله بخصوص N باشد و راه حل مسئله در عمق d باشد آنگاه *b فاکتور انشعابی است که یک درخت یکنواخت با عمق d و ۱ + N گره باید داشته باشد. بنابراین

$$N+1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d$$

برای مثال، اگر *A جواب را در عمق ۵ و با بسط ۵۲ گره پیدا کند آنگاه فاکتور انشعاب مؤثر آن برابر ۱.۹۲ خواهد بود. فاکتور انشعاب مؤثر می‌تواند در مسائل مختلف متفاوت باشد، اما اغلب برای مسائلی که تا حدی سخت هستند تا اندازه‌ای ثابت است. بنابراین اندازه گیری *b از طریق آزمایش بر روی یک مجموعه کوچک از مسئله‌ها می‌تواند راهنمای مناسبی برای ارزیابی کلی میزان سودمندی تابع ابتکاری باشد. یک تابع ابتکاری با طراحی مناسب دارای مقدار *b نزدیک به ۱ است که با استفاده از آن می‌توان مسائل نسبتاً بزرگی را حل نمود.

برای ارزیابی توابع ابتکاری h_1 و h_2، ۱۲۰۰ مسئله‌ی تصادفی تولید کردیم که اندازه‌ی راه حل آنها بین ۲ تا ۲۴ است و آنها را با الگوریتم جستجوی عمیق شونده‌ی تکراری و جستجوی درختی *A توسط توابع ابتکاری h_1 و h_2 حل کردیم. شکل ۴-۸ میانگین تعداد گره بسط داده شده و فاکتور انشعاب مؤثر هریک از این استراتژی‌ها را نشان می‌دهد. برآیند آنها نشان می‌دهد که h_2 بهتر از h_1 و بسیار بهتر از الگوریتم جستجوی عمیق شونده‌ی تکراری است. در راه‌حل‌هایی با اندازه ۱۴ *A با h_2 ۳۰۰۰۰ برابر بهینه‌تر از جستجوی الگوریتم عمیق شونده‌ی تکراری ناآگاهانه است.

	Search Cost			Effective Branching Factor		
D	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
۲	۱۰	۶	۶	۲.۴۵	۱.۷۹	۱.۷۹
۴	۱۱۲	۱۳	۱۲	۲.۸۷	۱.۴۸	۱.۴۵
۶	۶۸۰	۲۰	۱۸	۲.۷۳	۱.۳۴	۱.۳۰
۸	۶۳۸۴	۳۹	۲۵	۲.۸۰	۱.۳۳	۱.۲۴
۱۰	۴۷۱۲۷	۹۳	۳۹	۲.۷۹	۱.۳۸	۱.۲۲
۱۲	۳۶۴۴۰۳۵	۲۲۷	۷۳	۲.۷۸	۱.۴۲	۱.۲۴
۱۴	–	۵۳۹	۱۱۳	–	۱.۴۴	۱.۲۳
۱۶	–	۱۳۰۱	۲۱۱	–	۱.۴۵	۱.۲۵
۱۸	–	۳۰۵۶	۳۶۳	–	۱.۴۶	۱.۲۶
۲۰	–	۷۲۷۶	۶۷۶	–	۱.۴۷	۱.۲۷
۲۲	–	۱۸۰۹۴	۱۲۱۹	–	۱.۴۸	۱.۲۸
۲۴	–	۳۹۱۳۵	۱۶۴۱	–	۱.۴۸	۱.۲۶

شکل ۴-۸ مقایسه هزینه جستجو و فاکتور انشعاب موثر برای دو الگوریتم ITERATIVE-DEEPENING-SEARCH و *A با h_1 و h_2. این داده‌ها بر روی ۱۰۰ نمونه مسئله ۸-پازل میانگین گرفته شده است.حال این

[1]- Effective Branching Factor

سؤال مطرح است که آیا h۲ همواره بهتر از h۱ است؟ بله. از روی تعاریف دو تابع ابتکاری براحتی می‌توان دریافت که به ازای هر گره n داریم $h_2(n) \geq h_1(n)$ است. بنابراین می گوییم h۲ بر h۱ برتر[۱] است. برتری[۲] را دقیقاً می‌توان همان کارایی تفسیر کرد: A* با تابع ابتکاری h_2 هیچگاه از A* با تابع ابتکاری h_1 تعداد گره بیش‌تری بسط نخواهد داد. (مگر احتمالاً برای چند گره با ویژگی $f(n) = C^*$). اثبات این ادعا ساده است. مشاهدات بالا را به یاد بیاورید که در آن همه‌ی گره‌ها با ویژگی $f(n) < C^*$ قطعاً بسط داده خواهند شد. این بدین معنی است که بگوییم همه‌ی گره‌ها با ویژگی $h(n) < C^* - g(n)$ قطعاً بسط داده خواهند شد. اما چون به ازای همه‌ی گره‌ها h_2 حداقل برابر h_1 است، پس هر گره‌ای که قطعاً توسط A* با تابع ابتکاری h_2 بسط داده شود، قطعاً با تابع ابتکاری h_1 نیز بسط داده خواهد شد، بعلاوه اینکه h_1 ممکن است گره‌های دیگری را نیز بسط دهد. بنابراین از این پس بهتر است که از تابع ابتکاری استفاده کنیم که دارای مقادیر بزرگتری است البته مشروط بر آنکه تابع ابتکاری از مقدار واقعی بیشتر تقریب نزند و زمان محاسباتی برای محاسبه تابع ابتکاری خیلی طولانی نباشد.

۴-۲-۲- ابداع تابع ابتکاری قابل پذیرش

دیدیم که هر دو تابع ابتکاری h_1 (تعداد کاشی‌های در مکان نادرست) و h_2 (فاصله Manhattan) توابع ابتکاری نسبتاً مناسبی برای ۸- پازل بودند(h_2 مناسب تر است). اما چگونه می‌توان تابع ابتکاری مثل h_2 را ابداع کرد؟ آیا برای کامپیوتر امکان‌پذیر است که چنین تابع ابتکاری را به صورت مکانیکی ابداع کند؟

h_1 و h_2 تقریب و تخمینی از طول بقیه مسیر در ۸- پازل و اندازه دقیقی از مسیر برای نسخه‌ی ساده‌شده‌ی پازل هستند. اگر قوانین پازل به این شکل تغییر کند که هر کاشی به جای اینکه تنها به خانه‌های خالی همسایه برود می‌تواند به هر خانه دلخواه تغییر مکان دهد، آنگاه h_1 دقیقاً تعداد گام‌های لازم برای رسیدن به کوتاه‌ترین پاسخ مسئله را بدست می‌آورد. به طور مشابه،اگر کاشی‌ها بتوانند یک قدم در همه‌ی جهت‌ها حرکت کنند حتی اگر آن خانه پر باشد، آنگاه h_2 تعداد دقیق گام‌ها را برای رسیدن به کوتاه‌ترین مسیر را بدست می‌آورد. به مسئله‌ای که در آن محدودیت‌های کمتری روی کنش‌ها اعمال می‌شود، مسئله‌ی ساده شده[۳] گویند. هزینه‌ی جواب بهینه‌ی یک مسئله ساده شده، یک تابع ابتکاری قابل پذیرش برای مسئله‌ی اصلی محسوب می‌شود. این تابع ابتکاری قابل پذیرش است زیرا جواب بهینه در مسئله‌ی اصلی طبق تعریف، پاسخی برای مسئله‌ی ساده شده نیز محسوب می‌شود که هزینه‌ی آن حداقل برابر جواب بهینه‌ی مسئله، در مسئله‌ی ساده شده است. از آنجایی که تابع ابتکاری بدست آمده هزینه‌ی دقیق برای مسئله‌ی ساده شده است، بنابراین باید از نامساوی مثلثی تبعیت کند و بنابراین سازگار است.

اگر صورت مسئله‌ای به صورت رسمی نوشته شده‌باشد این امکان وجود دارد که به صورت اتوماتیک از روی آن مسئله‌ای ساده شده بسازیم. به عنوان مثال اگر کنش‌ها در ۸-پازل این گونه توصیف شوند:

۱- Dominate
۲- Domination
۳- Relaxed Problem

یک کاشی می‌تواند از خانه‌ی A به خانه‌ی B حرکت کند اگر A به صورت افقی یا عمودی همسایه‌ی B بوده و خالی باشد.

آنگاه می‌توانیم با حذف یک یا هر ۲ شرط آن، مسئله‌ی ساده شده بسازیم:

الف) کاشی‌ها می‌توانند از خانه‌ی A به خانه‌ی B بروند اگر A و B همسایه باشند.

ب) کاشی‌ها می‌توانند از خانه‌ی A به خانه‌ی B بروند اگر B خالی باشد.

ج) کاشی‌ها می‌توانند از خانه‌ی A به خانه‌ی B بروند.

از الف می توان h_2 (فاصله Manhattan) را نتیجه گرفت. دلیل این مسئله این است که اگر هر کدام از کاشی‌ها به نوبت در جای مناسب خود قرار بگیرند آنگاه h_2 محاسبه درستی خواهد بود. تابع ابتکاری برگرفته شده از ب در تمرین شماره ۴-۹ بحث شده است. از ج نیز می‌توانیم h_1 (تعداد کاشی‌های در مکان نادرست)را نتیجه‌گیری کنیم،زیرا اگر کاشی‌ها با یک حرکت در مکان درست خود قرار بگیرند آنگاه h_1 محاسبه درستی خواهد بود. به این نکته پراهمیت توجه داشته باشید که مسائل ساده شده‌ای که با این تکنیک بدست آمده‌اند بدون جستجو نیز قابل حل هستند، زیرا قوانین ساده شده این امکان را فراهم می‌آورند که مسئله به هشت زیر مسئله‌ی مستقل تجزیه شود. اگر مسئله‌ی ساده شده راه‌حل مشکلی داشته باشد، آنگاه بدست آوردن مقادیر تابع ابتکاری متناظر با آن پرهزینه خواهند بود.

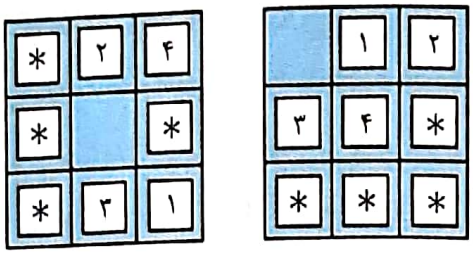
برنامه‌ای به نام ABSOLVER به صورت اتوماتیک و با استفاده از تکنیک "مسئله ساده شده" و تکنیک‌های متنوع دیگر از روی صورت مسئله، تابع ابتکاری ابداع می کند. ABSOLVER بهترین و جدیدترین تابع ابتکاری موجود برای ۸- پازل را تولید و اولین تابع ابتکاری مؤثر برای مسئله‌ی مشهور پازل مکعبی Rubik را کشف کرده‌است. مشکل تولید توابع ابتکاری جدید این است که گاهی اوقات ممکن است بدست آوردن "دقیقاً بهترین" تابع ابتکاری به شکست منجر شود. اگر مجموعه‌ای از توابع ابتکاری سازگار h۱....hm برای یک مسئله‌ی به‌خصوص موجود باشد و هیچ کدام بر دیگری برتری نداشته باشند، کدام‌یک را باید انتخاب کنیم؟این طور به نظر می‌رسد که نیازی نیست لزوماً یکی را انتخاب کنیم بلکه می‌توانیم با تعریف زیر، بهترین‌های جهان را داشته باشیم:

$$h(n) = \max \{h_1(n),\ldots,h_m(n)\}$$

این تابع ابتکاری مختلط، از تابعی که جواب دقیق‌تری به ازای آن گره در مسئله می دهد استفاده می‌کند. از آنجا که همه‌ی توابع ابتکاری تشکیل دهنده‌ی h قابل قبول هستند، h نیز قابل قبول است. همچنین بدیهی است که h سازگار است. بعلاوه h بر همه‌ی توابع ابتکاری تشکیل دهنده‌ی آن برتری دارد.

همچنین توابع ابتکاری قابل پذیرش را می‌توان از طریق محاسبه هزینه جواب زیر مسئله‌ی یک مسئله داده شده بدست آورد. برای مثال شکل ۴-۹ یک زیر مسئله از مسئله ۸- پازل شکل ۴-۷ را نشان می‌دهد. زیر مسئله شامل قرار دادن کاشی‌های ۱ و ۲ و ۳ و ۴ در مکان مناسب خود است. بدیهی است که هزینه‌ی جواب بهینه این زیر مسئله حد پایینی برای جواب مسئله‌ی اصلی است. نتیجه اینکه گاهی اوقات این جواب بسیار دقیق‌تر از فاصله Manhattan عمل می‌کند.

ایده‌ی "الگوی پایگاه داده[۱]"، ذخیره‌ی هزینه‌ی دقیق پاسخ به همه‌ی زیر مسئله‌های ممکن (در مثال بالا همه‌ی جایگشت‌های ممکن ۴ کاشی و جاهای خالی) برای یک مسئله است. توجه داشته باشید که حرکت ۴ کاشی دیگر (خالی) بی‌ارتباط با حل زیرمسئله است اما حرکت آن کاشی‌ها نیز جزء هزینه‌ی مسئله به حساب می‌آید. آنگاه به ازای هر حالت حقیقی که طی جستجو با آن برخورد می‌کنیم، یک تابع ابتکاری قابل پذیرش به نام h_{DB} را با استفاده از پیدا کردن زیر مسئله متناظر با آن، در پایگاه داده می‌سازیم. پایگاه داده، خود از جستجوی وارونه از هدف به عقب و ثبت هزینه‌ی الگوهای جدیدی که با آنها برخورد می‌کند به وجود می‌آید.



شکل ۴-۹ یک زیرمسئله از مسئله ۸-پازل داده شده در شکل ۴-۷. وظیفه ما این است که کاشی‌های ۱، ۲، ۳ و ۴ را در مکان‌های صحیح خود قرار دهیم بدون اینکه درباره کاشی‌های دیگر تصمیم‌گیری کنیم.

انتخاب کاشی‌های ۱-۲-۳-۴ تا اندازه‌ای اختیاری است، می‌توانیم برای ۵-۶-۷-۸ و ۲-۴-۶-۸ و غیره نیز پایگاه داده درست کنیم. هر پایگاه داده یک تابع ابتکاری قابل پذیرش تولید می‌کند و این توابع ابتکاری همانطور که پیش‌تر گفته شد با ماکزیمم‌گیری از مقادیر آنها می‌توانند ترکیب شوند. توابع ابتکاری از این نوع بسیار دقیق‌تر از تابع ابتکاری فاصله Manhattan است، تا حدی که تعداد گره‌های ایجاد شده برای حل یک حالت تصادفی از مسئله ۱۵- پازل می‌تواند ۱۰۰۰ برابر کاهش یابد.

شاید این تصور پیش بیاید که می‌توان ۲ تابع ابتکاری بدست آمده از پایگاه داده‌های ۱-۲-۳-۴ و ۵-۶-۷-۸ را به این دلیل که به نظر همپوشانی ندارند با هم جمع کرد. آیا تابع ابتکاری بدست آمده همچنان قابل پذیرش است؟ خیر، زیرا پاسخ زیر مسئله‌ی بدست آمده از ۱-۲-۳-۴ و ۵-۶-۷-۸ به ازای یک حالت خاص در بعضی حرکت‌ها اشتراک دارند – انتقال کاشی‌های ۱-۲-۳-۴ به خانه‌ی مورد نظر بدون تماس با کاشی‌های ۵-۶-۷-۸ غیر ممکن است و برعکس –اما اگر آن حرکت‌ها را نشماریم چطور؟ (حرکت جاهای خالی در زیرمسئله) این بدین معنی است که ما هزینه‌ی واقعی پاسخ زیر مسئله‌ی ۱-۲-۳-۴ را ثبت نکرده بلکه تنها حرکت‌های شامل کاشی‌های ۱-۲-۳-۴ را ثبت می‌کنیم. در این صورت بدیهی است که حاصل‌جمع این ۲ هزینه یک کران پایین برای پاسخ مسئله‌ی اصلی محسوب می‌شود. این ایده مربوط به "الگوی پایگاه داده‌های مستقل[۲]" است. استفاده از چنین پایگاه داده‌هایی حل یک حالت تصادفی از مسئله ۱۵-پازل را در چند میلی ثانیه ممکن می‌سازد و تعداد گره‌های ایجاد شده در مقایسه با تابع ابتکاری فاصله Manhattan را ۱۰۰۰۰ برابر کاهش می‌دهد. برای ۲۴-پازل این افزایش سرعت به طور تقریبی به یک میلیون بار می‌رسد. الگوی پایگاه داده‌های مستقل برای حرکت کاشی‌ها در پازل مؤثر است زیرا مسئله می‌تواند به گونه‌ای تقسیم شود که هر حرکت تنها یک زیر مسئله را تحت تأثیر قرار

۱- Pattern Database

۲-Disjoint Pattern Database

دهد. برای مسئله‌ای مثل مکعب Rubik این نوع تقسیم مسئله ممکن نیست زیرا هر حرکت ۸ یا ۹ مکعب از ۲۶ مکعب را تحت تأثیر قرار می‌دهد. هنوز مشخص نشده‌است که چگونه برای این نوع مسائل می توان از ایده‌ی پایگاه داده‌ای مستقل استفاده کرد.

۴-۲-۳- ساخت توابع ابتکاری یادگیرنده بر اساس تجربیات

وظیفه‌ی تابع ابتکاری h(n) این است که هزینه‌ی رسیدن به پاسخ مسئله را از گره n تخمین بزند. یک عامل چگونه می‌تواند چنین تابعی را ابداع کند؟ در بخش قبل به یک راه‌حل آن اشاره شد – بدین صورت که تابع ساده شده‌ای تعریف کنیم که براحتی بتوان جواب بهینه‌ی آن را بدست آورد. راه دیگر این است که از تجربه‌ها بیاموزیم. برای مثال "تجربه" به معنی حل تعداد زیادی مسئله ۸- پازل است. هر جواب بهینه برای مسئله‌ی ۸-پازل مثال‌هایی را فراهم می‌آورد که می‌توان از آن‌ها h(n) را آموخت. این مثال‌ها شامل حالتی از مسیر و هزینه‌ی واقعی پاسخ مسئله از نقطه‌ی مورد نظر است. از این مثال‌ها و با استفاده از الگوریتم یادگیری استقرایی[۱] می‌توان تابع ابتکاری h(n)ای را ابداع کرد که بتواند (با خوش شانسی) هزینه‌ی رسیدن به پاسخ مسئله را برای حالت‌های دیگری که در طول جستجو به آن‌ها بر می‌خورد پیش‌بینی کند که تیکنیک‌های لازم برای انجام آن با استفاده از شبکه‌های عصبی، درخت‌های تصمیم‌گیری انجام می‌شود.

روش‌های یادگیری استقرایی هنگامی بهترین عملکرد را دارند که از ویژگی‌هایی از یک حالت استفاده کنند که علاوه بر تعریف سطحی خود حالت مرتبط با ارزیابی آن (حالت) نیز باشند. به عنوان مثال، "تعداد کاشی‌های در مکان نادرست" ممکن است برای پیش‌بینی فاصله‌ی واقعی یک حالت از هدف مفید باشد. بگذارید نام این ویژگی را $x_1(n)$ بنامیم، می‌توانیم ۱۰۰ مسئله ۸-پازل با ترتیب تصادفی تولید و بر روی هزینه‌ی واقعی پاسخ مسئله سرشماری انجام دهیم. ممکن است به این نتیجه برسیم که هنگامی که $x_1(n)$ برابر ۵ است آنگاه میانگین هزینه‌ی پاسخ مسئله برابر ۱۴ است. با داشتن این ارقام می‌توانیم با استفاده از x_1 مقدار h(n) را پیش‌بینی کنیم علاوه بر این می‌توانیم از چند ویژگی به صورت هم‌زمان استفاده کنیم. ویژگی دوم $x_2(n)$ میتواند "تعداد کاشی‌های همسایه‌ای که در حالت هدف نیز همسایه هستند" باشد. چگونه می‌توان از ترکیب x_1 و x_2، h(n) را پیش‌بینی کرد؟ یک راه رایج استفاده از ترکیب خطی آن‌هاست:

$$h(n) = c_1x_1(n) + c_2x_2(n)$$

مقادیر ثابت c_1 و c_2 طوری تنظیم می‌شوند که عددی بسیار نزدیک به مقدار هزینه‌ی واقعی پاسخ مسئله را بدهد. احتمالاً C_1 مثبت و C_2 منفی می‌باشد.

۴-۳- الگوریتم‌های جستجوی محلی[۲] و مسائل بهینه‌سازی

الگوریتم‌های جستجویی که تا به حال دیدیم برای گشتن در فضاهای جستجو به صورت سیستماتیک طراحی شده‌اند. این سیستمی بودن از طریق نگهداری یک یا بیش از یک مسیر در حافظه و ثبت گزینه‌های جستجو شده در هر نقطه طی مسیر بدست آمده است. علاوه بر این هنگامی که هدف پیدا شد، مسیر رفتن به آن هدف خود پاسخی برای مسئله است.

۱- Inductive Learning
۲- Local Search

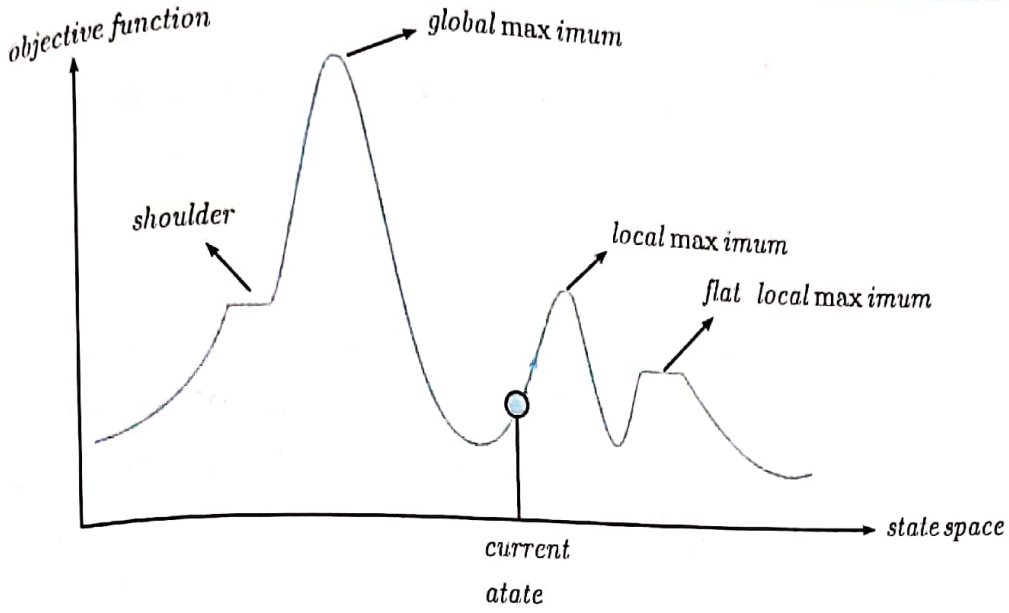
در بسیاری از مسائل مسیر رسیدن به جواب بی‌اهمیت است. به عنوان مثال، در مسئله ۸-وزیر (به صفحه مراجعه کنید) ترتیب اضافه شدن وزیرها مهم نیست بلکه ترتیب نهایی وزیرها حائز اهمیت است. این طبقه از مسائل شامل کاربردهای بسیار مهمی مانند طراحی مدارهای مجتمع، طرح کارخانه، زمانبندی کارها، برنامه‌نویسی اتوماتیک، بهینه سازی شبکه ارتباطات از راه دور، مسیریاب خودرو و مدیریت اسناد است.

اگر مسیر رسیدن به هدف مهم نیست، بنابراین شاید بهتر باشد طبقه‌ی دیگری از الگوریتم‌ها را مورد مطالعه قرار دهیم. الگوریتم‌هایی که در آنها اصلاً نگران مسیرها نخواهیم بود. الگوریتم‌های جستجوی محلی بر روی یک حالت جاری (به جای چند مسیر) عمل می‌کنند و معمولاً تنها به حالت‌های همسایه رفته و مسیرهای جستجو شده را نگهداری نمی‌کنند. اگرچه الگوریتم‌های جستجوی محلی سیستماتیک نیستند اما دو مزیت کلیدی دارند: ۱- از حافظه بسیار اندکی استفاده می کنند (اغلب یک مقدار ثابت) و ۲- در فضاهای حالت بزرگ یا نامتناهی (پیوسته) که در آنها استفاده از الگوریتم‌های سیستماتیک نامناسب است جواب‌های قابل قبولی را بدست می‌آورند.

علاوه بر پیدا کردن هدف، الگوریتم‌های جستجوی محلی برای حل مسائل بهینه سازی[1] محض که در آنها هدف پیدا کردن بهترین حالت بر اساس تابع هدف[2] است مفید می‌باشند.

بسیاری از مسائل بهینه‌سازی در طبقه جستجوی‌های استاندارد معرفی شده در فصل ۳ گنجانده نمی‌شوند. به عنوان مثال در طبیعت، یک تابع هدف (تطبیق تناسلی) تولید کرده که تکامل Darwinian تلاش برای بهینه سازی آن دارد حال آنکه هیچ "تست هدف" و "هزینه مسیری" در این مسئله وجود ندارد.

برای درک جستجوی محلی لازم است که به دورنمای فضای حالت (شکل ۴-۱۰) توجه کنید. یک دورنما هم دارای "مکان" (تعریف شده با حالت) و هم "ارتفاع" (تعریف شده با مقدار هزینه تابع ابتکاری یا تابع هدف). اگر ارتفاع معادل هزینه باشد، آنگاه هدف پیدا کردن پست ترین دره(مینیمم کلی) است و یا اگر ارتفاع معادل تابع هدف باشد آنگاه هدف پیدا کردن بلندترین قله است (می‌توانید این دو را با ضرب در منفی به یکدیگر تبدیل کنید). الگوریتم‌های جستجوی محلی این فضای حالت جستجو می‌کنند. یک جستجوی محلی کامل همواره هدف را (در صورت وجود) پیدا می‌کند، و یک الگوریتم بهینه همواره مینیمم/ ماکزیمم کلی را پیدا می‌کند.



شکل ۴-۱۰ یک دورنما یک- بعدی از جستجوی حالت که در آن ارتفاع متناظر با تابع هدف است. هدف پیدا کردن ماکزیمم کلی است. جستجوی تپه‌نوردی حالت فعلی را تغییر می‌دهد تا آن را ارتقاء دهد(همانطور که با پیکان نمایش داده شده است).

۴-۳-۱- جستجوی تپه‌نوردی

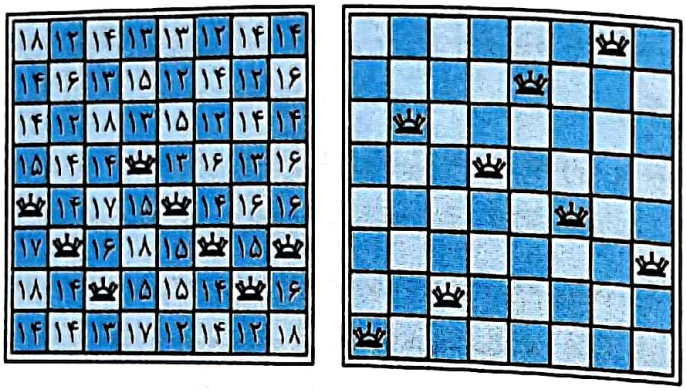
الگوریتم جستجوی تپه‌نوردی در شکل ۴-۱۱ نشان داده شده‌است. این الگوریتم یک حلقه ساده است که دائماً در جهت افزایش ارزش (صعود به بالای تپه) حرکت می‌کند و زمانی که به قله‌ای می‌رسد که در مجاورت آن مقدار بالاتری وجود ندارد خاتمه می‌یابد. این الگوریتم درخت جستجو را در خود نگه نمی‌دارد بلکه ساختمان داده گره جاری، تنها حالت و مقدار تابع هدف را ثبت می‌کند. الگوریتم تپه‌نوردی همسایه‌های متصل به حالت جاری را بررسی نمی‌کند. دقیقاً مثل این که در مه غلیظ در حالی که به فراموشی دچار شده‌ایم به دنبال قله‌ی کوه اورست بگردیم.

برای نشان دادن الگوریتم تپه نوردی از مسئله ۸-وزیر استفاده می‌کنیم. الگوریتم‌های جستجوی محلی معمولاً از فرموله‌کردن کامل حالات استفاده می‌کنند که در آن‌ها هر حالت نشان دهنده ۸ وزیر روی صفحه شطرنج (هر وزیر روی یک ستون) است. تابع گره پسین تمام حالت‌هایی را برمی‌گرداند که ممکن است با حرکت یک وزیر به خانه‌ای دیگر (در همان ستون) ایجاد شود. (بنابراین هر حالت ۸×۷ = ۵۶ گره پسین دارد). تابع ابتکاری h، تعداد جفت وزیرهایی است که به هم دیگر حمله می‌کنند، چه به صورت مستقیم چه غیر مستقیم. مینیمم کلی این تابع صفر است که تنها هنگام حل کامل مسئله رخ می‌دهد. شکل ۴-۱۲ (a) حالتی را با h = ۱۷ نشان می‌دهد. این شکل همچنین مقدار تمامی حالت‌های پسین را نشان می‌دهد که در آن بهترین حالت برابر h = ۱۲ است. الگوریتم تپه‌نوردی عموماً به طور تصادفی از میان مجموعه بهترین حالت‌های پسین (در صورتی که بیش از یکی باشد) یکی را انتخاب می‌کند.

```
function Hill-Climbing (problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← Make-Node(Initial-State[problem])
    loop do
        neighbor ← a highest- valued successor of current
        if Value[neighbor] ≤ Value[current] then return State[current]
        current ← neighbor
```

شکل ۴-۱۱ الگوریتم جستجوی تپه‌نوردی(نسخه انتخاب پرشیب‌ترین حرکت) که تکنیکی پایه در جستجوی محلی محسوب می‌شود. در هر مرحله گره جاری با بهترین همسایه جایگزین می‌شود،(در این نسخه منظور همسایه‌ای با بیش‌ترین مقدار است، البته اگر از ابتکار هزینه تقریبی استفاده شده باشد آنگاه منظور یافتن همسایه‌ای با کمترین h است).



شکل ۴-۱۲ (a) یک حالت از مسئله ۸- وزیر با ابتکار هزینه تقریبی h=۱۷. مقدار h برای هریک از حالت‌های پسین که از حرکت یک وزیر در ستون خود ایجاد می‌شود نشان داده شده است. بهترین حرکت‌ها مشخص شده‌اند. (b) یک مینیمم محلی در فضای جستجوی ۸-وزیر (این حالت دارای h=۱ است ولی بقیه حالت‌ها پسین همگی دارای مقدار بالاتری هستند).

به الگوریتم تپه‌نوردی گاهی اوقات جستجوی محلی حریصانه گفته می‌شود زیرا بسرعت یک حالت مناسب از همسایه را انتخاب می‌کند بدون اینکه فکر کند در آینده می‌خواهد به کجا برود. اگرچه حرص یکی از هفت گناه مرگ بار محسوب می‌شود، اما الگوریتم‌های حریصانه اغلب اوقات بسیار خوب عمل می‌کنند. از آنجا که الگوریتم تپه‌نوردی معمولاً قادر است بسیار راحت یک حالت بد را بهبود ببخشد، اغلب بسیار سریع به سمت پاسخ مسئله حرکت می‌کند. به عنوان مثال، از حالت شکل ۴-۱۲ (a) فقط با پنج مرحله به حالت شکل ۴-۱۲ (b) که دارای h =۱ و بسیار نزدیک به جواب مسئله است رسیدیم. متأسفانه، الگوریتم تپه‌نوردی گاهی به دلایل زیر دچار مشکل می شود:

ماکزیمم محلی: ماکزیمم محلی قله‌ای است که از تمامی حالت‌های همسایه‌اش بلندتر، اما از ماکزیمم کلی کوتاه‌تر است. الگوریتم‌های تپه‌نوردی که به همسایگی ماکزیمم محلی می‌رسند به سمت قله‌ی بالا کشیده می‌شوند اما در آنجا گیر کرده و دیگر نمی‌توانند جایی بروند. شکل ۴-۱۰ این مسئله را به صورت طرحوار نشان

می‌دهد. عیناً حالت نشان داده شده در شکل ۴-۱۲(b) یک ماکزیمم محلی است زیرا حرکت هر کدام از وزیرها وضعیت را بدتر می‌کند (یک مینیمم محلی برای h).

کوهان[1]: در شکل ۴-۱۳ کوهان نشان داده شده‌است. کوهان‌ها از دنباله‌ای از ماکزیمم‌های محلی به وجود می‌آیند که مسیریابی آن برای الگوریتم‌های حریصانه بسیار مشکل است.

فلات[2]: ناحیه‌ای از فضای حالت دورنما که در آن تابع ارزیابی ثابت است را فلات گویند. فلات می‌تواند یک ماکزیمم محلی مسطح باشد که هیچ سربالایی وبرآمدگی‌ای برای خروج و ادامه راه وجود ندارد (شکل ۴-۱۰ را مشاهده کنید). الگوریتم جستجوی تپه‌نوردی ممکن است نتواند از فلات خارج شود.

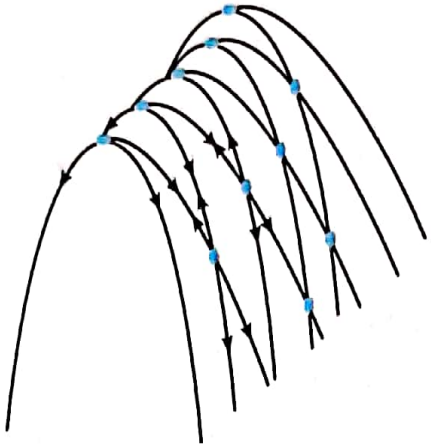
در هر صورت، الگوریتم به نقطه‌ای می‌رسد که در آن هیچ عملی انجام نمی‌شود. در ۸- وزیر با شروع از یک حالت تصادفی، پر شیب‌ترین صعودهای الگوریتم تپه‌نوردی در ۸۶٪ موارد به بن‌بست خورده و تنها در ۱۴٪ موارد موفق شده‌اند. این الگوریتم بسیار سریع عمل می‌کند، به طور میانگین در صورت موفقیت با چهار مرحله و در صورت شکست با سه مرحله خاتمه می‌یابد که برای فضای حالتی با $8^8 = 17$ میلیون حالت نتیجه‌ی بدی محسوب نمی‌شود.

این الگوریتم که در شکل ۴-۱۱ نشان داده شده‌است. درصورتی که به فلاتی برسد که در آن بهترین گره پسین مقداری برابر گره جاری دارد متوقف می‌شود. اما آیا بهتر نیست الگوریتم را ادامه دهیم؟ (یعنی اجازه دهیم الگوریتم حرکت یکسو به جلو[3] داشته باشد به این امید که شاید فلات درآینده برآمدگی داشته باشد (مانند شکل ۴-۱۰)). جواب در اغلب موارد مثبت است، اما باید مواظب باشیم. اگر ما همواره به الگوریتم اجازه حرکت یکسو به جلو را بدهیم در صورتی که هیچ سر بالایی وجود نداشته باشد، آنگاه الگوریتم زمانی که به یک ماکزیمم محلی مسطحی که برآمدگی ندارد برمی‌خورد در یک حلقه ناتمام گیر می‌کند. یک راه‌حل متداول برای این مسئله گذاشتن محدودیت برروی تعداد حرکت‌های یکسو به جلو است. به عنوان مثال، برای مسأله‌ی ۸-وزیر می‌توانیم (به فرض) تا سقف ۱۰۰ حرکت یکسو به جلو پی در پی را اجازه دهیم. این عمل درصد مسائل حل شده توسط الگوریتم تپه‌نوردی را از ۱۴٪ به ۹۴٪ افزایش می‌دهد. اگرچه هر موفقیتی هزینه بر است: در این حالت به طور میانگین الگوریتم در صورت موفقیت ۲۱ مرحله و درصورت شکست ۶۴ مرحله را طی می‌کند.

الگوریتم‌های گوناگونی از تپه‌نوردی ابداع شده‌است. الگوریتم تپه‌نوردی تصادفی[4] از بین حرکت‌های صعودی به طور تصادفی یکی را انتخاب می‌کند (احتمال انتخاب هر حرکت به تندی شیب آن بستگی دارد). این الگوریتم در بسیاری از موارد کندتر از انتخاب پرشیب‌ترین صعود عمل می‌کند اما در بعضی از فضاهای حالت جواب‌های بهتری را می‌یابد. الگوریتم تپه‌نوردی اولین-انتخاب، الگوریتم تپه‌نوردی تصادفی را به گونه‌ای پیاده سازی می‌کند که تا زمانی که اولین گره بهتر از گره جاری پیدا شود به صورت تصادفی گره پسین تولید می‌کند. این استراتژی برای زمان‌هایی مناسب است که حالت‌ها دارای تعداد زیادی (هزاران) گره پسین هستند. تمرین شماره ۴-۱۶ در این زمینه از شما سؤال می‌کند.

---

[1]- Ridge
[2]- Plateaux
[3]- Sideways move
[4]- Stochastic hill climbing



شکل ۴-۱۳ این شکل نشان می‌دهد که چرا کوهان‌ها برای الگوریتم تپه‌نوردی مشکل ایجاد می‌کنند. شبکه‌ای از حالات (دایره‌های سیاه) برروی کوهان‌ها سوار شده‌اند(از سمت چپ به سمت راست) که دنباله‌ای از ماکزیمم‌های محلی را می‌سازند که مستقیما با یکدیگر مرتبط نیستند. از همه‌ی ماکزیمم‌های محلی، همه کنش‌های موجود به سمت پایین اشاره می‌کنند.

الگوریتم‌های تپه‌نوردی که تاکنون شرح داده شده‌اند کامل نیستند زیرا اغلب در اثر گیر افتادن در ماکزیمم محلی به هدف نمی‌رسند.

الگوریتم تپه‌نوردی شروع مجدد تصادفی از ضرب‌المثل " اگر بار اول موفق نشدی دوباره تلاش کن" اقتباس شده است. این الگوریتم مجموعه‌ای از الگوریتم‌های جستجوی تپه‌نوردی که به صورت تصادفی از یک حالت اولیه تولید می‌شوند را اجرا می‌کند و هنگام پیدا کردن هدف خاتمه می‌یابد. این الگوریتم با احتمال نزدیک به ۱ کامل است زیرا بدیهی است که الگوریتم بالاخره گره هدف را به عنوان حالت اولیه تولید می‌کند. اگر جستجوی تپه نوردی با احتمال p موفق شود آنگاه تعداد شروع‌های مجدد مورد انتظار برابر $1/p$ خواهد بود. برای مسئله‌ی ۸-وزیر فاقد حرکت یکسو به جلو $p \sim 0/14$ است بنابراین ما تقریباً نیاز به ۷ تکرار برای پیدا کردن هدف داریم (۶ شکست و ۱ موفقیت). تعداد مراحل مورد انتظار، برابر هزینه‌ی یک بار تکرار موفق بعلاوه $1-p/p$ برابر هزینه شکست (تقریبا ۲۲ مرحله) است. هنگامی که اجازه‌ی حرکت یکسو به جلو می‌دهیم، به طور میانگین $\frac{1}{0/94} \sim 1/06$ تکرار و $25 \sim 64 \times \left(\frac{0/06}{0/94}\right) + (1 \times 21)$ مرحله مورد نیاز است. الگوریتم تپه‌نوردی شروع مجدد تصادفی برای مسئله‌ی ۸- وزیر کاراتر است. حتی برای سه میلیون وزیر، این روش می‌تواند جواب مسئله را زیر یک دقیقه بیابد.

موفقیت الگوریتم تپه‌نوردی شدیداً به شکل فضای حالت بستگی دارد: اگر تعداد ماکزیمم‌های محلی و فلات کم باشد آنگاه الگوریتم تپه‌نوردی شروع مجدد تصادفی بسرعت پاسخ مناسبی برای مسئله بدست می‌آورد. از سوی دیگر بسیاری از مسائل در دنیای واقعیت فضای حالتی شبیه به جوجه تیغی‌هایی بر روی سطح صاف دارند که بر روی نوک هر کدام از تیغ‌های آنان جوجه تیغی کوچک دیگری (و همین طور در مورد جوجه تیغی‌های کوچک‌تر) زندگی می‌کند. در مسائل NP- سخت معمولاً به تعداد نمایی ماکزیمم محلی وجود دارد که باعث بن‌بست می‌شوند. با این وجود بعد از چند شروع مجدد می‌توان ماکزیمم محلی مناسبی را یافت.

۴-۳-۲- جستجوی شبیه‌سازی ذوب فلزات[۱] (شبیه‌سازی گداختگی)

الگوریتم‌های تپه‌نوردی‌ای که هیچگاه به سوی حالت‌هایی با مقدار کمتر (هزینه بیش‌تر) حرکت "نزولی" نمی‌کنند غیر کامل هستند، زیرا این احتمال وجود دارد که در ماکزیمم محلی گیر کنند. در مقابل، گام برداشتن به شکل کاملاً تصادفی (به معنی حرکت به سمت گره پسین انتخاب شده تصادفی از میان مجموعه گره‌های پسین با احتمال یکسان) کامل است اما بسیار غیر کاراست. بنابراین منطقی به نظر می‌رسد که تلاش کنیم دو ایده‌ی الگوریتم تپه‌نوردی و گام‌های تصادفی را به شکلی ترکیب کرده که هم کارایی و هم کامل بودن را بدست آوریم. الگوریتم شبیه سازی ذوب فلزات چنین الگوریتمی است. در متالوژی گداختن فرایندی است که از آن برای خمیر کردن و سخت کردن فلزات و شیشه‌ها بوسیله حرارت در دمای بالا و بتدریج سرد کردن آنان به منظور به هم آمیخته شدن مواد به یک حالت کریستال کم انرژی استفاده می‌شود. برای درک الگوریتم شبیه سازی گداختگی بگذارید به جای تپه‌نوردی از دید شیب نزولی (کاهش هزینه) به مسأله نگاه کنیم، حال تصور کنید که می‌خواهید یک توپ پینگ-پونگ را به درون عمیق‌ترین حفره بر روی یک سطح ناصاف بیاندازید. اگر توپ را بغلتانیم، در اولین مینیمم محلی قرار می‌گیرد. اگر سطح را تکان دهیم آنگاه می‌توانیم توپ را از روی مینیمم محلی جهش دهیم. ایده این است که به اندازه‌ای تکان دهیم که توپ از روی مینیمم محلی جهش کند و از مینیمم کلی گذر نکند. راه حل الگوریتم شبیه سازی گداختگی این است که ابتدا با تکان‌های شدید شروع کرده (در دمای بالا) و بتدریج از شدت تکان‌ها بکاهیم (در دماهای پایین‌تر). درونی‌ترین حلقه الگوریتم شبیه سازی گداختگی (شکل ۴-۱۴) بسیار شبیه الگوریتم تپه‌نوردی است. به جای انتخاب بهترین حرکت، به طور تصادفی حرکت‌ها را انتخاب می‌کند. اگر آن حرکت شرایط را ارتقاء دهد، همواره انتخاب می‌شود، در غیر این صورت الگوریتم با احتمال کمتر از یک آن را می‌پذیرد. این احتمال به صورت نمایی به نسبت "نامناسب بودن" حرکت کاهش می‌یابد (مقدار ΔE با بدتر شدن مقدار ارزیابی کاهش می‌یابد) این احتمال همچنین با کاهش دما T نیز کم می‌شود: احتمال اجازه دادن برای حرکت‌های نامناسب در آغاز که دما بیش‌تر است زیادتر است، و با کاهش T کم می‌شود. می‌توان اثبات کرد که اگر جدول زمانی که T را کم می‌کند به اندازه کافی کند باشد آنگاه این الگوریتم با احتمال نزدیک به یک مینیمم کلی راپیدا خواهد کرد.

الگوریتم شبیه سازی گداختگی ابتدا به طور گسترده برای حل مسائل طراحی VLSI در اوایل ۱۹۸۰ مورد استفاده قرار می‌گرفت. این الگوریتم به صورت گسترده در زمان‌بندی کارخانه‌ها و کارهای بهینه سازی در ابعاد بزرگ به کار گرفته شده‌است. در تمرین شماره‌ی ۴-۱۶از شما درباره مقایسه‌ی کارایی این الگوریتم با الگوریتم تپه‌نوردی شروع مجدد تصادفی در مسئله n- وزیر سؤال شده‌است.

۴-۳-۳- جستجوی پرتوی محلی

نگه داشتن تنها یک گره در حافظه به نظر یک نوع عکس‌العمل افراطی نسبت به مسئله‌ی کمبود حافظه است. الگوریتم جستجوی پرتوی محلی[۲] به جای یک حالت k حالت را نگهداری می‌کند. این الگوریتم ابتدا k حالت را به صورت تصادفی تولید کرده و در هر مرحله همه‌ی گره‌های پسین هر k حالت تولید می‌شوند. اگر یکی از آنها

۱- Simulated annealing Seurch
۲- Local Beam Search

هدف باشد آنگاه الگوریتم خاتمه می‌یابد در غیر این صورت k تا از بهترین گره‌های پسین را از همه گره‌های پسین موجود انتخاب و الگوریتم ادامه می‌یابد.

در نگاه اول، الگوریتم جستجوی پرتوی محلی با k حالت شاید به نظر چیزی بیش‌تر از اجرای موازی (به جای اجرای به نوبت) k الگوریتم شروع مجدد تصادفی نداشته‌باشد اما در واقع این دو الگوریتم کاملاً متفاوت هستند. در جستجوی شروع مجدد تصادفی هر فرایند جستجو به صورت مستقل از بقیه عمل می‌کند. در جستجوی پرتوی محلی اطلاعات مفیدی در میان k فرآیند جستجوی موازی ردوبدل می‌شود. به عنوان مثال، اگر یک حالت تعداد زیادی گره پسین خوب تولید کند و k-۱ حالت دیگر همگی گره‌های پسین بدی تولید نمایند آنگاه نتیجه‌ی آن این است که اولین حالت به بقیه می‌گوید " اینجا بیایید اینجا نتایج بهتراند". الگوریتم به سرعت جستجوهای بی‌حاصل را رها کرده و منابع خود را به سمتی می‌برد که پیشرفت بیش‌تری حاصل شود.

در ساده‌ترین شکل، جستجوی پرتوی محلی ممکن است از کمبود تنوع میان k حالت رنج ببرد در این صورت آنها به سرعت در ناحیه کوچکی از فضای حالت متمرکز شده و جستجو را به مقدار کمی پرهزینه‌تر از یک نسخه پرهزینه تپه‌نوردی می‌کند. الگوریتم دیگری به نام جستجوی پرتوی تصادفی (متشابه الگوریتم تپه‌نوردی تصادفی) کمک می‌کند که این مشکل تعدیل شودو به جای انتخاب بهترین k گره پسین از بین گره‌های پسین موجود، k گره پسین را به صورت تصادفی انتخاب می‌کند به طوری که احتمال انتخاب شدن یک گره پسین تابعی صعودی از مقدارش است. جستجوی پرتوی تصادفی تا حدودی به فرآیند انتخاب در طبیعت شبیه است، که در آن گره‌های پسین (فرزند) یک حالت (موجود زنده) نسل بعد از خود را بر اساس مقدار (شایستگی) آنها تولید می‌کنند.

```
function Simulated-Annealing(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a"temperature" controlling the probability of downward steps

    Current ← Make-Node(Initial- State [problem])
    for t ←۱ to ∞ do
        T ← schedule[t]
        if T=۰ then return current
        next ← a randomly selected successor of current
        ΔE ←Value[next]-Value[current]
        if ΔE >۰ then current ← next
        else current ← next only with probability e^(ΔE/T)
```

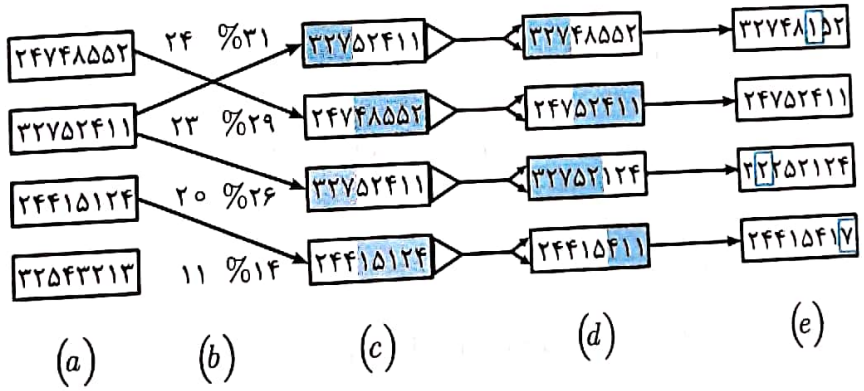
شکل ۴-۱۴ الگوریتم شبیه‌سازی ذوب(نسخه تپه‌نوردی تصادفی که در آن بعضی از حرکت‌های به سمت پایین اجازه داده می‌شود). حرکت‌های به‌سمت پایین دقیقا در اوایل زمان‌بندی ذوب پذیرفته می‌شود و هرچه زمان می‌گذرد احتمال پذیرش آن کمتر می‌شود. ورودی این زمان‌بندی مقدار T را به عنوان تابعی از زمان تعیین می‌کند.

۴-۳-۴- الگوریتم ژنتیک

الگوریتم ژنتیک[۱] یا (GA) نوعی از جستجوی پرتوی تصادفی[۲] است که در آن حالت‌های پسین از ترکیب دو حالت به عنوان والد (به جای تغییر در یک حالت) به وجود می‌آیند. عملکرد انتخاب طبیعی[۳] بسیار شبیه به جستجوی پرتوی تصادفی است.

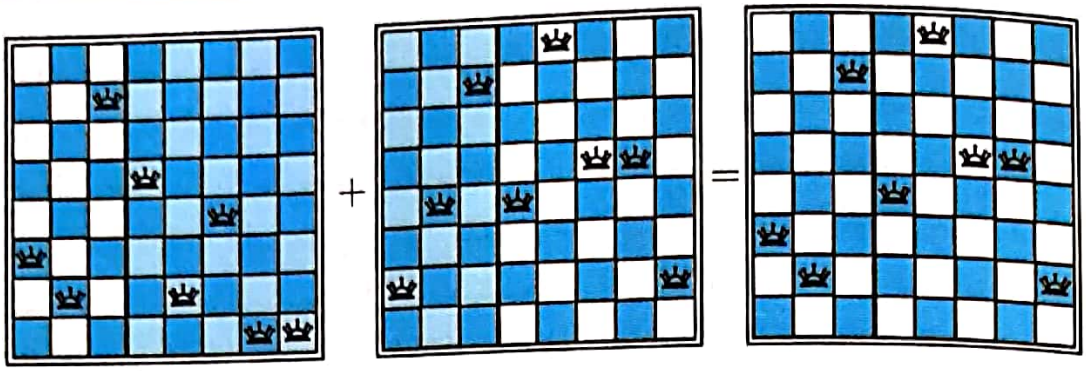
GA همچون جستجوی پرتو با مجموعه‌ای از k حالت تصادفی شروع می‌شود که به آن جامعه[۴] گویند که در آن هر حالت (یا فرد[۵]) با رشته‌ای از مجموعه‌ای متناهی از حروف نمایش داده می‌شود (متداول‌ترین آن‌ها رشته‌هایی از ۰ و ۱هاست). به عنوان مثال، حالت‌ها در مسئله ۸- وزیر مکان هریک از ۸ وزیر (هر کدام در خانه‌ای از یک ستون) را مشخص می‌کنند بنابراین به $۸ \times \log_۲ ۸ = ۲۴$ بیت نیاز داریم. این حالات همچنین با ۸ رقم بین ۱ تا ۸ نیز قابل نمایش می‌باشند.(بعدها خواهیم دید که این دو نوع کدگذاری به گونه متفاوتی عمل می‌کنند.) شکل ۴-۱۵(a) جامعه‌ای از ۴ رشته ۸ رقمی را که حالات ۸-وزیر را نمایش می‌دهند نشان می‌دهد.

نسل بعد حالات تولید شده در شکل ۴-۱۵ (e)-(b) نمایش داده شده‌است. در (b) هر حالت بر اساس تابع ارزیابی یا (در اصطلاح GA) تابع برازندگی[۶] رده بندی شده‌است. تابع برازندگی به ازای حالات بهتر عدد بزرگتری را برمی‌گرداند بنابراین برای مسئله ۸- وزیر این مقدار برابر تعداد جفت وزیرانی است که به یکدیگر حمله نمی‌کنند که مقدار آن برای راه حل مسئله برابر ۲۸ است. این مقدار برای ۴ حالت فوق الذکر ۲۴، ۲۳، ۲۰ و ۱۱ است. در این نوع خاص از الگوریتم ژنتیک احتمال انتخاب یک حالت برای تولید مثل به طور مستقیم با نمره برازندگی آن نسبت دارد که درصدها در کنار نمره‌های خام آن‌ها نشان داده شده‌است.



شکل ۴-۱۵ الگوریتم ژنتیک. جمعیت اولیه در (a) توسط تابع برازندگی در (b) دسته‌بندی می‌شود. که در نهایت منجر به جفت شدن آن‌ها در (c) می‌شود. آن‌ها در (d) فرزندان خود را تولید می‌کنند. و در (e) جهش می‌یابند.

۱- Genetic Algorithm
۲- Stochastic Beam Search
۳- Natural Selection
۴- Population
۵- individual
۶- Fitness function



شکل ۴-۱۶ حالت‌های ۸- وزیر متناظر با ۲ حالت والد در شکل ۴-۱۵ (c) و فرزندان اول در شکل ۴-۱۵ (d). ستون‌های هاشور زده شده در مرحله ادغام از بین رفته و ستون‌های دیگر باقی مانده‌اند.

در (c) دو جفت براساس احتمال نشان داده شده در (b) به صورت تصادفی برای تولید مثل انتخاب شده‌اند. به این نکته توجه کنید که یکی از افراد دوبار و یکی هیچ‌گاه انتخاب نشده‌است. در این الگوریتم در هر بار جفت شدن یک نقطه به صورت تصادفی در رشته مورد نظر برای ادغام[1] انتخاب می‌شود. در شکل ۴-۱۵ نقاط ادغام برای جفت اول بعد از رقم سوم و برای جفت دوم بعد از رقم پنجم است.

در قسمت (d) فرزندان از ادغام رشته‌های والد در نقطه ادغام به وجود می‌آیند. به عنوان مثال فرزند نخست اولین جفت سه رقم اول را از والد اول و بقیه رقم‌ها را از والد دوم می‌گیرد از سوی دیگر فرزند دوم سه رقم اول را از والد دوم و مابقی را از والد اول می‌گیرد. حالت‌های ۸-وزیری که در این تولید مثل شرکت داشتند در شکل ۴-۱۶ نشان داده شده‌است. این مثال نشان می‌دهد که وقتی دو حالت والد کاملا متفاوت هستند آنگاه عمل ادغام می‌تواند حالتی را تولید کند که با والدین آن بسیار متمایز باشد. گاهی اوقات در ابتدای فرآیند جامعه بسیار ناهمگون است بنابراین ادغام (همچون الگوریتم شبیه سازی گداختگی) در ابتدای فرآیند جستجو گام‌های بزرگی در فضای جستجو برمی‌دارد و بعد از مدتی که همه‌ی افراد همگون شدند گام‌ها کوچک‌تر می شود.

سرانجام در (e) همه مکان‌ها با احتمال اندک و مستقلی به طور تصادفی جهش می‌کنند. در شکل (e) در هر یک از بچه‌های اول، سوم و چهارم یک عدد جهش یافته‌است. در مسئله ۸- وزیر این جهش نظیر انتخاب یک وزیر به طور تصادفی و انتقال آن به یک خانه‌ی تصادفی در همان ستون است. شکل ۴-۱۷ الگوریتمی که همه‌ی این مراحل را پیاده سازی کرده‌است نشان می‌دهد.

همچون الگوریتم پرتوی تصادفی، الگوریتم ژنتیک دو ایده تمایل به صعود با اکتشاف تصادفی و تبادل اطلاعات میان فرایندهای موازی جستجو را ترکیب کرده‌است. نخستین مزیت الگوریتم ژنتیک (در صورت وجود) عمل ادغام است. به طور شهودی می‌توان دریافت که مزیت این الگوریتم به دلیل قابلیت ادغام بلاک‌های بزرگی از حروف است که به صورت مستقل تکامل یافته‌اند. می‌توان به صورت ریاضی نشان داد که اگر در ابتدای الگوریتم مکان کدهای ژنتیک به صورت تصادفی بایکدیگر عوض کنیم آنگاه دیگر عمل ادغام بی‌فایده خواهد بود زیرا موجب افزایش سطح دانه بندی(درشتی) عملکرد جستجو می‌شود (زیرا با این کار بلاک‌های بزرگ مستقلاً تکامل یافته را شکسته و به دانه‌های کوچک‌تر تقسیم می‌کنیم).

[1]- Crossover

نظریه‌ی الگوریتم‌های ژنتیک برای پیاده‌سازی این مسئله از ایده‌ی الگو[۱] استفاده می‌کند. الگو زیر رشته‌ای است که در آن مقدار بعضی از مکان‌ها نامشخص است. برای مثال الگوی ۲۴۶*****، وضعیت ۸ وزیری را توصیف می‌کند که در آن سه وزیر اول به ترتیب در خانه‌های ۲، ۴ و ۶ قرار دارند. رشته‌هایی که در این الگو صدق می‌کنند (مثل ۲۴۶۱۳۵۷۸) را نمونه‌هایی از این الگو می‌نامند. می‌توان نشان داد که اگر میانگین عدد برازندگی نمونه‌های یک الگو بالاتر از میانه باشد، آنگاه تعداد نمونه‌های آن الگو با گذشت زمان در جامعه بیش‌تر می‌شود. بدیهی است در صورتی که بیت‌های مجاور کاملاً مستقل و نامرتبط به یکدیگر باشند این ویژگی چندان معنی پیدا نمی‌کند، زیرا در این صورت بلاک‌های پیوسته بسیار کمی پیدا می‌شود که دارای منافع هم‌جهت با یکدیگر باشند. الگوریتم ژنتیک در صورتی که الگوهای آن با مؤلفه‌های بامعنی مسئله ارتباط داشته‌باشند، بهترین عملکرد را دارد. برای مثال اگر رشته‌ی مورد نظر نشان دهنده‌ی یک آنتن است، آنگاه الگوها در آن می‌تواند نشان دهنده‌ی مؤلفه‌ای از آنتن مثل بازتابنده[۲] یا انحراف دهنده[۳] باشد. یک مؤلفه خوب شاید برای طراحی‌های متفاوتی مناسب باشد. این بدین معنی است استفاده موفق از الگوریتم ژنتیک نیازمند مهندسی دقیق نمایش این رشته‌هاست.

```
function Genetic-Algorithm(population, Fitness- Fn) returns an individual
  inputs: population, a set of individuals
          Fitness-Fn, a function that measures the fitness of an individual
  repeat
    new-population ← empty set
    loop for i from ۱ to size(population) do
      x ← Random-Selection(population, Fitness-Fn)
      y ← Random-Selection(population, Fitness-Fn)
      child ← Reproduce(x,y)
      if(small random probability) then child ← Mutate(child)
      add child to new-population
    population ← new- population
  until some individual is fit enough, or enough thime has elapsed
  return the best individual in population, according to Fitness-Fn
```

```
function Reproduce(x,y) returns an individual
  inputs: x,y, parent individuals

  n ← Length(x)
  c ← random number from ۱ to n
  return: Append(Substring(x,۱,c), Substring(y,c + ۱,n))
```

شکل ۴-۱۷ الگوریتم ژنتیک. این الگوریتم همانند الگوریتم نمایش داده شده در شکل ۴-۱۵ است اما با یک تفاوت: در این نسخه مشهورتر، از جفت شدن دو والد تنها یک فرزند بوجود می‌آید نه دو تا.

۱- Schema
۲- Reflector
۳- Deflector

در عمل، الگوریتم‌های ژنتیک کاربردهای گسترده‌ای در مسائل بهینه‌سازی نظیر طراحی مدار، برنامه ریزی کارها دارد. در حال حاضر هنوز مشخص نیست که آیا همه گیر شدن الگوریتم ژنتیک به دلیل کارایی آن می‌باشد یا به دلیل اینکه منشأ آن در نظریه‌ی تکامل است. هنوز باید کارهای زیادی انجام شود برای اینکه دریافت که الگوریتم ژنتیک تحت چه شرایطی خوب عمل می‌کند.

۴-۴- جستجوی محلی در فضای پیوسته

در فصل ۲ تفاوت میان محیط‌های گسسته و پیوسته را شرح دادیم همچنین گفته شد که بیش‌تر محیط‌های واقعی در جهان، پیوسته هستند. هیچ کدام از الگوریتم‌هایی که ما تاکنون معرفی کردیم نمی‌توانند فضاهای حالت پیوسته را اداره کنند زیرا تابع پسین در آن‌ها در اغلب موارد تعداد نامتناهی حالت را برمی‌گردانند. در این قسمت به صورت خلاصه به معرفی بعضی از تکنیک‌های جستجوی محلی در فضای حالت پیوسته برای یافتن پاسخ بهینه می‌پردازیم. مقالات در این زمینه بسیار گسترده‌اند. بسیاری از تکنیک‌های اساسی در قرن ۱۷ بعد از گسترش ماشین حساب توسط نیوتن و لیبنیز ابداع شده‌اند. ما در بسیاری از بخش‌های کتاب از جمله یادگیری، بینایی و رباتیک و خلاصه در هر جا که مربوط به دنیای واقعی است از این تکنیک‌ها استفاده می‌کنیم.

بگذارید با یک مثال شروع کنیم. فرض کنید می‌خواهیم سه فرودگاه جدید در کشور رومانی به گونه‌ای قرار دهیم که حاصل‌جمع مربع فاصله هر شهر روی نقشه (شکل ۳-۲)از نزدیک‌ترین فرودگاه نسبت به خود مینیمم باشد. آنگاه فضای حالت را براساس مختصات فرودگاه‌ها تعریف می‌کنیم: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ که یک فضای شش-بعدی است ویا می‌گوییم حالات با شش متغیر تعریف شده‌اند (به طور کلی حالت‌ها با یک بردار n بعدی از متغیر x تعریف می‌شوند). حرکت درون این فضا معادل با حرکت یک یا چند فرودگاه در نقشه است. محاسبه تابع هدف[1] $f(x_1, y_1, x_2, y_2, x_3, y_3)$ به ازای هر حالت خاص نسبتاً ساده است اما نوشتن آن در حالت کلی دشوار است.

یک راه ساده برای اجتناب از مسائل پیوسته، گسسته‌سازی همسایگی حالت‌هاست. به عنوان مثال اجازه می‌دهیم در هر لحظه تنها یک فرودگاه و با مقدار ثابت $\pm s$ در جهت x یا y حرکت کند که با ۶ متغیر به ازای هر حالت ۶ حالت پسین داریم. حال می‌توانیم هر یک از الگوریتم‌های جستجوی محلی توضیح داده شده را برروی آن اعمال کنیم. همچنین الگوریتم‌هایی مثل تپه‌نوردی تصادفی و شبیه سازی گداختگی را می‌توان مستقیماً و بدون گسسته‌سازی فضا اعمال کرد زیرا در آنها حالت‌های پسین به صورت تصادفی انتخاب می‌شوند بنابراین می‌توان بردارهایی تصادفی با اندازه s تولید کرد.

روش‌های زیادی وجود دارد که تلاش می‌کند با استفاده از گرادیان، ماکزیمم یک منحنی را بیابند. گرادیان تابع هدف برداری همانند ∇f است که اندازه و جهت تندترین شیب را می‌دهد. برای این مسئله داریم:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

در بعضی موارد می‌توان با حل معادله $\nabla f = \phi$ ماکزیمم را پیدا کرد (به عنوان مثال اگر تنها یک فرودگاه داشته باشیم آنگاه پاسخ مسئله میانگین مختصات تمامی شهرها خواهد بود). با این حال در بسیاری از موارد این معادله

[1]- Objective Function

درحالت فرم بسته[۱] آن غیر قابل حل خواهد بود. به عنوان مثال در مسئله‌ی سه فرودگاه، عبارت گرادیان به این بستگی دارد که کدام شهرها به هریک از فرودگاه‌ها در حالت جاری نزدیک‌تر است. این بدین معنی است که گرادیان به صورت محلی و نه به صورت کلی قابل حل است. بنابراین با گذاشتن حالت جاری در فرمول زیر و بروزرسانی حالت جاری بوسیله آن، می‌توان الگوریتم پرشیب‌ترین صعود در تپه‌نوردی را اجرا کرد:

$x \leftarrow x + a\nabla f(x)$ (a ثابتی کوچک است)

حال ممکن است گاهی تابع هدف مشتق‌پذیر نباشد. به عنوان مثال مقادیر مجموعه‌ی خاصی از مکان فرودگاه‌ها ممکن است با استفاده از نرم‌افزارهای شبیه‌ساز اقتصادی محاسبه شود. در این گونه موارد گرادیان تجربی را می‌توان از طریق پاسخ به تغییرات مثبت و منفی بر روی مختصات‌ها بدست آورد. جستجوی گرادیان تجربی مانند الگوریتم انتخاب پرشیب‌ترین صعود تپه‌نوردی در فضای حالت گسسته عمل می‌کند.

با توجه به اینکه "a مقداری ثابت و کوچک است" روش‌های بسیار زیادی برای تعیین a مطرح شده‌است. مشکل اصلی این است که اگر a بسیار کوچک باشد، آنگاه تعداد مراحل زیاد می‌شود و اگر a بسیار بزرگ باشد جستجو ممکن است از مقدار ماکزیمم تجاوز کرده و مرتکب خطا شود. تکنیک جستجوی خطی تلاش می‌کند این مشکل را ازطریق گسترش جهت گرادیان فعلی (معمولا با دو برابر کردن مقدار a) برطرف نماید این گسترش جهت گرادیان ادامه می‌یابد تا زمانی که مقدار f شروع به کاهش بکند. نقطه‌ای که در آن چنین اتفاقی می‌افتد به عنوان حالت جاری انتخاب می‌شود. عقاید متفاوتی در زمینه چگونگی انتخاب جهتی جدید برای این نقطه وجود دارد.

در بسیاری از مسائل کاراترین الگوریتم، روش با ارزش نیوتن-راپسون است (نیوتن:۱۶۷۱ راپسون:۱۶۹۰) که روشی کلی برای بدست آوردن ریشه‌های یک تابع(حل معادلاتی به شکل $g(x) = 0$) است. روش کار بدین صورت است که بر اساس فرمول نیوتن، تقریبی از ریشه x محاسبه می‌شود:

$$x \leftarrow x - g(x) / g(x)'$$

برای اینکه ماکزیمم و مینیمم f را بیابیم باید به دنبال xای بگردیم که به ازای آن گرادیان صفر شود ($\nabla f(x) = 0$) بنابراین $g(x)$ در فرمول نیوتن همان $\nabla f(x)$ است. بنابراین می‌توان نوشت:

$$x \leftarrow x - H_f^{-1}(x)\ \nabla f(x))$$

که در آن $H_f(x)$ ماتریس Hessian مشتق دوم است که عناصر H_{ij} به صورت $\partial^2 f/\partial x_i \partial x_j$ داده شده‌اند. از آنجا که Hessian، n^2 تا عنصر دارد، روش نیوتن-راپسون در فضاهایی با ابعاد بزرگ بسیار پر هزینه می‌شود به همین دلیل تقریب‌های بسیار زیادی ابداع شده‌است.

روش‌های جستجوی محلی از وجود ماکزیمم‌های محلی،کوهان‌ها و فلات‌ها در فضاهای جستجوی پیوسته(همچون گسسته) رنج می‌برد. الگوریتم‌های شروع مجدد تصادفی[۲] و شبیه‌سازی گداختگی گاهی اوقات می‌توانند مفید واقع شوند اگرچه فضاهای پیوسته چندبعدی مکان‌های بزرگی هستند که احتمال گم شدن در آن‌ها بسیار زیاد است.

۱- Closed Form
۲- Random Restart

مطلب آخری که مفاهیم گفته شده در آن بسیار پرکاربرد است مسائل بهینه‌سازی محدود شده[1] است. زمانی به یک مسئله‌ی بهینه‌سازی، محدود شده گویند که در آن پاسخ مسئله محدودیت‌های اعمال شده برروی مقادیر متغیرها را ارضاء کند. برای مثال در مسئله‌ی فرودگاه‌ها ممکن است مکان قرار دادن آن‌ها را فقط محدود به خاک کشور رومانی و درون خشکی بکنیم (به جای وسط دریاچه). دشواری مسائل بهینه‌سازی محدود شده بیش‌تر به ذات محدودیت‌های اعمال شده و تابع هدف بستگی دارد که شناخته شده‌ترین گروه از آنها مسائل برنامه‌ریزی خطی هستند که در آن‌ها محدودیت‌ها، نامساوی‌هایی خطی هستند که یک ناحیه محدب را تشکیل می‌دهند و از طرفی تابع هدف نیز خطی است. مسائل بهینه‌سازی خطی می‌توانند در زمان چندجمله‌ای بر حسب تعداد متغیرها حل شوند. این مسائل با محدودیت‌های متفاوت و توابع هدف متفاوت مورد مطالعه قرار گرفته‌اند (همچون برنامه‌ریزی درجه دو، برنامه ریزی درجه چهار و غیره).

۴-۵- عامل‌های جستجوی برخط و محیط‌های ناشناخته

تااینجا تمرکز ما برروی عامل‌هایی بود که از الگوریتم‌های جستجوی "برون خط" استفاده می‌کردند. آنها راه حل را به صورت کامل محاسبه، سپس پا به دنیای واقعی گذاشته و راه حل خود را بدون توجه به ادراکات (دریافتی از محیط) خود اجرا می‌کنند (به شکل ۳-۱ توجه کنید).در نقطه‌ی مقابل جستجوی "برخط" به صورت یکی در میان محاسبه و کنش انجام می‌دهد: ابتدا یک کنش بروز می‌دهد، سپس محیط را تحت نظر داشته و کنش بعدی را بروز می‌دهد. جستجوی "برخط" برای محیط‌های پویا و نیمه پویا ایده‌ی مناسبی است (محیط‌هایی که در آن‌ها تعلل حاصل از محاسبات طولانی ما را دچار خسران می‌کند). جستجوی "برخط" حتی در محیط‌های تصادفی نیز ایده‌ی خوبی است. به طور کلی جستجوی "برون خط" با رویدادهایی سروکار دارد که به صورت نمایی بزرگ و در آن‌ها تمامی پیشامدهای ممکن بررسی می‌شود. در صورتی که جستجوی "برخط" تنها به رویدادهای اتفاق افتاده توجه می‌کند. به عنوان مثال عاقلانه است که یک عامل بازی شطرنج بدون توجه به مسیر بازی، از قبل حرکت اول خود را انتخاب کرده‌باشد.

جستجوی "برخط" برای مسائل اکتشافی که در آنها حالات و کنش‌ها برای عامل نامعلوم است بسیارضروری است. یک عامل در این حالت به دلیل جهلی که(از محیط) دارد باید از کنش‌های خود به عنوان آزمایشی برای تعیین کنش‌های بعدی استفاده کند و در نتیجه باید محاسبه و کنش را به صورت یکی در میان انجام دهد.

یک مثال متداول از جستجوی "برخط" رباتی است که در ساختمان جدیدی قرار داده شده‌است و باید آن را کشف کند تا نقشه‌ای بسازد که به وسیله آن بتواند از نقطه A به نقطه B برود. روش‌های بیرون آمدن از مارپیچ (دانش مورد نیاز برای قهرمانان باستان) نیز نمونه‌ای از الگوریتم جستجوی "برخط" است. یک نوزاد تازه متولد شده را در نظر بگیرید: نوزاد احتمال بروز کنش‌های زیادی را دارد، اما نتیجه بروز هیچ یک از این کنش‌ها را نمی‌داند بلکه به تدریج مقداری از آنها را در حد توان خود تجربه می‌کند. کشف تدریجی یک نوزاد از عملکرد محیط اطراف خود نوعی فرآیند جستجوی "برخط" است.

[1]- onstrained OptimizationC

۴-۵-۱- مسائل جستجوی "برخط"

مسئله‌های جستجوی "برخط" تنها توسط عامل‌هایی قابل حل هستند که به جای فرآیند محاسبه‌ی محض از خود کنش بروز می‌دهند. فرض کنید، عامل تنها از موارد زیر مطلع است:

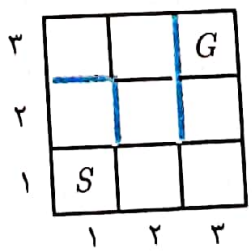
- ACTIONS(s) لیستی از کنش‌هایی که در حالت s می‌تواند بروز دهد را بر می‌گرداند.
- تابع هزینه هر مرحله c(s,a,s') -توجه داشته باشید این تابع تا زمانی که عامل نمی‌داند که نتیجه s' است قابل استفاده نیست.
- GOAL-TEST(s)

به این نکته توجه داشته باشید که یک عامل نمی‌تواند به حالت‌های پسین یک حالت دسترسی داشته‌باشد مگر اینکه واقعاً تمامی کنش‌های ممکن برای آن حالت را امتحان کرده‌باشد. به عنوان مثال در مسئله‌ی مارپیچ که در شکل ۴-۱۸ نشان داده شده‌است عامل نمی‌داند که با حرکت به سمت بالا از (۱,۱) به نقطه (۱,۲) می‌رسد و اگر به سمت بالا برود مجدداً نمی‌داند که با حرکت به سمت پایین به نقطه (۱,۱) می‌رسد. این درجه از ناآگاهی در بعضی روش‌ها قابل تقلیل است. برای مثال یک ربات کاوشگر ممکن است درمورد چگونگی عملیات حرکتی خود اگاهی داشته باشد و فقط از مکان موانع بی‌اطلاع باشد.

حال فرض می‌کنیم که که عامل حالتی را که قبلا دیده می‌شناسد و کنش‌های عامل به صورت قطعی هستند. سرانجام عامل به تابع ابتکاری قابل پذیرش h(s) دسترسی دارد که فاصله بین حالت جاری و حالت هدف را تقریب می‌زند. به عنوان مثال در شکل ۴-۱۸ عامل مکان هدف را می‌داند و می‌تواند از تابع ابتکاری فاصله Manhattan استفاده کند.

به طور معمول مقصود عامل پیدا کردن هدف با کمترین هزینه است (شاید در بعضی موارد هدف پویش کامل محیط باشد). هزینه همان هزینه‌ی کلی مسیری است که عامل واقعاً آن را طی می‌کند که معمولا این هزینه، را با هزینه‌ای که عامل در صورت آگاهی کامل از فضای جستجو می‌پیماید(یعنی همان هزینه‌ی واقعی کوتاهترین مسیر) مقایسه می‌کنند که در اصطلاح الگوریتم "برخط" به آن نسبت رقابتی[1] گویند و ما می‌خواهیم تا جای ممکن آن را کوچک کنیم.

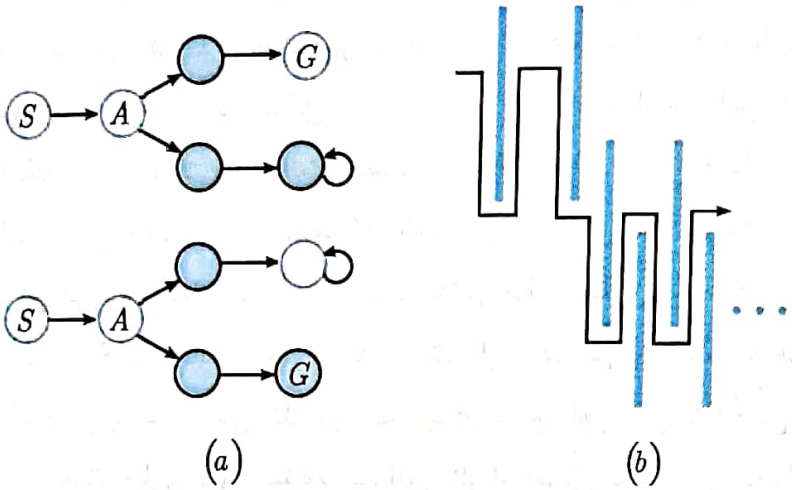
اگرچه این موضوع منطقی به نظر می‌رسد اما در بعضی موارد مشاهده می‌شود که بهترین نسبت رقابتی قابل دسترس بی‌نهایت است. به عنوان مثال اگر بعضی کنش‌ها برگشت‌ناپذیر باشند آنگاه جستجوی "برخط" ممکن است به طور اتفاقی به حالت بن‌بست برسد که از آنجا دیگر هیچ هدفی قابل دسترسی نباشد.



شکل ۴-۱۸ یک مسئله مارپیچ ساده. عامل از S شروع به کار می‌کند، و باید به G برسد وهیچ اطلاعاتی درباره محیط ندارد.

[1]- Competitive ratio

شاید بگویید عبارت " به طور اتفاقی" غیر قابل پذیرش است زیرا ممکن است الگوریتمی باشد که هنگام پویش دچار بن‌بست نشود. ما ادعا می‌کنیم که چنین الگوریتمی وجود ندارد که هیچگاه در فضای حالت به بن‌بست نخورد. به دو فضای حالت بن‌بست در شکل ۴-۱۹(a) دقت کنید. از نظر جستجوی "برخط"ای که دو حالت S و A را بررسی کرده‌است این دو فضای حالت یکسان هستند بنابراین باید تصمیم مشابهی در آنها اتخاذ کند. بنابراین در یکی از آن‌ها به شکست می‌خورد. این مثالی از برهان خصمانه[1] است (بدین معنی که دشمنی را تصور می‌کنیم که فضای حالت را می‌سازد و هنگامی که عامل آن را پویش می‌کند دشمن هدف‌ها و بن‌بست‌ها را تغییر مکان داده و هر جا که بخواهد می‌گذارد).



شکل ۴-۱۹ (a) دو فضای حالتی که ممکن است عامل جستجوی برخط را به بن‌بست برساند. هر عامل دلخواه حداقل در یکی از این دو فضا شکست می‌خورد. (b) یک محیط دوبعدی که عامل را به سمت یک مسیر دلخواه غیربهینه تا هدف می‌کشاند. هر گزینه‌ای را که عامل انتخاب می‌کند، مجددا دیوارها عامل را مجبور به طی کردن مسیر طولانی‌تری می‌کنند، به همین دلیل مسیر طی شده بسیار طولانی‌تر از بهترین مسیر موجود است.

بن‌بست‌ها یکی از مشکلات واقعی ربات‌های کاوشگر هستند (پله‌ها، پیچ‌ها، صخره‌ها و پرتگاه‌ها و همه‌ی مناطق طبیعی دارای احتمال به وجود آوردن کنش‌هایی برگشت ناپذیر هستند). حال فرض می‌کنیم که فضای جستجو ایمن[2] است (بدین معنی که به ازای هر حالت قابل دسترسی یک هدف قابل دسترس وجود داشته باشد). فضاهای حالتی با کنش‌های برگشت پذیر همچون بازی مارپیچ و ۸- پازل را می‌توان جزء گراف‌های بدون جهت دسته‌بندی کرد که جستجو در آنها ایمن است.

حتی در محیط‌های ایمن هم اگر مسیرهایی با هزینه بی‌نهایت وجود داشته باشد آنگاه نمی‌توان تضمین کرد که عدد نسبت رقابتی کران‌دار باشد. همانطور که در شکل ۴-۱۹ (b) نشان داده‌شده‌است اثبات این مسئله برای محیط‌هایی با کنش‌های برگشت ناپذیر بسیار راحت است اما این مسئله در موارد برگشت پذیر نیز صادق است. به همین دلیل به طور معمول در توصیف کارایی الگوریتم‌های جستجوی "برخط" (در نسبت رقابتی) به جای استفاده از عمق کم عمق‌ترین هدف از اندازه کل فضای حالت استفاده می‌کنند.

۴-۵-۲- عامل‌های جستجوی "برخط"

عامل بعد از هر کنشی که از خود بروز می‌دهد ادراکی را از محیط دریافت می‌کند که به او می‌گوید به کدام حالت رسیده‌است. با استفاده از این اطلاعات عامل می‌تواند نقشه(شناخت) خود را از محیط افزایش دهد. سپس

[1]- Adversary argument
[2]- Safely Explorable

با استفاده از این نقشه در مورد حرکت بعدی خود تصمیم‌گیری می‌کند. انجام یکی در میان دو عمل کنش و تصمیم‌گیری بدین معنی است که جستجوی "برخط" با الگوریتم‌های جستجوی "برون خط" که قبلاً دیدیم بسیار متفاوت است. به عنوان مثال در الگوریتم‌های "برون خط" مثل A* این امکان وجود دارد که گره‌ای را در دو نقطه از فضای حالت به صورت همزمان بسط داد زیرا بسط گره‌ها در آن نوعی شبیه‌سازی است و به صورت کنش‌های واقعی نیست. در صورتی که در الگوریتم‌های جستجوی "برخط" تنها می‌توانیم گره‌ای را بسط دهیم که به طور فیزیکی در آن قرار داریم. برای جلوگیری از حرکت بیش از حد برروی کل مسیر درخت برای بسط یک گره بهتر است گره‌ها را به صورت منطقه‌ای بسط دهیم. جستجوی عمق-اول دقیقاً این ویژگی را دارد زیرا همواره گره‌ای که بسط داده می‌شود فرزند گره پیشین بسط داده شده‌است (جز در موارد برگشت به عقب). یک عامل جستجوی عمق-اول "برخط" در شکل ۴-۲۰ نشان داده شده‌است. این عامل نقشه‌ی خود را در جدولی مثل result[a,s] نگه داشته و نتیجه حالت بدست آمده از اجرای کنش a را در حالت s ذخیره می‌کند. تا زمانی که کنشی از حالت جاری وجود داشته‌باشد که عامل آن را پویش نکرده‌باشد، عامل آن کنش را امتحان می‌کند. مشکل، زمانی شروع می‌شود که عامل تمامی کنش‌های حالت جاری را امتحان کرده‌باشد. در این حالت در جستجوی عمق-اول "برون خط" حالت دیگری به راحتی از صف بیرون کشیده می‌شود اما در جستجوی "برخط" عامل به صورت فیزیکی برگشت می‌کند. در جستجوی عمق-اول این به معنی برگشت به حالتی است که عامل به تازگی از آن به حالت جاری وارد شده‌است. این کار از طریق نگهداری جدولی که به ازای هر حالت لیستی از حالت‌های پیشینی که هنوز عامل به آن‌ها برگشت نکرده‌است را داشته باشد امکان پذیر است. اگر حالت‌هایی که عامل می‌تواند به آنها برگشت کند تمام شود آنگاه جستجو کامل است.

به خوانندگان پیشنهاد می‌کنیم که مسیر پیشرفت ONLINE-DFS-AGENT را که بر روی مسئله ماریچ شکل ۴-۱۸ اعمال شده‌است دنبال کنند. به راحتی می‌توان دریافت که عامل در بدترین حالت در پایان الگوریتم از هر لینک در فضای حالت دوبار عبور کرده‌است که در مسائل اکتشافی، پویشی بهینه است. از طرف دیگر حالتی را تصور کنید که هدف بسیار نزدیک به حالت اولیه است و عامل مسیری طولانی را(برای یافتن هدف) طی می‌کند در این حالت عدد نسبت رقابتی می‌تواند بسیار نامناسب شود. نوع "برخط" از الگوریتم عمیق‌شونده‌ی تکراری، این مسئله را (در محیط‌هایی که درخت یکنواخت هستند) حل کرده که عدد نسبت رقابتی چنین عامل ثابتی کوچک است.

این الگوریتم به دلیل خصوصیت برگشت به عقب، در فضاهای حالتی به درستی عمل می‌کند که در آن‌ها کنش‌ها برگشت پذیرند. الگوریتم‌های پیچیده دیگری وجود دارند که عموماً (در فضاهای حالت) خوب عمل کنند، اما در هیچ یک از آنان نسبت رقابتی عددی کران‌دار نیست.

```
function ONLINE-DFS-AGENT(s′)returns an action
  inputs: s′, a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the backtracks not yet tried
          unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
          s, a, the previous state and action, initially null
  if Goal-Test(s′) then return stop
  if s′ is a new state then unexplored[s′]← Actions(s′)
  if s is not null then do
     result[a,s]← s′
```

```
        add s to the front of unbacktracked[s']
    if unexplored[s'] is empty then
        if unbacktracked[s'] is empty then return stop
        else a← an action b such that result[b,s'] = Pop (unbacktracked[s'])
    else a ← Pop(unexplored[s'])
    s ←s'
    return a
```

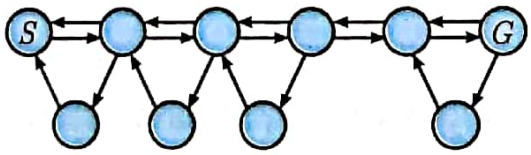
شکل ۴-۲۰ یک عامل جستجوی برخط که از پویش عمق-اول استفاده می‌کند. عامل تنها در فضاهای حالت دوطرفه قابل پیاده‌سازی است.

۴-۵-۳- جستجوی محلی "برخط"

جستجوی تپه‌نوردی همچون جستجوی عمق-اول، خصوصیت بسط منطقه‌ای گره‌ها را دارا می‌باشد. در واقع از آنجا که این الگوریتم تنها حالت جاری را در حافظه نگه می‌دارد نوعی جستجوی "برخط" محسوب می‌شود. متأسفانه این الگوریتم در ساده‌ترین شکل آن چندان مفید نیست زیرا عامل را در ماکزیمم محلی گیر انداخته و دیگر حرکت نمی‌کند. از طرفی الگوریتم شروع مجدد تصادفی نیز قابل استفاده نیست زیرا عامل نمی‌تواند (به طور ناگهانی) خود را به حالت جدیدی انتقال دهد.

به جای الگوریتم شروع مجدد تصادفی می‌توان از قدم‌های تصادفی[1] برای پویش محیط استفاده کرد. در الگوریتم قدم‌های تصادفی (با این فرض که فضای حالت کران‌دار است) از میان کنش‌های موجود در حالت جاری یکی به صورت تصادفی انتخاب می‌شود (ترجیحاً کنش‌هایی که تا به حال امتحان نشده‌اند). به راحتی می‌توان اثبات کرد که الگوریتم قدم‌های تصادفی سرانجام یکی از هدف‌ها را پیدا کرده و یا پویش خود را کامل می‌کند. اما از سوی دیگر این فرآیند ممکن است بسیار کند باشد. شکل ۴-۲۱ محیطی را نشان می‌دهد که در آن مراحل پیدا کردن هدف در الگوریتم قدم‌های تصادفی به صورت نمایی زیاد است زیرا در هر مرحله مسیر بازگشت دو برابر مسیر پیشروی است.

اگرچه این مثال را ما طراحی کردیم اما در دنیای واقعی نیز فضاهای حالت بسیاری وجود دارد که چنین مشکلاتی برای الگوریتم قدم‌های تصادفی به وجود می‌آورند.



شکل ۴-۲۱ محیطی که در آن، گام برداشتن به صورت تصادفی، تعداد مراحل را برای یافتن هدف به تعداد نمایی و بسیار زیاد می‌رساند.

افزایش حافظه برای الگوریتم تپه‌نوردی به جای اجرای آن به صورت تصادفی روش کاراتری خواهد بود. ایده‌ی اصلی ذخیره "بهترین تقریب جاری" H(s) (هزینه رسیدن به هدف) از حالت‌های دیده شده است. H(s) در ابتدا همان مقدار h(s) است؛ اما با گذشت زمان که عامل در فضای حالت کسب تجربه می‌کند به روزرسانی می‌شود.

---

[1]- Random Walk

شکل۴-۲۲ مثال ساده‌ای از یک فضای حالت یک بعدی را نمایش می‌دهد. در قسمت (a) به نظر می‌رسد عامل در قسمت سایه زده شده در یک مینیمم محلی گیر کرده‌است. حال به جای توقف، عامل باید براساس تقریب هزینه جاری بهترین مسیری را که به نظر می‌رسد برای رسیدن به هدف از همسایه‌هایش مناسب است دنبال کند. هزینه‌ی تقریبی رسیدن از همسایه‌ای مثل s' برابر هزینه‌ی رسیدن به s' بعلاوه‌ی هزینه‌ی تقریبی رسیدن از آنجا به هدف است. یعنی $c(s,a,s') + H(s')$. در مثال فوق‌الذکر دو کنش با هزینه‌ی تقریبی ۱+۹ و ۱+۲ وجود دارد بنابراین به نظر می‌رسد که بهتر است به سمت راست حرکت کنیم. حال واضح است که هزینه‌ی تقریبی ۲ برای حالت سایه زده شده بسیار خوش‌بینانه است. از آنجا که هزینه بهترین مسیر ۱ است و سپس به حالتی می‌رسیم که دو مرحله از هدف فاصله دارد بنابراین از حالت سایه زده شده حداقل ۳ مرحله تا هدف فاصله داریم، بنابراین همان طور که در شکل ۴-۲۲(b) نشان داده‌شده‌است H باید به‌روز رسانی شود. در ادامه این فرآیند، عامل بعد از دوبار رفت و برگشت و به‌روز رسانی H و "یکنواخت‌سازی" مینیمم محلی سرانجام به سمت راست می‌رود.

به عاملی که چنین الگویی را پیاده‌سازی می‌کند یادگیرنده بلادرنگ[1] A* (LRTA*) می‌گوییم که در شکل۴-۲۳ نشان داده‌شده‌است. همچون ONLINE-DFS-AGENT این الگوریتم نقشه‌ای از محیط می‌سازد و در جدول result نگه‌داری می‌کند. این الگوریتم هزینه تقریبی را برای حالتی که از آن بیرون می‌آید به‌روزرسانی کرده و حالتی که "ظاهراً بهترین" است را براساس تقریب هزینه جاری برای حرکت انتخاب می‌کند. یک نکته ظریف این است که همواره فرض می‌کنیم کنش‌هایی که هنوز در حالت s امتحان نشده‌اند با حداقل هزینه ممکن h(s) به هدف می‌رسند. این خوش‌بینی تحت عدم قطعیت[2] عامل را تشویق می‌کند که مسیرهای امیدوار کننده موجود دیگر را نیز کشف کند.

عامل LRTA* تضمین می‌کند که در محیط‌های کران‌دار و ایمن هدف را پیدا کند. برخلاف A* این الگوریتم در فضاهای حالت نامحدود غیر کامل است یعنی مواردی وجود دارد که در آن الگوریتم تا بی‌نهایت دچار گمراهی می‌شود. این الگوریتم در بدترین حالت محیطی با n حالت را در زمان $O(n^2)$ پویش می‌کند(اگرچه اغلب بسیار سریع تر عمل می‌کند). عامل LRTA* تنها یک نوع از مجموعه‌ای از عامل‌های "برخط" است که در آن‌ها قوانین انتخاب کنش و قوانین به‌روزرسانی به طرق مختلف تعریف می‌شود.

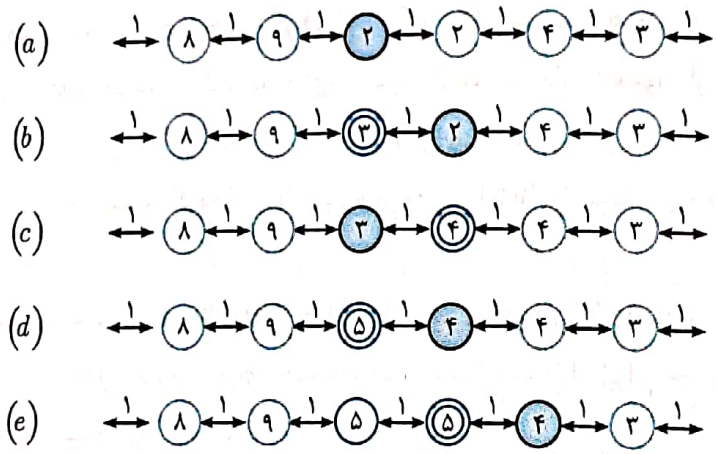
۴-۵-۴- یادگیری در جستجوی "برخط"

عامل‌های جستجوی "برخط" در ابتدا هیچ اطلاعاتی ندارند، این جهل اولیه (از محیط) فرصت‌های مناسبی برای یادگیری فراهم می‌آورد. اولاً عامل نقشه محیط را با استفاده از ثبت تجربه‌هایش -به طور دقیق نتایج هر کنش در هر حالت- می‌آموزد (به این نکته توجه داشته باشید که قطعی بودن محیط بدین معنی است که کافی است هر حالت را تنها یک بار تجربه کنیم). ثانیاً عامل‌های جستجوی محلی با استفاده از قوانین به‌روزرسانی محلی تخمین‌های دقیق‌تری از مقدار هریک از حالت‌ها بدست می‌آورند (همچون LRTA*).

---
[1]- Learning Real-Time
[2]- Optimism Under Uncertainty

خواهیم دید این به‌روزرسانی‌ها بالاخره منجر به مقادیر دقیق حالت‌ها می‌شوند البته با این فرض که عامل فضای حالت را به درستی پیمایش کند. هنگامی که این مقادیر دقیق شناخته شدند آنگاه می‌توان تصمیمات بهینه‌ای را با حرکت به سمت بالاترین مقدار پسین گرفت (در این هنگام استفاده از الگوریتم تپه‌نوردی به تنهایی یک استراتژی بهینه خواهد بود).

اگر به رفتار ONLINE-DFS-AGENT در محیط شکل ۴-۱۸ توجه کنیم درمی‌یابیم که عامل چندان باهوش نیست. به عنوان مثال بعد از اینکه عامل دریافت که کنش "بالا" ما را از نقطه (۱,۱) به (۱,۲) می‌برد هیچ ایده‌ای در مورد اینکه کنش "پایین" ما را به نقطه (۱,۱) بازمی‌گرداند و یا اینکه کنش "بالا" ما را از (۱,۲) به (۲,۲) و یا از (۲,۲) به (۲,۳) می‌برد نداشت. به طور کلی می‌خواهیم به عامل بیاموزیم که "بالا" مختصات y را افزایش می‌دهد مگر اینکه دیواری مانع راه باشد و "پایین" مختصات y را کاهش می‌دهد و غیره. برای این منظور ما به دو چیز نیازمندیم. اول اینکه برای این دسته از قوانین کلی ما نیازمند یک نمایش رسمی، صریح و انعطاف پذیر هستیم. تاکنون ما اطلاعات را درون جعبه سیاهی به نام تابع پسین پنهان می‌کردیم. دوم اینکه نیازمند الگوریتمی هستیم که بتواند این قوانین کلی را از مشاهدات خاصی که عامل انجام می‌دهد بسازد.



شکل ۴-۲۲ پنج تکرار از الگوریتم LRTA* در فضای حالت یک بعدی. هر حالت با H(s) (هزینه تقریب فعلی برای رسیدن به هدف) برچسب‌گذاری شده است و هر یال با هزینه آن مرحله برچسب‌گذاری شده است. حالت سایه‌زده شده مکان عامل را مشخص می‌کند، و مقادیر به‌روز شده در هر مرحله با دایره‌ای دور آن مشخص می‌شوند.

```
function LRTA*-Agent(s′) returns an action
    inputs: s′, a percept that identifies the current state
    static: result, a table, indexed by action and state, initially empty
            H, a table of cost estimates indexed state, initially null
            s, a, the previous state and action, initially null

    if Goal-Test(s′) then return stop
    if s′ is a new state(not in H) then H(s′)← h(s′)
    unless s is null
        result[a,s]← s′
        H[s] ←  min    LRTA*-Cost(s,b, result[b,s],H)
              b∈ Actions(s)
    a ← an action b in Actions(s′) that minimizes LRTA*-Cost(s′,b, result[b,s′],H)
```

```
    s ← s′
    return a

function LRTA*-COST (s,a,s′,H) returns a cost estimate
    if s′ is undefined then return h(s)
    else return c(s,a,s′)+H[s′]
```

شکل ۴-۲۳ LTRA*-AGENT کنش مورد نظر را براساس مقادیر حالت‌های همسایه (که هنگامی که عامل در فضای جستجو حرکت می‌کند به‌روز رسانی می‌شوند) انتخاب می‌کند.

۴-۶- خلاصه

این فصل به بررسی کاربردهای ابتکارها برای کاهش هزینه‌های جستجو می‌پردازد. تعدادی از الگوریتم‌هایی را که از ابتکارها استفاده می‌کنند، بررسی کردیم و دریافتیم که بهینگی حتی با یک ابتکار مناسب هزینه‌ی بسیار زیادی برای جستجو در بردارد.

- جستجوی بهترین- اولین نوعی از جستجوی گراف است که در آن مینیمم هزینه از میان گره‌های بسط داده نشده برای بسط دادن انتخاب می‌شود. الگوریتم‌های بهترین-اولین به طور معمول از یک تابع ابتکاری h(n) استفاده می‌کنند که هزینه رسیدن به جواب را از گره n تخمین می‌زند.
- جستجوی بهترین-اولین حریصانه گره‌هایی با مینیمم مقدار h(n) را بسط می‌دهد. این الگوریتم بهینه نیست، اما اغلب کارا[1] است.
- جستجوی A* گره‌هایی با مینیمم f(n)= g(n)+h(n) را بسط می‌دهد. A* کامل و بهینه است، البته با این فرض که تضمین کنیم h (n) قابل پذیرش (برای جستجو در درخت) و سازگار (برای جستجو گراف) است. اگرچه پیچیدگی فضای حافظه A* بسیار پرهزینه است.
- کارایی الگوریتم‌های جستجوی ابتکاری وابسته به تابع ابتکاری آن است. گاهی می‌توان با ساده‌سازی صورت مسئله ویا محاسبه هزینه پاسخ مسئله (از قبل) برای زیرمسئله‌ها و نگهداری آن‌ها در الگو پایگاه داده و یا با یادگیری از طریق تجربیات و آزمایشات ابتکارهای بسیار خوبی را ابداع کرد.
- RBFS و SMA*، الگوریتم‌های جستجوی قوی و بهینه‌ای هستند که از مقدار اندکی حافظه استفاده می‌کنند. با فرض اینکه زمان کافی به این الگوریتم داده شود، مسائلی را که توسط A* به دلیل کمبود حافظه قابل حل نیست را حل می‌کنند.
- روش‌های جستجوی محلی همچون الگوریتم تپه‌نوردی براساس فرمول‌بندی کامل حالت کار می‌کنند. تعداد اندکی گره در حافظه نگه می‌دارند. الگوریتم‌های تصادفی(همچون شبیه‌سازی ذوب که در صورتی که زمان-بندی مناسبی برای سرد شدن T به آن داده شود جواب بهینه مسئله را پیدا می‌کند) بسیار زیادی توسعه داده شده است.

[1]- Efficient

- الگوریتم ژنتیک یک نوع جستجوی تپه‌نوردی تصادفی است که در آن جمعیت بزرگی از حالت‌ها نگهداری می‌شود. حالت‌های جدید از طریق جهش و ادغام (که در آن جفت‌هایی از جمعیت ترکیب می‌شوند) تولید می‌شوند.

- مسائل اکتشافی[۱]، مسائلی هستند که در آن عامل، هیچ ایده‌ای درباره حالت‌ها و کنش‌های محیط خود ندارد. "عامل جستجوی برخط" می‌تواند نقشه‌ای ساخته و هدف را در صورت وجود پیدا کند. "ابتکار تقریبی به‌روز کننده[۲]" با استفاده از تجربیات، روشی مؤثر را برای بیرون رفتن از مینیمم محلی فراهم می‌کند.

۴-۷- تمرین‌ها

۴-۱ عملکرد الگوریتم جستجوی A* در مسئله رفتن به بخارست از لوگوج که در آن از ابتکار "فاصله خط مستقیم" استفاده شده‌است را زیر نظر بگیرید. دنباله‌ای از گره‌ها که الگوریتم به آن‌ها برخورد می‌کند و مقدار f و g و h را برای هر گره مشخص کنید.

۴-۲ الگوریتم "ابتکار مسیر" نوعی از جستجوی بهترین-اولین است که در آن تابع هدف برابر $f(n) = (2 - w)\, g(n) + wh(n)$ است. به ازای چه مقادیری از w این الگوریتم لزوما بهینه است؟ (می‌توانید h را قابل پذیرش در نظر بگیرید). در هریک از این حالات الگوریتم چه نوع جستجویی انجام می‌دهد، $w = 0$، $w = 1$، $w = 2$؟

۴-۳ هریک از موارد زیر را اثبات کنید:

جستجوی سطح-اول حالت خاصی از جستجوی هزینه-یکنواخت است.

جستجوی سطح-اول، عمق-اول، هزینه-یکنواخت حالت‌های خاصی از جستجوی بهترین-اولین هستند.

جستجوی هزینه- یکنواخت حالت خاصی از جستجوی A* است.

۴-۴ فضای حالتی ابداع کنید که در آن A* در GRAPH-SEARCH راه‌حلی غیربهینه برمی‌گرداند (با تابع h(n)ای که قابل‌پذیرش اما ناسازگار است).

۴-۵ در متن مشاهده کردید که ابتکار "فاصله خط مستقیم"، جستجوی بهترین- اولین حریصانه را در مسئله رفتن از آیسی به فاگاراس دچار اشتباه می‌کند. درحالی که این ابتکار در مسئله معکوس آن یعنی رفتن از فاگاراس به آیسی بسیار عالی عمل می‌کند. آیا مسئله‌هایی وجود دارد که ابتکار، الگوریتم را در هر دو جهت به اشتباه بیاندازد؟

۴-۶ برای مسئله ۸- پازل تابع ابتکاری ابداع کنید که گاهی بیش‌تر از مقدار واقعی تخمین بزند، و نشان دهید که چگونه در یک مسئله خاص منجر به یک جواب غیربهینه می‌شود. اثبات کنید، اگر h هیچگاه از مقدار c بیش‌تر از واقعیت تخمین نزند، آنگاه A* استفاده کننده از این h ، جوابی را برمی‌گرداند که بیش‌تر از c با جواب بهینه اختلاف ندارد.

۴-۷ اثبات کنید که اگر یک ابتکار سازگار باشد، آنگاه باید قابل پذیرش نیز باشد. یک ابتکار قابل پذیرشی بسازید که سازگار نباشد.

---

[۱]-Exploration

[۲]- Updating Heuristic Estimates

۴-۸ مسئله فروشنده دوره‌گرد (TSP) از طریق ابتکار درخت پوشای کمینه (MST) قابل حل است (که در آن هزینه اتمام یک دور تخمین زده می‌شود). هزینه MST مجموعه‌ای از شهرها، کوچکترین حاصل‌جمع هزینه‌های لینک‌های درختی است که همه‌ی شهرها را به یکدیگر مرتبط می‌کند.

الف) نشان دهید که از نسخه ساده شده مسئله TSP چگونه یک ابتکار بدست می‌آید؟

ب) نشان دهید که ابتکار MST بر "فاصله خط مستقیم" برتری دارد.

ج) برنامه تولیدکننده مسئله‌ای بنویسید که TSPای تولید کند که در آن شهرها به صورت نقطه‌هایی تصادفی در مربع واحد نمایش داده می‌شوند.

د) الگوریتم کارایی برای ساخت MST بیابید و از آن بهمراه یک الگوریتم جستجوی قابل‌پذیرش برای حل نمونه‌هایی از TSP استفاده کنید.

۴-۹ در متن معمای ساده شده ۸- پازل را بگونه‌ای تعریف کردیم که یک کاشی می‌تواند از خانه A به خانه B برود در صورتی که B خالی باشد. راه حل دقیق این مسئله همان ابتکار Gasching است. توضیح دهید که چرا ابتکار Gasching حداقل به‌اندازه h_1(کاشی‌های در مکان نادرست) دقیق است. و مواردی را مثال بزنید که این مورد حتی از h_1 و h_2(فاصله Manhattan) دقیق‌تر است. آیا می‌توانید راه حل کارایی برای محاسبه ابتکار Gasching ارائه دهید؟

۴-۱۰ ما دو ابتکار ساده برای مسئله ۸- پازل ارائه کردیم: فاصله Manhattan و کاشی‌هایی که در مکان صحیح خود قرار نداشتند. محققان روش‌های دیگر ارائه نموده و ادعا کرده‌اند که روش‌های آنان بهتر است(برای مثال نیلسون[۱] (۱۹۷۱) موستو[۲] و پریدیتس[۳] (۱۹۸۹) و هانسون[۴](۱۹۹۲)) این ادعاها را پیاده‌سازی و تست کرده و کارایی الگوریتم‌های حاصل را مقایسه کنید.

۴-۱۱ الگوریتمی که از هریک از حالت‌های خاص زیر نتیجه می‌شود را نام ببرید:

الف) جستجوی پرتوی محلی با k=۱.

ب) جستجوی پرتوی محلی با یک حالت اولیه، و بدون محدودیت برروی تعداد حالت‌های نگهداری شده.

ج) الگوریتم شبیه‌سازی ذوب با T=۰(و با حذف کردن تست پایان).

د) الگوریتم ژنتیک با اندازه جمعیت N=۱.

۴-۱۲ گاهی اوقات هیچ تابع ارزیابی مناسبی برای یک مسئله وجود ندارد اما یک تکنیک مقایسه‌ای مناسب وجود دارد: راهی برای اینکه بگوییم که آیا یک گره بهتر از گره دیگر است(بدون اینکه مقداری به هریک از آنان نسبت دهیم). نشان دهید که این برای اجرای الگوریتم جستجوی بهترین- اولین کافی است. آیا تشابهی با A* دارد؟

۴-۱۳ رابطه بین پیچیدگی زمانی الگوریتم LRTA* و پیچیدگی فضای حافظه آن را بیان کنید.

۴-۱۴ فرض کنید یک عامل درون محیط ۳×۳ یک مارپیچ است (همچون شکل ۴-۱۸). عامل می‌داند که مکان اولیه (۱,۱) است و هدف در (۳,۳) قرار دارد. از طرفی می‌داند که ۴ کنش بالا، پایین، چپ، راست اثرهای طبیعی خود را دارند مگر اینکه جلوی عامل دیوار باشد. عامل نمی‌داند که دیوارهای داخلی کجا

۱- Nilson
۲- Mostow
۳- Prieditis
۴- Hansson

قرار دارند. در هر حالت داده شده، عامل مجموعه کنش‌های مجاز را درک می‌کند (دریافت می‌کند) همچنین می‌تواند تشخیص دهد که این حالت را قبلا دیده است یا خیر.

الف) توضیح دهید که این مسئله جستجوی برخط، چگونه می‌تواند به‌صورت یک جستجوی برون خط در "فضای حالت باور[۱]" تعبیر شود که در آن حالت باور اولیه همه‌ی وضعیت‌های ممکن محیط را شامل می‌شود.

ب) چند ادراک متمایز در حالت اولیه وجود دارد؟

ج) چند ادراک اولیه طرح احتمالی[۲] را برای این مسئله تشریح کنید. اندازه کل نقشه چقدر است؟

توجه کنید که این نقشه اقتضایی راه‌حلی برای هر محیط ممکنی است که با توصیف خاصی متناسب است. لذا، یک در میان کردن جست‌وجو و اجرا، حتی در محیط‌های ناشناخته نیز الزامی نیست.

۱۵-۴ در این تمرین به دنبال استفاده روش‌های جستجوی محلی در حل TSPها می‌گردیم.

الف) رویکرد الگوریتم تپه‌نوردی را برای حل مسئله TSP به‌کار گیرید. نتایج را با راه‌حل بهینه حاصل از الگوریتم A* و ابتکار MST (تمرین ۴-۸) مقایسه کنید.

ب) رویکرد الگوریتم ژنتیک را برای حل مسئله TSP به کار گیرید. نتایج را با رویکردهای دیگر مقایسه کنید.

۱۶-۴ تعداد زیادی از نمونه مسائل ۸- پازل و ۸- وزیر را تولید و با الگوریتم تپه‌نوردی (پرشیب‌ترین و اولین انتخاب)، الگوریتم تپه‌نوردی با شروع مجدد تصادفی، الگوریتم شبیه‌سازی گداختگی حل کنید. هزینه جستجوی هریک از آن‌ها و درصد مسائل حل شده را ارزیابی کرده و آن را درقیاس با هزینه راه‌حل بهینه برروی نمودار بکشید. نتایج را مقایسه کنید.

۱۷-۴ در این تمرین، الگوریتم تپه‌نوردی را در کاربرد رهیابی ربات مورد بررسی قرار خواهیم داد(به عنوان مثال، با استفاده از محیط شکل ۳-۲۲).

الف) تمرین ۳-۱۶ را با استفاده از الگوریتم تپه‌نوردی تکرار کنید. آیا عامل شما هیچگاه در مینیمم محلی گیر می‌کند؟ آیا امکان گیر کردن آن در موانع محدب وجود دارد؟

ب) محیط چندضلعی غیرمحدبی بسازید که عامل در آن گیرکند.

ج) الگوریتم تپه‌نوردی را به‌گونه‌ای تغییر دهید که به جای انجام جستجوی عمق-۱ (برای تصمیم‌گیری در مورد نقل مکانش) از جستجوی عمق-k استفاده کند. درحقیقت این الگوریتم باید بهترین مسیر k- مرحله را پیدا و یک گام برروی آن پیشرفته و فرآیند را تکرار کند.

د) آیا kای وجود دارد که الگوریتم جدید برمبنای آن همواره از مینیمم محلی بگریزد؟

ه) توضیح دهید که در این مورد چگونه LRTA* عامل را قادر به فرار از مینیمم محلی می‌کند.

۱۸-۴ کارایی A* و RBFS را روی مجموعه‌ای از مسائل تصادفی تولید شده در ۸- پازل (با فاصله Manhattan) و TSP (با MST) مقایسه کنید. درمورد نتایج بحث کنید. زمانی که یک عدد تصادفی کوچک به مقادیر ابتکار در ۸- پازل اضافه می‌شود چه اتفاقی برروی کارایی RBFS می‌افتد؟

۱-Belief State Space

۲- Contingency Plan

تست‌های طبقه‌بندی‌شده فصل چهارم

۱- روش جستجوی *A، تحت چه شرایطی یافتن پاسخ بهینه را تضمین می‌کند؟ (کامپیوتر ۸۱)

۱) اصولا روشهای ابتکاری(Heuristic) از جمله *A قادر به یافتن پاسخ بهینه نیستند.

۲) شرایط لازم برای اینکه روش *A یافتن پاسخ بهینه را تصمین کند، به دامنه‌ی مسئله بستگی دارد.

۳) درصورتی که تابع ابتکاری (Heuristic Function) مورد استفاده، فاصله‌ی وضعیت‌های مختلف تا وضعیت هدف را، هرگز بیش‌تر از مقدار واقعی تخمین نزند.

۴) درصورتی که تابع ابتکاری (Heuristic Function) مورد استفاده، فاصله‌ی وضعیت‌های مختلف تا وضعیت هدف را حداکثر به اندازه‌ی مقدار کوچک دلتا و بیشتر از مقدار واقعی تخمین بزند.

۲- شرط پذیرشی *(Admissible)* بودن یک الگوریتم برای یافتن جواب مساله کدام است؟ (کامپیوتر ۸۱)

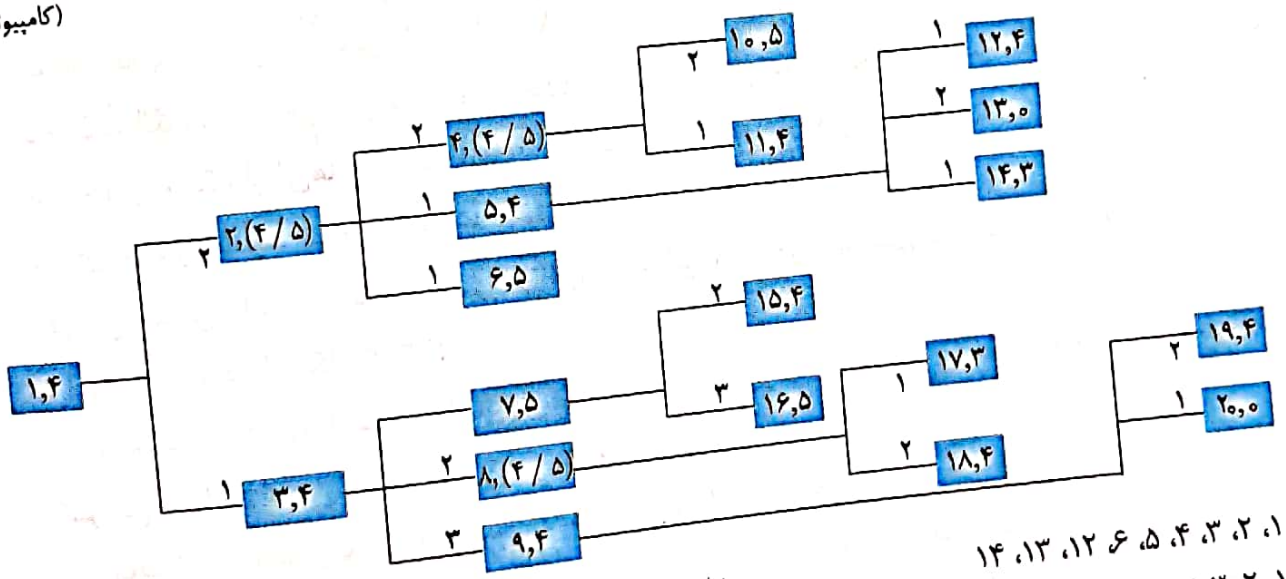
۱) $\exists n\ h(n) \le h^*(n),\ g(n) \ge g^*(n)$

۲) $\forall n\ h(n) \le h^*(n),\ g(n) \ge g^*(n)$

۳) $\exists n\ h(n) \ge h^*(n),\ g(n) \ge g^*(n)$

۴) $\forall n\ h(n) \le h^*(n),\ g(n) \le g^*(n)$

۳- در درخت تصمیم‌گیری زیر با استفاده از جستجوی *A کدام گزینه شماره گره‌های مورد بررسی را مشخص می‌کند؟ توجه کنید که هزینه هر گره در کنار شماره آن و هزینه هر شاخه روی آن نوشته است. (در هرگره اولین شماره گره و دومین عدد هزینه می‌باشد). (کامپیوتر ۸۱)



۱) ۱، ۲، ۳، ۴، ۵، ۶، ۱۲، ۱۳، ۱۴

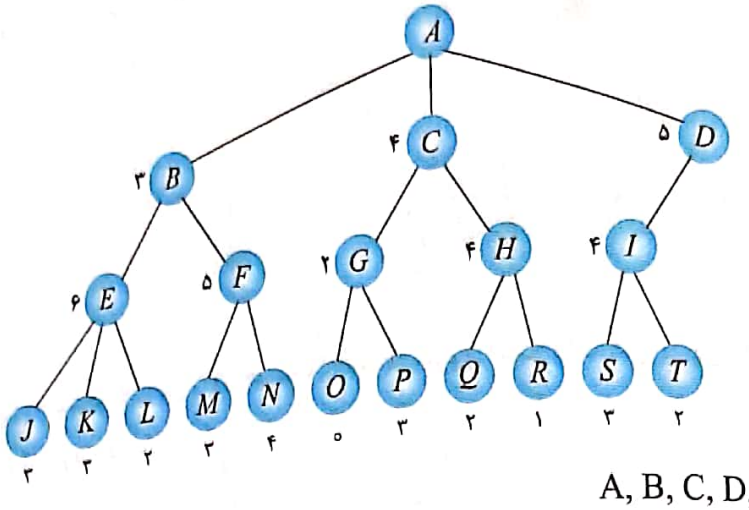
۲) ۱، ۲، ۳، ۷، ۸، ۹، ۱۹، ۲۰

۳) ۱، ۲، ۳، ۷، ۸، ۹، ۱۷، ۱۸، ۱۹، ۲۰

۴) ۱، ۲، ۳، ۷، ۸، ۹، ۴، ۵، ۶، ۱۲، ۱۳، ۱۴

۴- در درخت جستجوی زیر به شرطی که گره O گره هدف باشد. براساس الگوریتم جستجوی Best First ترتیب دیدن گره‌ها کدام است؟

(کامپیوتر ۸۱)



۱) A, B, C, D, E, F, G, H, O

۲) A, B, C, D, E, F, G, O

۳) A, B, E, J, K, L, F, M, N, C, G, O

۴) A, B, C, D, E, F, G, H, I, J, K, L, M, N, O

۵- نقطه‌ی ضعف روش جستجوی IDA^* (Iterative Deepening A*) در چیست؟

(کامپیوتر ۸۲)

۱) کامل نبودن ۲) دوباره‌ی کاری ۳) کارایی پایین ۴) مصرف حافظه‌ی زیاد

۶- می‌خواهیم با استفاده از روش جستجوی A* پاسخ بهینه‌ی (Optimal) مساله‌ای را بیابیم، با فرض اینکه هریک از سه تابع ابتکاری h۱، h۲، h۳ برای این منظور قابل استفاده باشد، کدام یک از توابع ترکیبی زیر نیز برای یافتن حل بهینه‌ی مساله قابل استفاده خواهد بود؟

(کامپیوتر ۸۳)

۱) $h_1 + h_2 + h_3$ ۲) $\frac{h_1 + h_2 + h_3}{3}$

۳) $h_1.h_2.h_3$ ۴) $\sqrt{h_1.h_2.h_3}$

۷- کدام‌یک از موارد زیر درخصوص روش جستجوی RTA^* $(Real\ time\ A^*)$ در مقایسه با روش A^* صحیح‌تر است؟

(کامپیوتر ۸۳)

۱) RTA^* اغلب تمایل بیشتری به ادامه‌ی مسیر جاری دارد.

۲) RTA^* همواره مسیرهای کوتاه‌تری را می‌یابد.

۳) RTA^* اغلب تمایل کمتری به ادامه مسیر جاری دارد.

۴) RTA^* همواره مسیرهای طولانی‌تری را می‌یابد.

۸- فرض کنید می‌خواهیم الگوریتم A^* را برای جستجو بکار ببریم و سه هیوریستیک H_1، H_2، H_3 موجودند که همگی قابل‌پذیرش هستند و برای تمام وضعیت‌ها داریم: $H_3 > H_2 > H_1$ کدامیک از عبارات زیر درست است؟

(کامپیوتر ۸۴)

۱) مسیر بهینه فقط از بکارگیری H۲ حاصل می‌شود اما H۱ از H۳ بهتر است.

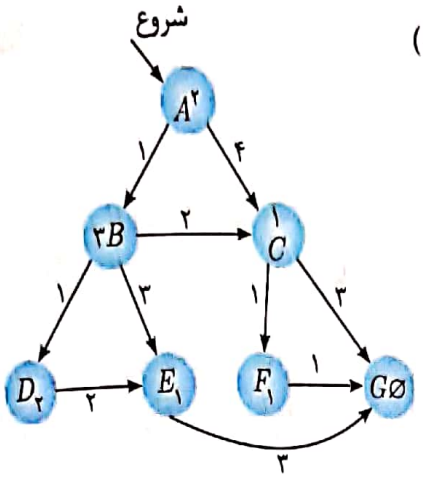
۲) با هرکدام از هیوریستیک‌های فوق مسیر بهینه حاصل می‌شود اما H۲ مسیر ارزانتری را پیدا می‌کند.

۳) با هرکدام از هیوریستیک‌های فوق مسیر بهینه حاصل می‌شود اما H۲ آن را با کمترین بسطها پیدا می‌کنند.

۴) راه‌حل‌های حاصل از این هیوریستیک‌ها از نظر بهینگی به همان ترتیب H۳>H۲>H۱ هستند.

۹- مسیر یافت‌شده توسط الگوریتم جستجوی A^* برای گراف مقابل چیست؟ (اعداد روی یال‌ها، هزینه واقعی هریال و اعداد داخل دایره‌ها هزینه تخمینی آن گره تا هدف است)

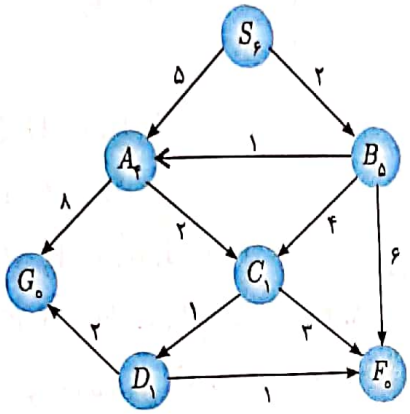
(کامپیوتر ۸۴)



۱) ABCFG
۲) ABCG
۳) ACG
۴) ABEG

۱۰- در گراف مقابل حاصل جستجو با روش A* چیست؟ S نقطه شروع است و اعداد روی یال‌ها هزینه واقعی و اعداد داخل دایره‌ها مقدار h گره مورد نظر است؟

(کامپیوتر ۸۵)



۱) SBF
۲) SBAG
۳) SBCDG
۴) SBACDF

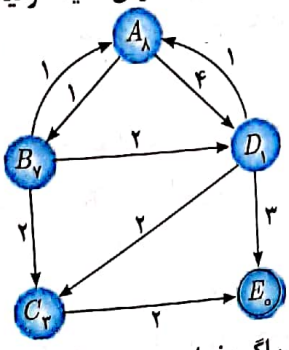
۱۱- پیچیدگی زمانی در جستجوی با تعمیق تکراری (iterative deepening) به کدام‌یک از عوامل زیر بستگی دارد؟

(کامپیوتر ۸۶)

۱) بیشترین عمق درخت
۲) سایز فضای حالت
۳) تابع مکاشفه‌ای انتخاب‌شده
۴) عمق کم عمیق‌ترین گره هدف

۱۲- گراف مقابل را درنظر بگیرید. در این گراف هزینه هر حرکت برروی یال مربوطه و مقدار تابع مکاشفه‌ای h (هزینه تخمینی هر گره تا هدف) داخل دایره نوشته شده است. اگر E گره هدف باشد، کدام یک از روش‌های جستجوی زیر برای این فضای حالت مسیر بهینه $ABCE$ را پیدا می‌کند؟ (در همه روش‌ها فرض کنید ترتیب تولید فرزندان یک گره به ترتیب حروف الفباست.)

(کامپیوتر ۸۶)



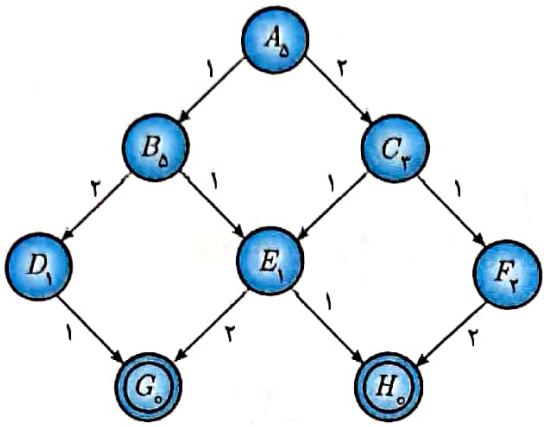
۱) جستجوی A*
۲) جستجوی عرض اول (Breath-first)
۳) جستجوی uniform cost
۴) جستجوی عمق اول (Depth-first)

۱۳- در مقایسه بین روش‌های مختلف جستجو از نظر حافظه‌بری، اگر بخواهیم روش‌ها را از پیچیده‌ترین تا ساده‌ترین (از نظر پیچیدگی حافظه‌ای) مرتب نمائیم، کدام گزینه در اغلب موارد صحیح است؟

(کامپیوتر ۸۷)

۱) RBFS → breadth First → SMA^* → A^*
۲) RBFS → breadth First → A^* → SMA^*
۳) breadthFirst → A^* → RBFS → SMA^*
۴) breadthFirst → A^* → SMA^* → RBFS

۱۴- در گراف مقابل اگر در جستجو با الگوریتم A* تست هدف یک بار در لحظه تولید و بار دیگر در لحظه بسط صورت گیرد به ترتیب چه مسیری یافت خواهد شد؟(اعداد روی یال‌ها هزینه مسیر و اعداد داخل گره‌ها هزینه تخمینی گره تا هدف است) ترتیب ملاقات فرزندان هر گره به ترتیب حروف الفباست. (کامپیوتر ۸۸)

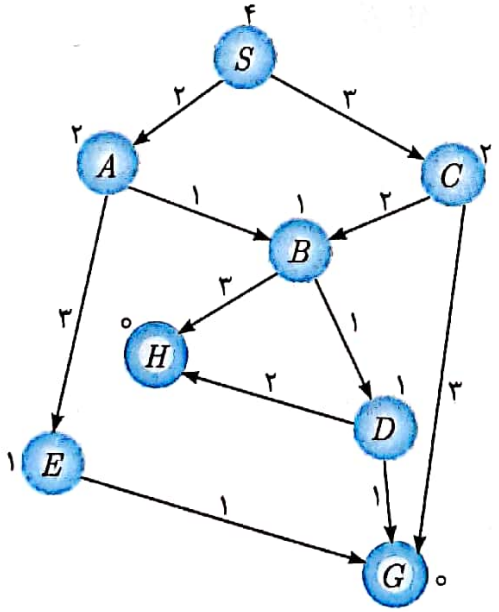


۱) (با تست در لحظه تولید) ACEG – (با تست در لحظه بسط) ACEH
۲) (با تست در لحظه تولید) ABEH – (با تست در لحظه بسط) ABEH
۳) (با تست در لحظه تولید) ACEH – (با تست در لحظه بسط) ABEH
۴) (با تست در لحظه تولید) ACEG – (با تست در لحظه بسط) ACEG

۱۵- فرض کنید الگوریتم *Local Beam Search* با $k = ۱$ اجرا می شود. این جستجو معادل کدام یک از جستجوهای زیر است؟ (کامپیوتر ۹۱)

۱) *Hill Climbing*
۲) *Simulated Annealing*
۳) *Genetic Algorithm*
۴) *Constraint Satisfaction*

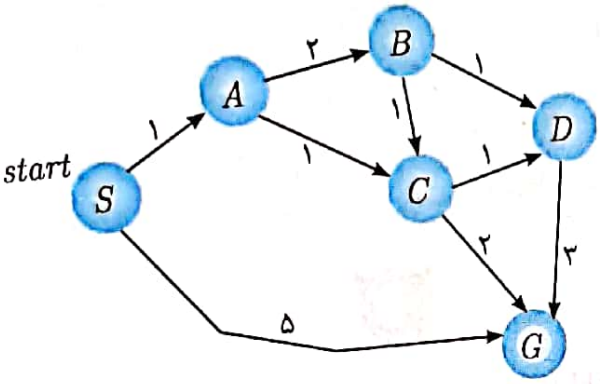
۱۶- در گراف زیر، H و G گره‌های هدف و S گره شروع است. کدام یک از موارد زیر به‌ترتیب از چپ به راست، گره‌های ملاقات شده توسط جستجوی A^* را نشان می‌دهد. هزینه هر انتقال از یک گره به گره دیگر روی یال واصل و هزینه‌ی تخمینی هر گره تا هدف در کنار آن گره نوشته شده است. در شرایط مساوی به گره‌ای که زودتر تولید شده است. اولویت دهید. (کامپیوتر ۹۲)



۱) *SABCDG*
۲) *SACBDH*
۳) *SABDH*
۴) *SABDG*

۱۷- در گراف جستجوی زیر، S گره‌ی شروع و G گره‌ی هدف است. پیمایش گره‌ها در شرایط مساوی براساس ترتیب الفبا صورت می‌گیرد. براساس دو تابع هیوریستیک نشان داده شده در جدول زیر، کدام گزاره صحیح است؟

(کامپیوتر ۹۲)



گره	h_1	h_2
S	۳	۴
A	۳	۲
B	۴	۳
C	۲	۱
D	۳	۱
G	۰	۰

۱) هر دو تابع هم *admissible* و هم *consistent* هستند.

۲) هر دو تابع *admissible* هستند. اما فقط h_1، *consistent* است.

۳) هیچ‌یک از دو تابع *admissible* نیست. اما فقط h_2، *consistent* است.

۴) هیچ‌یک از دو تابع *consistent* نیست. اما فقط h_2، *admissible* است.

پاسخنامه تشریحی فصل چهارم

۱) گزینه ۳ درست است.

اثبات:

فرض کنید هدف غیربهینه G در لیست گره‌ها قرار دارد. از طرفی مقدار جواب بهینه مسئله برابر C است. از آنجا که h(G) = ۰ (برای تمامی هدف‌ها صفر است) و G یک هدف غیربهینه است داریم:

$$f(G) = g(G) + h(G) = g(G) > C$$

حال فرض کنید گره n، گره‌ای در مسیر بهینه مسئله و در لیست گره‌ها قرار دارد. اگر h(n) تابعی قابل پذیرش (admissible) باشد، آنگاه:

$$f(n) = g(n) + h(n) <= C$$

بنابراین:

$$f(n) <= C < f(G)$$

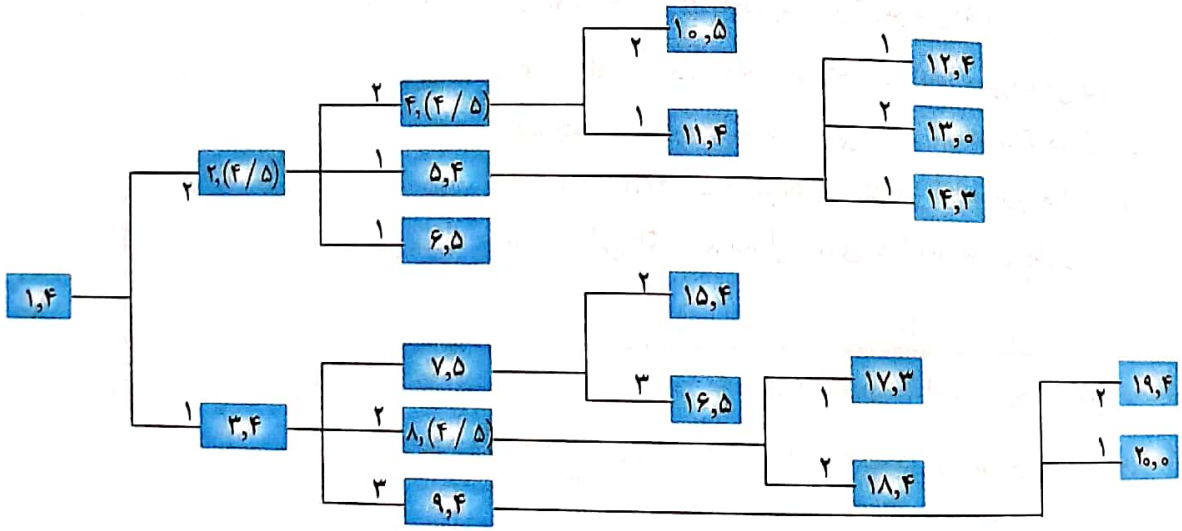
بنابراین g بسط داده نمی‌شود و A* جواب بهینه را برمی‌گرداند.

۲) گزینه ۴ درست است.

نکته ۱: شرط بهینه بودن الگوریتم A* برای درخت‌ها این است که تابع h(n) در آن admissible باشد. یعنی این تابع هیچگاه هزینه رسیدن تا هدف را بیش‌تر از مقدار واقعی تخمین نزند. و یعنی اینکه برای تمام n ها، f(n)<= f*(n) است. لازم بذکر است که در عمل g(n) = g*(n) است و عملا شرط مذکور فقط روی h(n)<=h*(n) قابل اعمال است.

۳) گزینه ۴ درست است.

درخت جستجوی A* در شکل زیر نشان داده شده است.



۴) گزینه ۱ درست است.

الگوریتم به شکل زیر عمل می‌کند:

ابتدا گره ریشه بسط داده می‌شود.

(B, ۳) – (C, ۴) – (D, ۵)

گره B که کمترین هزینه را دارد انتخاب می‌شود.

(C, ۴) – (D, ۵) – (E, ۶) – (F, ۵)

گره C که کمترین هزینه را دارد انتخاب می‌شود.

(D, ۵) – (E, ۶) – (F, ۵) – (G, ۲) – (H, ۴)

گره G که کمترین هزینه را دارد انتخاب می‌شود.

بعد از بسط گره G به هدف که O است می‌رسیم.

۵) گزینه ۲ درست است.

نکته ۱: IDA^* برگرفته شده از الگوریتم A^* است. اما با این تفاوت که مقدار حافظه مصرفی در آن کاهش پیدا کرده است. ایده این است که (همچون الگوریتم تعمیق تکراری) روی هزینه کل محدودیت می‌گذاریم. در هر بار تکرار، تمامی گره‌هایی که درون محدوده هزینه ما هستند را بسط می‌دهیم. هنگامی که جستجو درون محدوده موردنظر تمام و یا به نتیجه نرسید، الگوریتم را با مقدار جدیدی برای محدوده تکرار می‌کنیم. این مقدار کوچکترین مقدار دور ریخته شده در تکرار قبلی خواهد بود.

۶) گزینه ۲ درست است.

برای اینکه بتوان از توابع ترکیبی استفاده کرد، باید تابع حاصل نیز همچنان شرایط admissible بودن را حفظ کند. از آنجا که هر یک از این ۳ تابع برای مسئله قابل استفاده بنابراین admissible هستند. بنابراین تابع حاصل باید از $\max(h_1, h_2, h_3)$ کوچکتر باشد. در میان گزینه‌ها تنها گزینه ۲ این ویژگی را داراست. زیرا میانگین ۳ عدد همواره کوچکتر از بزرگترین آن هاست.

۷) گزینه ۱ درست است.

توضیح: در الگوریتم $RTA\left(\text{Real} - time\ A^*\right)$ تعریف تابع $g(n)$ به‌صورت زیر تغییر می‌کند:

در هر لحظه یک گره به عنوان گره جاری وجود دارد که در واقع در حال حاضر جستجو در آنجا صورت می‌گیرد. $g(n)$ برابر است با فاصله هر گره n تا گره جاری. در ابتدای جستجو همان گره اولیه است ولی با هر حرکتی، گره جاری نیز تغییر می‌کند. بنابراین اگر در یک جهت حرکت صورت گیرد، عملاً $g(n)$ تمام گره‌هایی که در آن مسیر قرار دارند کاهش می‌یابد و تمام گره‌های سایر مسیرها افزایش می‌یابد و بنابراین این الگوریتم تمایل زیادی به ادامه مسیر جاری دارد.

۸) گزینه ۳ درست است.

نکته ۱: هرچه تابع هیوریستیک ما بزرگتر باشد (در عین حالی که admissible است) یعنی تابع دقیق‌تری است و به واقعیت نزدیک‌تر است. مزیت دقیق بودن تابع هیوریستیک این است که در زمان کمتر و با بسط گره‌های کمتری جواب بهینه را می‌یابد.

نکته ۲: شرط الگوریتم A* برای پیدا کردن جواب بهینه admissible بودن تابع هیوریستیک است. بنابراین هرسه تابع مورد نظر جواب بهینه را برمی‌گردانند.

۹) گزینه ۱ درست است.

ابتدا ریشه انتخاب شده و فرزندان آن بسط داده می‌شوند.

(B, ۴) – (C, ۵)

گره B انتخاب می‌شود. و فرزندان آن بسط داده می‌شوند.

(C, ۴) – (D, ۴) – (E, ۵)

دقت کنید در این حالت، هزینه برای گره C تغییر کرد، زیر از مسیر جدید به مقدار کوچکتری رسیدیم.

گره C انتخاب، و فرزندان آن بسط داده می‌شوند.

(D, ۴) – (E, ۵) – (F, ۵) – (G, ۶)

دقت کنید در این حالت با اینکه گره G، گره هدف است، اما چون گره دیگری با هزینه کمتر وجود دارد، آن را انتخاب نخواهیم کرد.

گره D انتخاب و فرزندان آن بسط داده می‌شوند.

(E, ۵) – (F, ۵) – (G, ۶)

گره E انتخاب می‌شود.

(F, ۵) – (G, ۶) – (G, ۷)

گره F انتخاب می‌شود.

(G, ۵) – (G, ۶) – (G, ۷)

بنابراین (G,۵) بسط داده می‌شود.

بنابراین مسیر ABCFG است.

راه دیگر: با توجه به اینکه تابع h در شکل، تابعی admissible است، بنابراین کوتاه‌ترین مسیر انتخاب می‌شود.

هزینه گزینه ۱: ۵

هزینه گزینه۲: ۷

هزینه گزینه۳: ۶

هزینه گزینه۴: ۷

۱۰) گزینه ۴ درست است.

ابتدا گره ریشه انتخاب و فرزندان آن بسط داده می‌شوند.

(A, ۹) – (B-۷)

گره با کمترین هزینه، یعنی گره B انتخاب می‌شود و فرزندان آن بسط داده می‌شوند.

(A, ۷) – (C, ۷) – (F, ۸)

گره با کمترین هزینه، یعنی گره A انتخاب می‌شود و فرزندان آن بسط داده می‌شوند.

(F, ۸) – (C, ۶) – (G, ۱۱)

گره با کمترین هزینه، یعنی گره C انتخاب می‌شود و فرزندان آن بسط داده می‌شوند.

(F, ۸) – (G, ۱۱) – (D, ۷)

گره با کمترین هزینه، یعنی گره D انتخاب می‌شود و فرزندان آن بسط داده می‌شوند.

(F, ۸) – (G, ۱۱) – (F, ۷) – (G, ۸)

گره با کمترین هزینه، یعنی گره F انتخاب می‌شود که هدف است.

۱۱) گزینه ۴ درست است.

تعداد گره‌های تولید شده توسط الگوریتم جستجوی تعمیق تکراری برابر است با:

$$N(IDS) = (d)\,d + (d-1)\,b^2 + \cdots + (1)\,b^d$$

که پیچیدگی زمانی $O\left(b^d\right)$ به ما می‌دهد.

بنابراین پیچیدگی زمانی الگوریتم تعمیق تکراری به فاکتور انشعاب و عمق کم‌عمق‌ترین گره هدف بستگی دارد.

۱۲) گزینه ۳ درست است.

گزینه ۱ نادرست است. با نگاه به مقادیر موجود برای تابع مکاشفه‌ای در خواهیم یافت که این تابع admissible نیست، بنابراین جواب بهینه را نخواهد یافت. برای مثال، مقدار روی گره A برابر ۸ است در حالی که هزینه واقعی رفتن به هدف بیشتر از آن است.

بد نیست این الگوریتم را پیداسازی و ادعای مطرح شده را امتحان کنیم:

ابتدا گره ریشه انتخاب و فرزندان آن بسط داده می‌شوند.

(B, ۸) – (D, ۵)

(B, ۸) – (C, ۹) – (E, ۷)

گره E به عنوان گره هدف انتخاب می‌شود که پاسخی بهینه نیست.

گزینه ۲ نادرست است. ترتیب تولید گره‌ها در این الگوریتم به شکل زیر است:

A

گره ریشه انتخاب می‌شود و فرزندان آن بسط داده می‌شوند.

B – D

گره B انتخاب و فرزندان آن بسط داده می‌شوند.

D – A – C - D

گره D انتخاب و فرزندان آن بسط داده می‌شوند.

A – C – D – A – C – E

در این الگوریتم اولین گره هدفی که بسط داده شود، انتخاب می‌شود. بدین معنی که تمامی هدف‌های دیگر انتهای صف اضافه می‌شوند، بنابراین نزدیک‌ترین هدف به ابتدای صف به عنوان هدف انتخاب شده که در این جا از مسیر A-D-E عبور می‌کند.

گزینه ۴ نادرست است.

این الگوریتم در حلقه زیر گیر می‌کند. A-B-A-B

۱۳) گزینه ۲ درست است.

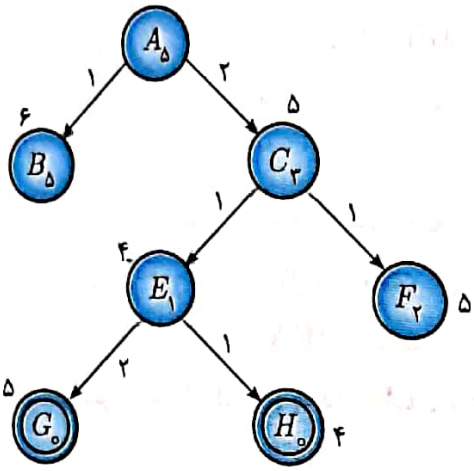
هزینه فضای الگوریتم BFS برابر است با: $O(b^{d+1})$

هزینه فضای الگوریتم A* تمامی گره‌هایی که $f(n)<C^*$ است را بسط می‌دهد. بنابراین تعدادی از گره‌ها را هرس می‌کند اما همچنان پیچیدگی فضای حافظه آن به صورت نمایی $O(b^d)$ است.

الگوریتم RBFS دارای پیچیدگی حافظه خطی است.

هزینه فضای حافظه الگوریتم SMA* به اندازه مسیر کم‌عمق‌ترین هدف است.

۱۴) گزینه ۱ درست است.



درخت بسط A* در شکل بالا نشان داده شده است. اگر هنگام تولید، هدف را تست کنیم دیگر گره H تولید نمی‌شود. اما اگر هنگام بسط، هدف را تست کنیم آنگاه گره H که کمترین هزینه را دارد انتخاب می‌شود.

۱۵) گزینه ۱ درست است.

عملا local beam search ، K عدد تپه نوردی همزمان است.

۱۶) گزینه ۱ درست است.

با اجرای الگوریتم A* براحتی دیده میشود که گره‌ها به ترتیب صعودی مقدار $f(n) = g(n) + h(n)$ ملاقات میشوند. در گزینه ۲، $f(C) > f(B)$ در حالیکه B بعد از C ملاقات شده است. در گزینه ی ۳، مقدار $f(C) = f(D)$ است و طبق سوال بایستی به ترتیب زمان تولید ملاقات گردد یعنی C باید قبل از D دیده شود. گزینه ۴ نیز مانند ۳ نادرست است.

۱۷) گزینه ۴ درست است.

با بررسی مقدار هر دو تابع و بررسی آنها با مقدار واقعی h^* براحتی دیده میشود که $h^*(B) = ۳$ در حالیکه $h۱(B) = ۴$ یعنی اینکه تابع $h۱$ قابل قبول نیست ولی $h۲$ قابل قبول است. تنها گزینه ای که مقدار admissible بودن را ذکر کرده است، گزینه ۴ است.

تست‌های تألیفی

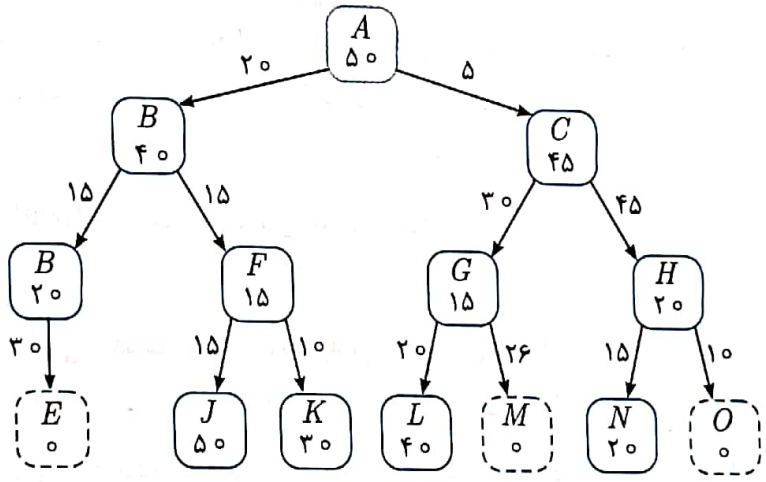
۱- مقدار حافظه مصرفی کدامیک از روش های غیر هوشمندانه زیر کمتر است؟

۱) Breadth First Search
۲) Uniform Cost
۳) Depth First Search
۴) Iterative Deepening search

۲- کدامیک از موارد زیر غلط است:

۱) هزینه زمانی الگوریتم bfs و dfs برابر است.
۲) هزینه زمانی iterative deepening بدلیل تکرار الگوریتم dfs بیش از dfs است.
۳) هزینه زمانی الگوریتم دو طرفه (Bidirectional) در صورت امکان اجرا، کمتر از dfs است.
۴) موارد الف و ب

۳- درخت جستجوی زیر داده شده است. گره A، وضعیت اولیه می‌باشد. وضعیت‌های جواب نیز با مربع‌های نقطه‌چین نشان شده‌اند. اعداد روی یال‌ها هزینه استفاده از آن مسیر (یال) را نشان می‌دهد. اعداد داخل هر گره نیز تخمین فاصله تا هدف را مشخص می‌کند. اگر برای جستجو از روش IDA* استفاده شود، میزان حد آستانه که جهت انتخاب گره ها، هنگام ورود به صف در نظر گرفته می‌شود پس از گرفتن مقدار اولیه ، چند بار تغییر می‌کند:



۱) ۰ بار ۲) ۱ بار ۳) ۲ بار ۴) ۳ بار

۴- اگر الگوریتم A* را برای مساله فوق بخواهیم اجرا کنیم ، کدام گزینه نادرست است:

۱) الگوریتم گارانتی می‌کند که به جواب برسد ولی به جواب بهینه نمی‌رسد.
۲) الگوریتم گارانتی می‌کند که به جواب بهینه برسد.
۳) الگوریتم گارانتی نمی‌کند که اصلا به جواب برسد.
۴) الگوریتم ممکن است که به جواب بهینه برسد.

۵- مسئله ۸-puzzle را در نظر بگیرید. یک database داریم که به ازای هر ترکیب چهار تایی از اعداد ۱-۸ ، میانگین هزینه حل مسئله برای آن چهار عدد تا جواب ، در آن ذخیره شده است. به عنوان مثال c(۱,۲,۳,۴) تعداد جابجایی مورد نیاز جهت جابجا کردن صرفا اعداد ۱و۲و۳و۴ تا محلهای جواب است. آنگاه بهترین تابع admissible از بین گزینه ها کدام است:

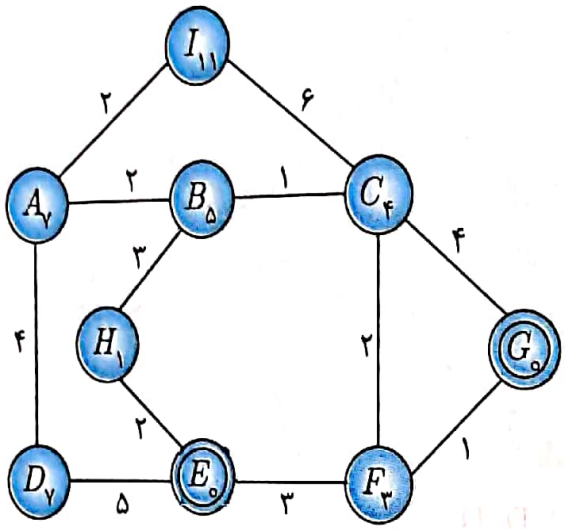
۱) H(n) = min(c(۱,۲,۳,۴),c(۱,۳,۵,۷))
۲) H(n) = sqrt(c(۱,۲,۳,۴)+c(۵,۶,۷,۸))
۳) H(n) = (c(۱,۲,۳,۴)+c(۵,۶,۷,۸))/۲
۴) نمی‌توان به قطعیت گفت کدام تابع بهتر است.

۶- با استفاده از الگوریتم A^* جواب بهینه‌ی مسأله‌ای با استفاده از سه تابع ابتکاری h_1, h_2, h_3 و روش جستجوی درختی قابل محاسبه است. اگر بخواهیم این مسأله را با جستجوی گرافی حل کنیم، کدام یک از توابع ابتکاری زیر یافتن جواب بهینه را تضمین می‌کنند؟

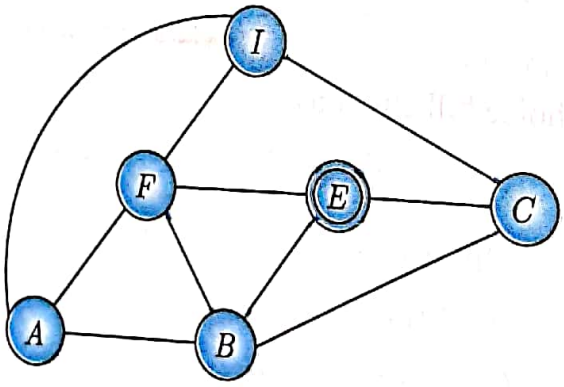
۱) $h_1 \times h_2 \times h_3$ ۲) $\sqrt{h_1 \times h_2 \times h_3}$ ۳) $\frac{h_1 + h_2 + h_3}{3}$ ۴) اطلاعات داده شده کافی نیست.

۷- مسیر یافته‌شده توسط الگوریتم A^* برای گراف زیر چیست؟ (اعداد روی یال‌ها هزینه‌ی واقعی هر یال و اعداد داخل دایره‌ها هزینه‌ی تخمینی آن گره تا هدف است. گره شروع I است.) مسیر ها از چپ به راست است.

۱) $I.A.B.H.E$
۲) $I.A.B.C.F.G$
۳) $I.A.B.E$
۴) $I.A.B.C.G$



۸- مسیر پاسخ جستجوی DFS در گراف زیر به ترتیب از چپ به راست چیست؟ گره‌ها به ترتیب حروف الفبا دیده می‌شوند.



۱) $I.A.C.F.B.E$ ۲) $I.A.B.C.E$ ۳) $I.A.B.E$ ۴) $I.C.E$

۹- فرض کنید تابع ارزیابی ما بصورت $f(n) = g(n) + h(n)$ باشد. g(n)- هزینه از ابتدا تا رسیدن به گره‌ی n، h(n)- هزینه تخمینی از گره n تا رسیدن به هدف (heuristic) کدامیک از گزینه های زیر درست است:

۱) برای بعضی از admissible heuristic ها، f میتواند در یک مسیر نزولی باشد، زیرا g صعودی است ولی h صعودی نیست

۲) از آنجایی که تابع g و heuristic (تابع h) صعودی هستند، پس تابع f در یک مسیر نمیتواند نزولی باشد.

۳) اگر تابع f صعودی باشد، آنگاه حتما heuristic (تابع h) قابل پذیرش admissible است.

۴) همه موارد درست است.

۱۰- چند گزینه زیر درست است:

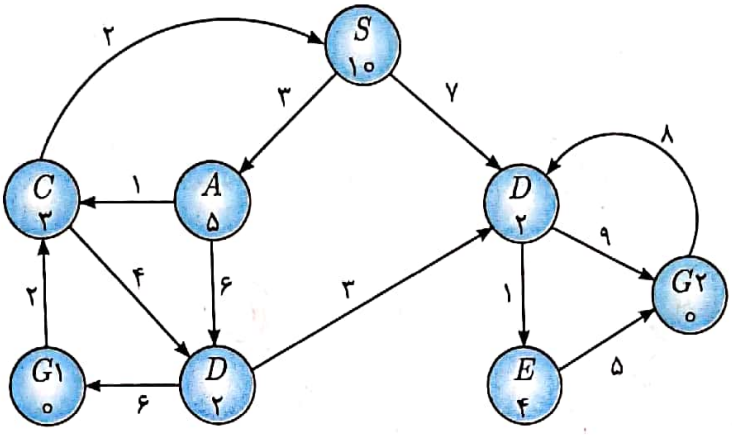
BFS کامل است. حتی اگر حرکت با هزینه صفر داشته باشیم.

امکان هرس کردن در game tree هایی که chance node دارند، وجود دارد.

تعداد node های بسط داده شده در DFS همواره بزرگتر یا مساوی تعداد node های بسط داده شده در A* (با یک admissible heuristic) است.

۱) ۰ گزینه ۲) ۱ گزینه ۳) ۲ گزینه ۴) ۳ گزینه

۱۱- در گراف زیر، اگر از الگوریتم A* استفاده شود، چه گره هایی به ترتیب گسترش می‌یابند تا هدف یافت شود؟ (اعداد روی یال‌ها هزینه‌ی واقعی هر یال و اعداد داخل دایره‌ها هزینه‌ی تخمینی آن گره تا هدف است.)



۱) S, A, B, C, D, E,
۲) S, A, C, D, D, B, E,
۳) S, A, B, E, G۲
۴) S, A, C, D, G۱

۱۲- رابطه بین simulated annealing و حالات مختلف hill climbing (در حالت کلی) را در نظر بگیرید:

وقتی T = ∞ ، simulated annealing میشود:

۱) Steepest ascent
۲) first-choice hill climbing
۳) random restart hill climbing
۴) هیچکدام

پاسخ تشریحی تست‌های تألیفی

۱) هزینه حافظه BFS برابر با $O(b^{d+1})$ است. هزینه حافظه روش $uniform\cos t$ برابر با $O(b^{\frac{C^*}{\varepsilon}})$ که C^* طول مسیر بهینه و ε حداقل هزینه یالها است. هزینه حافظه DFS برابر با $O(bm)$ است و هزینه حافظه IDS برابر با $O(bd)$ است که از همه کمتر است

۲) هزینه زمانی bfs برابر با $O(b^{d+1})$ است در حالیکه هزینه زمانی dfs برابر با O(bm) است. همچنین هزینه زمانی IDS برابر با dfs است. بنابراین گزینه د صحیح است.

۳) پس از اولیه مقدار اولیه به حد آستانه (مقدار ۵۰ که از گره A بدست می‌آید)، بر مبنای این مقدار گره های C، G وارد و خارج می‌شود سپس به مقدار بعدی که ۶۰ است افزایش می‌یابد که توسط آن گره های B و D و F وارد صف و سپس خارج می‌شوند . برای بار دوم مقدار آستانه به ۶۲ (که برابر با f(M) است) افزایش می‌یابد که در آنجا جستجو به اتمام می‌رسد. مقدار آستانه دوبار تغییر کرده است.

۴) با توجه به مقدار h(H)=۲۰ که از فاصله تا جواب که ۱۰ است، بیشتر است بنابراین تابع h ، همیشه Underestimate نیست و بنابراین optimal نیست. و گزینه ۲ نادرست است و پاسخ سؤال است.

۵) میانگین حسابی دو زیرمساله که اشتراک ندارند، در حالت کلی بزرگتر از دو حالت دیگر است و هر سه مورد نیز قابل پذیرش است.بنابراین گزینه ۳ درست است.

۶) گزینه ۴ درست است.

برای آنکه A* روی جستجوی گرافی درست باشد، بایستی یکنواخت (monotone) باشد. در حالیکه طبق مساله، هر سه تابع قابل پذیرش هستند یعنی under estimate هستند، هیچیک از گزینه های ۱و ۲و ۳ تضمین نمیکند که تابع نتیجه یکنوا باشد. بنابراین گزینه ۴ درست است.

۷) گزینه ۱ درست است.

الگوریتم A* در این مساله بدلیل h(I) = ۱۱ ، قابل پذیرش نیست و بنابراین نمی‌توان مستقیما بهترین مسیر از I به هدف را به عنوان جواب خروجی داد. ولی اجرای A* حتی با این تابع h (که Under estimate نیست)، باز هم جواب بهینه می‌دهد. شکل زیر مقدار صف برای A* با هزینه هر یک را نشان می‌دهد.(صف از چپ به راست است)

I	A	C	B	D	C	H	G
۱۱	۹	۱۰	۹	۱۳	۹	۸	۹

۸) گزینه ۲ درست است.

از هر گره به اولین فرزندی که ملاقات نشده است، حرکت می‌کند

۹) گزینه ۱ درست است.

گزینه ۲ و ۳ نادرست است. چون تابع h صعودی نیست.

۱۰) گزینه ۲ درست است.

فقط جمله اول درست است. هرس برای بازیهای شانسی فقط در صورتی معنا دارد که عدد ارزیابی آن دارای سقف و کف باشد. جمله سوم نیز واضح است که همواره درست نیست.

۱۱) گزینه ۲ درست است.

سوال گره هایی که بسط داده میشوند را خواسته است که ترتیب صحیح آن گزینه ۲ است.

۱۲) گزینه ۳ درست است.

اگر T بینهایت شود، احتمال انتخاب random بعدی برابر یک خواهد بود. یعنی هر جا به گره ای رسیدیم که بدتر از گره جاری باشد، حتما به آنجا میرویم .. این معادل یک random restart خواهد بود.

فصل پنجم

مسائل ارضاء محدودیت

در بخش ۳ و ۴ ایده‌ی حل مسئله از طریق جستجو در فضای حالت را بررسی کردیم. این حالت‌ها از طریق توابع ابتکاری با دامنه‌ی خاص ارزیابی می‌شود که آیا حالت مورد نظر هدف است یا خیر؟ از نقطه‌نظر الگوریتم های جستجو، هر حالت به صورت جعبه‌ی سیاهی است که ساختار داخلی آن چندان مشخص نیست و با استفاده از ساختمان داده‌ای اختیاری نمایش داده می‌شود که تنها توسط روال های خاصی (تابع پسین، تابع ابتکاری، تابع تست هدف) از مسئله قابل دسترسی است.

در این فصل به بررسی مسائلی به نام ارضاء محدودیت[۱] که در آن‌ها حالت‌ها و تابع هدف به صورت استاندارد، ساخت‌یافته و ساده نمایش داده شده‌اند، خواهیم پرداخت (بخش ۵-۱). از طرفی الگوریتم‌های جستجو به گونه‌ای تعریف می‌شوند که از ساختار حالت‌ها بهره برده و به جای استفاده از تابع ابتکاری خاص منظوره برای یک مسئله از تابعی همه‌منظوره برای حل تعداد بسیار زیادی از مسائل استفاده کنند(بخش ۵-۲ و ۵-۳). و از همه مهم‌تر نمایش تابع تست هدف به صورت استاندارد خود ساختار مسئله را بدرستی مشخص می‌کند (بخش ۵-۴) و این منجر به روش‌هایی برای تجزیه‌ی مسئله و فهم ارتباط درونی بین ساختار مسئله و دشواری حل آن می‌شود.

۵-۱- مسائل ارضاء محدودیت

یک مسئله ارضاء محدودیت (یا*CSP*) بصورت مجموعه‌ای ازمتغیرهای $X_1,X_2,...X_n$ و محدودیت‌های $C_1,C_2,...,C_m$ تعریف می‌شود که در آن هر متغیر X_i یک مقدار ناتهی از مقادیر موجود در دامنه D_i را می‌پذیرد و هر محدودیت C_i شامل مجموعه‌ای از متغیرهاست که مشخص‌کننده ترکیب‌های مجاز مقادیر به آن مجموعه است. حالت‌ها در مسئله نیز از طریق مقداردهی مقادیر به چند یا همه‌ی متغیرها تعریف می‌شوند.

$.\{X_i = v_i, X_j = v_j .....\}$

مقداردهی که در آن هیچ یک از محدودیت ها نقض نشود، مقداردهی سازگار یا قانونی نامیده می‌شود. مقداردهی کامل نیزمقداردهی‌ای است که در آن به همه متغیرها مقداری نظیر شده باشد. پاسخ مسئله در *CSP* مقداردهی کاملی است که تمامی محدودیت ها را ارضاء کرده باشد. اگرچه بعضی از *CSP* ها نیازمند جوابی هستند که تابع هدف[۲] را ماکزیمم کند.

حال می خواهیم ببینیم منظور از تعاریف بالا چیست؟ به جای رومانی به نقشه استرالیا در شکل ۵-۱ (a) دقت کنید که در آن حالت‌ها و مناطق استرالیا نشان داده شده است. وظیفه ما این است که هر منطقه را با سه رنگ

۱- Constraint Satisfaction

۲- Objective function

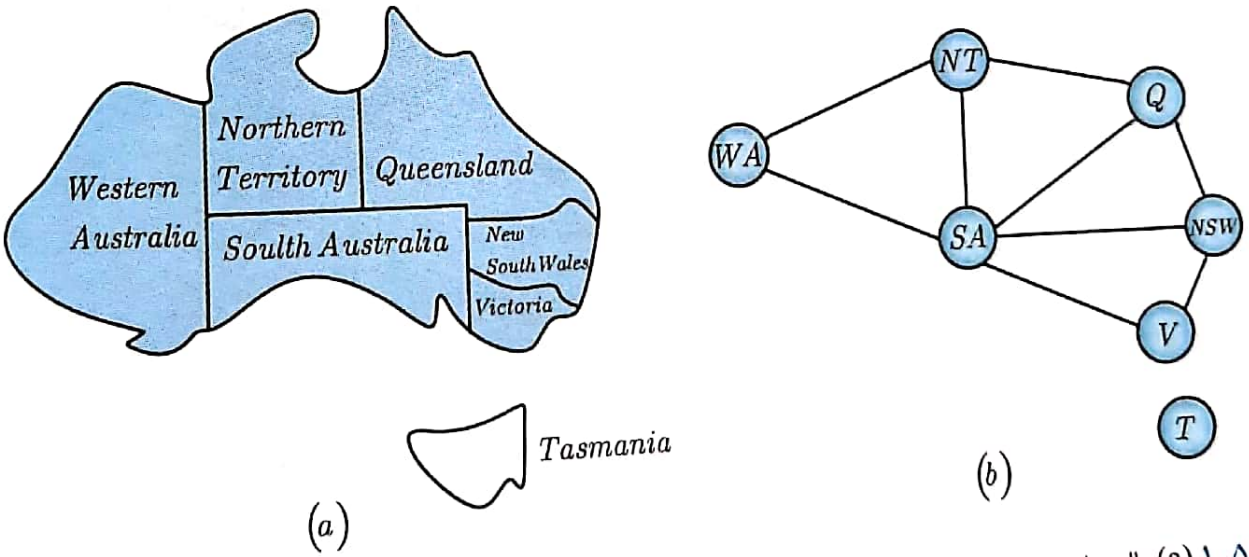
قرمز، سبز و آبی به گونه‌ای رنگ کنیم که هیچ دو همسایه‌ای دارای رنگ یکسان نباشند. برای این که این مسئله را به صورت CSP فرموله کنیم ابتدا هر منطقه را به عنوان یک متغیر تعریف می‌کنیم: WA,NT,Q,NSW,V,SA,T. دامنه هر متغیر مجموعه {آبی، سبز، قرمز} است. محدودیت‌ها نیز همان متفاوت بودن رنگ های همسایه‌ها هستند. به عنوان مثال ترکیب مجاز برای دو ناحیه NT و WA به صورت زیر است:

{(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}

(این محدودیت به طورخلاصه از طریق نامساوی WA≠NT قابل نمایش است (البته با این فرض که الگوریتم ارضاء محدودیت راهی برای ارزیابی این عبارت داشته باشد). راه حل‌های زیادی برای این مسئله وجود دارد. مثل:

{WA=red,NT=green,Q=red,NSW=green,V=red,SA=blue,T=red}

همانطور که در شکل ۵-۱(b) نشان داده شده است مجسم کردن CSP به صورت گراف محدودیت می‌تواند سودمند باشد. نودهای گراف نشان‌دهنده متغیرهای مسئله و یال‌ها نشان‌دهنده محدودیت‌ها هستند. نگاه به مسئله از دید یک مسئله CSP امتیازات بسیار مهمی را دربردارد. از آنجا که نمایش حالات درCSP به شکل استاندارد است (مجموعه‌ای از متغیرها با مقادیر منتسب به آنها) توابع پسین و تست هدف به صورت عمومی تعریف می‌شوند و قابل استفاده در تمام مسائل CSP هستند. علاوه براین می‌توانیم توابع ابتکاری عمومی و موثری را ابداع کنیم که به تخصص خاص یک دامنه نیازی نداشته باشد. بالاخره ساختار گراف محدودیت می تواند فرآیند حل مسئله را بسیار ساده کند (در بعضی کاربردها به صورت نمایی پیچیدگی را کاهش می دهد). نمایش CSP اولین و ساده‌ترین نمایش از میان سلسله الگوهای نمایشی است که در این کتاب توسعه داده شده است.



شکل ۵-۱ (a)حالت‌ها ومناطق اصلی استرالیا. رنگ‌آمیزی این نقشه را می‌توان به عنوان مسئله ارضاء محدودیت در نظر گرفت. هدف، رنگ‌آمیزی مناطق است. به نحوی که رنگ ناحیه‌های همجوار یکسان نباشد. (b) مسئله‌ی رنگ‌آمیزی نقشه که به صورت گراف محدودیت نشان داده شده است.

بسادگی می‌توان دریافت که CSP می‌تواند به صورت یک فرمول‌بندی افزایشی[۱] و همچون یک مسئله جستجوی استاندارد عمل کند:

- حالت اولیه: مقداردهی تهی {} که در آن به هیچ یک از متغیرها مقداری نظیر نشده است.
- تابع پسین: به هریک از متغیرهای تهی می‌توان مقداری را نسبت داد به این شرط که این مقداردهی با مقادیر منتسب به متغیرهای قبلی ناسازگار نباشد.
- تست هدف: کامل بودن مقداردهی جاری بررسی می‌شود.
- هزینه مسیر: یک مقدار ثابت(به عنوان مثال ۱) به ازای هر مرحله.

پاسخ مسئله یک مقداردهی کامل است که در عمق n (تعداد متغیرها) اتفاق می‌افتد. علاوه براین ازآنجا که جستجوی درخت همواره تا عمق n ادامه پیدا می‌کند جستجوی عمق-اول در CSPها بسیار متداول است (به بخش ۵-۲ مراجعه کنید). از آنجا که مسیر رسیدن به جواب اهمیتی ندارد می توانیم از فرمول‌بندی کامل حالت استفاده کنیم که در آن هر حالت یک مقداردهی کامل بوده و لزوما محدودیت‌ها را ارضاء نمی‌کند. روش‌های جستجوی محلی در این نوع از فرمول‌بندی بسیار خوب عمل می کنند(به بخش ۵-۳ مراجعه کنید).

در ساده‌ترین نوعCSP، متغیرها گسسته و دامنه‌ها کران دارند که مسائلی همچون رنگ‌آمیزی نقشه نیز از این نوع‌اند. مسئله ۸-وزیر که در فصل ۳ به آن اشاره شد نیز می تواند به صورت CSP با دامنه کران‌دار تعریف شود که در آن متغیرهای $Q_1,...,Q_8$ مکان وزیرها در ستون های ۱،.....،۸ بوده و هریک از متغیرها دارای دامنه $\{1,2,3,4,5,6,7,8\}$ می‌باشند. اگر حداکثر اندازه دامنه هر متغیر در CSP برابرd باشد آنگاه تعداد مقداردهی‌های کامل مجاز، برابر $O(d^n)$ خواهد بود که برحسب تعداد متغیرها به صورت نمایی است.CSPهای بولی[۲] که در آنها متغیرها تنها مقادیر درست یا نادرست می‌پذیرند نیز نوعی از CSPهای با دامنه کران‌دار هستند.CSPهای بولی شامل بعضی از مسائل NP-کامل (همچون ۳SAT) می‌شوند. بنابراین در برخی حالات نمی‌توان انتظار داشت که مسائل CSP با دامنه کران‌دار را در زمانی کمتر از نمایی حل کنیم. در بسیاری از کاربردهای عملی، الگوریتم-های همه منظوره CSP مسائل را با مرتبه‌ای بزرگتر از الگوریتم‌های جستجوی همه‌منظوره فصل ۳ حل می‌کنند. البته متغیرهای گسسته نیز ممکن است دامنه بی‌کران داشته باشند(مثلا مجموعه‌ای از اعداد یا رشته‌ها). به عنوان مثال در مسئله زمان‌بندی کارها، زمان شروع هریک از کارها یک متغیر و مقدار آن عددی صحیح و بزرگتر از تاریخ فعلی است. در دامنه‌های بی‌کران تعریف محدودیت‌ها از طریق برشمردن ترکیب همه‌ی مقادیر مجاز امکانپذیر نیست و به جای آن باید از زبانی به نام زبان محدودیت استفاده کنیم. برای مثال اگر کار(۱) (که ۵ روز زمان می‌برد)پیش نیاز کار(۳) باشد،آنگاه ما نیاز به یک زبان محدودیت به صورت نامساوی جبری داریم که در آن : «شروع کار(۱) + ۵ ≤ شروع کار(۳)». همچنین حل چنین محدودیت‌هایی از طریق برشمردن تک تک مقداردهی‌ها غیرممکن خواهد بود زیرا تعداد آن‌ها بی‌کران است. الگوریتم های خاصی برای حل مسئله های محدودیت خطی روی متغیرهای صحیح (همچون مثال فوق که محدودیت ها به صورت خطی ظاهر شدند) وجود دارد که در اینجا به آن‌ها نخواهیم پرداخت. می‌توان نشان داد هیچ الگوریتم عمومی، برای حل مسائل محدودیت غیرخطی برروی متغیرهای صحیح وجود ندارد. در برخی از موارد می توانیم مسائل محدودیت روی

---

[۱]- Incremental formulation

[۲]- Boolean

اعداد صحیح را با استفاده از محدود کردن مقادیر تمامی متغیرها، به مسائلی با دامنه محدود تبدیل کنیم. برای مثال در مسئله زمان‌بندی کارها می‌توانیم حد بالایی برابر طول همه‌ی کارها برای زمان‌ها تعیین کنیم.

مسائل ارضاء محدودیت با دامنه‌های پیوسته در جهان واقع بسیار متداول هستند و تحقیقات گسترده‌ای برروی آنها انجام شده است. به عنوان مثال برنامه‌ریزی برای آزمایشات برروی تلسکوپ فضایی هابل نیازمند زمان‌بندی بسیار دقیقی است (برای انجام مشاهدات). شناخته‌شده‌ترین نوع از انواع CSPها با دامنه پیوسته مسائل برنامه‌ریزی خطی هستند که در آنها محدودیت‌ها، نامساوی‌هایی خطی هستند که تشکیل یک ناحیه محدب می‌دهند. همچنین مسائل دیگری با محدودیت‌ها و توابع هدف متفاوتی مورد مطالعه قرار گرفته است(برنامه‌ریزی درجه چهار، برنامه‌ریزی درجه دو و غیره).

علاوه بر بررسی انواع متغیرهای موجود در CSPها خوب است که به بررسی انواع محدودیت‌ها بپردازیم. ساده‌ترین نوع آن محدودیت یگانه است که در آن محدودیت‌ها تنها روی یک متغیر اعمال می‌شود. برای مثال فرض کنید بگوییم استرالیای جنوبی نباید سبز باشد. محدودیت‌های یگانه را می‌توان بااستفاده از پیش‌پردازش برروی دامنه متغیرهای مورد نظر و حذف مقادیری که با محدودیت‌ها تداخل دارند (از دامنه‌ها) نادیده گرفت. نوع دیگر، محدودیت دوگانه است که با دو متغیر ارتباط دارد برای مثال SA≠NSW یک محدودیت دوگانه است.CSP دوگانه نوعی از CSP است که در آن تنها محدودیت‌های دوگانه وجود دارد و می‌توان آن را همچون شکل ۵-۱(b) با گراف محدودیت نمایش داد.

محدودیت‌های مرتبه‌های بالاتر شامل سه یا بیش‌ازسه متغیر هستند. یک مثال آشنا در این زمینه معمای رمزنگاری است(به شکل ۵-۲(a) مراجعه کنید). به طور معمول در معمای رمزنگاری هر حرف به طور یکتا یک رقم را نشان می‌دهد. در مثال شکل شماره ۵-۲(a) محدودیت شش متغیر Alldiff(F,T,U,W,R,O) نشان داده شده است. همچنین می‌توان این محدودیت را با مجموعه‌ای از محدودیت‌های دوگانه همچون F≠T نمایش داد. محدودیت‌های دیگر برروی چهار ستون معما اعمال شده است که شامل تعداد زیادی متغیر می‌شود که آن‌ها را می‌توان بشرح زیر نوشت:

$$O + O = R + ۱۰ \times X_۱$$

$$X_۱ + W + W = U + ۱۰ \times X_۲$$

$$X_۲ + T + T = O + ۱۰ \times X_۳$$

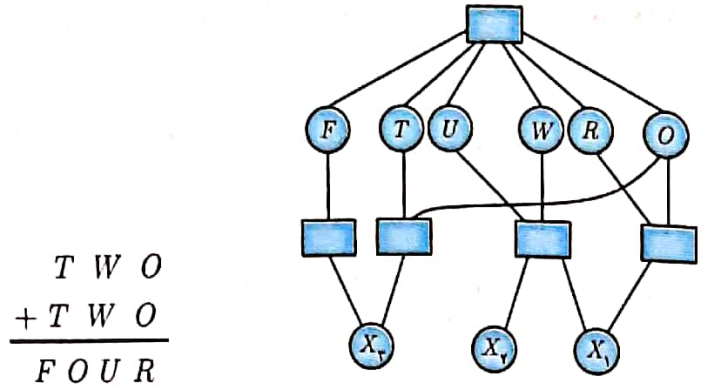
$$X_۳ = F$$

که در آن $X_۱$، $X_۲$ و$X_۳$ متغیرهای کمکی هستند که رقمی (صفر یا یک) را نشان می‌دهند که به ستون بعدی منتقل می‌شود. محدودیت‌هایی با مراتب بالاتر را می‌توان همچون شکل ۵-۲(b) در ابرگراف[1] محدودیت نمایش داد. با نگاهی تیزبینانه خواننده درمی‌یابد که محدودیت‌های Alldiff به محدودیت‌های دوگانه قابل تجزیه هستند. به طورمثال F≠T ، F≠U و غیره. تمرین شماره ۵-۱۱ از شما می‌خواهد، اثبات کنید که هر محدودیتی با مرتبه بالا و دامنه محدود قابل تجزیه به محدودیتی دوگانه است (البته با فرض استفاده از متغیرهای کمکی). به همین دلیل در این فصل ما تنها به بررسی محدودیت‌های دوگانه خواهیم پرداخت.

---

[1]- Hyper-Graph

محدودیت‌هایی که تا به حال تعریف کردیم همگی محدودیت‌های مطلق هستند به طوری که نقض هریک از آنها موجب حذف یک راه‌حل بالقوه می‌شود. در بسیاری از مسائل در دنیای واقعی CSPها دارای محدودیت اولویت‌دار هستند که راه‌حل ارجح را مشخص می‌کنند. به‌عنوان مثال در مسئله جدول زمانی دانشگاه ممکن است پروفسور X کلاس‌های صبح را ترجیح داده و پروفسور Y بعدازظهرها را بیش‌تربپسندد. جدول زمانی که در آن پروفسور X، ساعت ۲ بعدازظهر تدریس می‌کند پاسخی برای مسئله است اما پاسخی بهینه نخواهد بود. محدودیت‌های اولویت‌دار را می‌توان گاهی اوقات به‌شکل هزینه‌ای برای مقداردهی‌های خاص به متغیرها تعریف کرد. برای مثال گذاشتن کلاس بعدازظهر برای پروفسور X دو امتیاز منفی و کلاس صبح ۱ امتیاز مثبت برای هزینه کل تابع هدف محسوب می‌شود. با استفاده از این اطلاعات، CSPهای اولویت‌دار با استفاده از روش‌های جستجوی بهینه‌سازی قابل حل هستند(مبتنی بر مسیر یا محلی).



$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

(a) (b)

شکل ۵-۲ (a) مسئله رمزنگاری. هر حرف نشان‌دهنده یک رقم یکتا است. هدف، یافتن جایگزینی از ارقام برای حرف‌هاست، به‌طوری که حاصلجمع حاصل از نظر ریاضی درست باشد (صفرهای اول به حساب نمی‌آیند).(b) ابرگراف محدودیت برای مسئله رمزنگاری، محدودیت‌های Alldiff و محدودیت‌های اضافه کردن ستون را نشان می‌دهد. هر محدودیت یک مربع است که به متغیری متصل می‌شود که آن را محدود می‌کند.

۵-۲- جستجوی عقبگرد[1] برای CSP

در بخش قبل فرمول‌بندی CSPها را به عنوان یک مسئله جستجو بیان کردیم. بااستفاده از این فرمول‌بندی‌ها هرکدام از الگوریتم‌های جستجو فصل ۳ و ۴ قادرند CSP را حل نمایند. فرض کنید جستجوی سطح-اول را برروی هر یک از مسائل فرمول‌بندی عمومی CSP ارائه شده در بخش قبل اعمال کنیم. ناگهان متوجه مسئله بغرنجی می‌شویم: فاکتور انشعاب در بالاترین سطح برابر nd است زیرا همه مقادیر می‌توانند به هریک از متغیرها نسبت داده شوند. در مرحله بعدی فاکتور انشعاب برابر d(n-۱) است و تا n سطح به همین ترتیب ادامه پیدا می‌کند. بنابراین ما $n!\times d^n$ برگ تولید کردیم در حالی که تنها d^n مقداردهی کامل وجود دارد.

مسئله فرمول‌بندی ساده و منطقی ما، یک خاصیت بسیار حیاتی را (که در همه‌ی CSP ها مشترک است) نادیده گرفته است: جابجاپذیری. یک مسئله جابه‌جاپذیر است اگر ترتیب به‌کارگیری هر مجموعه‌ی دلخواه از کنش‌ها تاثیری بر نتیجه کار نداشته باشد. این ویژگی مختص CSP است زیرا هنگامی که مقادیر را به متغیرها نسبت

[1]- Backtracking

می‌دهیم صرف نظر ازترتیب آنها به مقداردهی یکسانی می‌رسیم. لذا تمام الگوریتم‌های جستجوی CSP برای تولید جانشین‌ها در درخت جستجو فقط مقداردهی‌های مربوط به یک متغیر از هر گره را در نظر می‌گیرند. برای مثال، در گره ریشه از درخت جستجوی رنگ‌آمیزی نقشه استرالیا ممکن است بین گزینه های SA= قرمز، SA= سبز، SA=آبی حق انتخاب داشته باشیم، اما هیچگاه بین SA= قرمز و WA= آبی انتخابی انجام نمی‌دهیم. با استفاده از این ویژگی تعداد برگ‌ها به d^n می‌رسد (درست همانطورکه می‌خواستیم.)

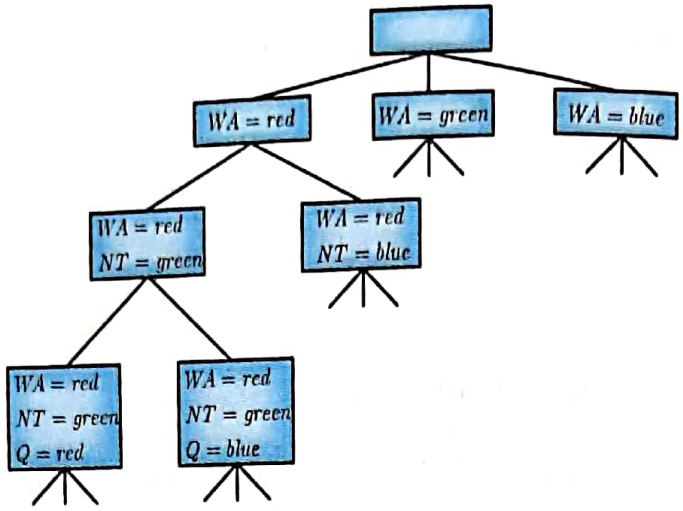
جستجوی عقبگرد در جستجوی عمق- اول کاربرد دارد. جستجویی که در آن در هر لحظه مقداری را برای یک متغیر انتخاب می‌کند و هنگامی که برای یک متغیر هیچ مقدار مجازی برای مقداردهی باقی نماند، عقب گرد می‌کند. این الگوریتم در شکل ۵-۳ نشان داده شده است.

توجه داشته باشید این الگوریتم از روشی استفاده می‌کند که درآن در هر لحظه تنها یک متغیر پسین و به صورت گام‌به‌گام و از طریق اعمال تغییر برروی مقداردهی جاری(به جای کپی مجدد آن) تولید می‌شود. از آنجا که CSP به صورت استاندارد نمایش داده می‌شود، هنگام استفاده از الگوریتم جستجوی عقبگرد نیازی به تعریف حالت اولیه، تابع پسین و تابع تست هدف برای یک دامنه به طور خاص نداریم. قسمتی از درخت جستجوی مسئله استرالیا در شکل ۵-۴ نشان داده شده است که در آن متغیرها به ترتیب Q،NT،WA، ... مقدار گرفته‌اند.

```
function Backtracking-Search(csp) returns a solution, or failure
    return Recursive-Backtracking({},csp)

function Recursive-Backtracking (assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ←Select –Unassigned-Variable (Variables[csp], assignment, csp)
    for each value in Order-Domain-Values (var, assignment, csp) do
        if value is consistent with assignment according to Constraints [csp] then
            add {var= value) to assignment
            result ← Recursive-Backtracking (assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
```

شکل ۵-۳ الگوریتم عقبگرد ساده برای مسائل ارضای محدودیت.این الگوریتم به صورت جستجوی عمق- اول بازگشتی (که در فصل ۳ درباره آن توضیح داده شد) مدل شده است. توابعSELECT-UNASSIGNED-VARIABLE و ORDER-DOMAIN-VALUES می‌توانند برای پیاده‌سازی توابع ابتکاری همه‌منظوره اشاره شده در متن استفاده شوند.



شکل ۵-۴ قسمتی از درخت جستجویی که با الگوریتم عقبگرد ساده در مسئله رنگ‌آمیزی نقشه در شکل ۵-۱ تولید شده است.

الگوریتم عقبگرد ساده یک نوع الگوریتم غیرآگاهانه(در اصطلاح فصل ۳) است بنابراین ما انتظار نداریم که برای مسائل بزرگ چندان موثر باشد. نتایج بدست آمده برای یک‌سری از مسائل به طور نمونه در ستون اول شکل ۵-۵ نشان داده شده است که در آن نتایج مطابق پیش‌بینی‌ها است.

در فصل ۴ با استفاده از توابع ابتکاری که به طور خاص برای یک دامنه بوده و برگرفته از دانش ما از مسئله بودند توانستیم عملکرد ضعیف الگوریتم‌های جستجوی غیرآگاهانه را جبران کنیم. نکته جالب این است که ما می توانیم CSP ها را بدون داشتن اطلاعات خاص مسئله به صورت کارایی حل کنیم. درعوض ما در CSPها به دنبال روش‌های همه‌منظوره‌ای می‌گردیم که به سوالات زیر پاسخ می‌دهد:

- کدام متغیر باید مقداردهی شود وچه ترتیبی از مقادیر باید روی آن امتحان شوند؟
- انتساب مقدار فعلی به متغیر جاری چه تاثیری برروی متغیرهای بدون مقدار دارد؟
- وقتی مسیری شکست می‌خورد(بدین معنی که به حالتی می‌رسیم که در آن هیچ متغیری مقدار مجازی برای مقداردهی ندارد)آیا جستجو می‌تواند از تکرار شکست درمسیرهای بعدی جلوگیری کند؟ در قسمت‌های بعدی به‌ترتیب به این سوال‌ها پاسخ می‌دهیم.

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(>۱,۰۰۰K)	(>۱,۰۰۰K)	۲K	٦۰	٦٤
n- Queens	(>٤۰,۰۰۰K)	۱۳,٥۰۰K	(>٤۰,۰۰۰K)	۸۱۷K	٤K
Zebra	۳,۸٥۹K	۱K	۳٥K	۰.٥K	۲K
Random ۱	٤۱٥K	۳K	۲٦K	۲K	
Random ۲	۹٤۲K	۲۷K	۷۷K	۱٥K	

شکل ۵-۵ مقایسه الگوریتم‌های مختلف CSP برروی چند مسئله. الگوریتم‌ها از چپ به راست عبارتند از عقبگرد ساده، عقبگرد با ابتکار MRV، بررسی رو به جلو، بررسی روبه‌جلو با MRV، و جستجوی محلی با کمترین برخورد. در هر ستون به طور میانگین تعداد بررسی‌های انجام شده برای تطبیق سازگاری بعد از پنج بار اجرای برنامه برای حل مسئله نوشته شده است. دقت کنید همه‌ی مقادیر به جز دوتای بالا در سمت راست براساس مقیاس کیلو(K) هستند. مقادیر درون پرانتز بدین معنی است که بعد از این تعداد بار بررسی سازگاری جوابی برای مسئله یافت نشد. مسئله اول رنگ‌آمیزی ۵۰ استان آمریکا با ۴ رنگ است. مسائل دیگر از Bacchus و Van Run در جدول ۱ گرفته شده است. مسئله دوم تعداد بررسی‌های انجام شده برای حل مسئله n- وزیر به ازای n از ۲ تا ۵۰ را نشان می‌دهد. مسئله سوم "Zebra puzzle" است دو تای

آخر دو مسئله تصادفی هستند. نتایج نشان می‌دهد که الگوریتم بررسی پیشرو با ابتکار MRV از بقیه الگوریتم‌های عقبگرد بهتر عمل می‌کند(اگرچه همواره از الگوریتم حداقل- تداخل بهتر عمل نمی‌کند).

۵-۲-۱- ترتیب انتخاب متغیر ها و مقادیر آنها

الگوریتم عقب‌گرد شامل دستور زیر است:

Var←SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)

به طور پیش‌فرض تابع SELECT-UNASSIGNED-VARIABLE متغیر بدون مقدار را به‌ترتیب لیست VARIABLE[csp] انتخاب می‌کند. این نوع ایستا از ترتیب متغیرها به ندرت منجر به کاراترین جستجو می‌شود. به عنوان مثال بعد ازمقداردهی WA= قرمز و NT = سبز، تنها یک مقدار مجاز برای SA باقی می‌ماند بنابراین منطقی به نظر می‌رسد به جای مقداردهی Q ابتدا SA = آبی را نسبت دهیم. در حقیقت بعد ازمقداردهی SA، تنها یک انتخاب برای NSW،Q و V باقی می‌ماند. این ابتکار(انتخاب متغیری با کمترین مقدار مجاز) که به طور شهودی نیز قابل درک است، ابتکار کمترین مقادیر باقیمانده[۱](MRV) نامیده می‌شود. به این روش «انتخاب متغیر با بیش‌ترین محدودیت» ویا «اکتشاف اولین-شکست»(زیرا این روش متغیری که بیش‌ترین احتمال بروز شکست را دارد انتخاب و بدین ترتیب درخت را هرس می‌کند) نیز می‌گویند. متغیرX را در نظر بگیرید که هیچ مقدار مجازی برای مقداردهی ندارد.MRV، متغیر X را انتخاب و این مقداردهی با شکست مواجه می‌شود بنابراین MRV از جستجوهای بی‌فایده‌ای که همه آن‌ها سرانجام با انتخاب X به شکست منجر می‌شدند، جلوگیری می‌کند.

ستون دوم شکل ۵-۵ کارایی ابتکار BT+MRV را نشان می‌دهد. کارایی این روش بسته به نوع مسئله از ۳ تا ۳۰۰۰ برابر بهتر از الگوریتم عقبگرد ساده است. توجه داشته باشید که ما در معیارهای اندازه‌گیری کیفیت خود، هزینه محاسباتی این ابتکار(MRV) را نادیده گرفتیم. در بخش بعدی روشی را توضیح خواهیم داد که به مدیریت این هزینه‌ها می‌پردازد.

از آنجا که در ابتدا همه‌ی نواحی دارای سه رنگ مجاز هستند بنابراین ابتکار MRV در انتخاب اول برای رنگ آمیزی نقشه استرالیا کمکی به ما نمی‌کند. در اینگونه موارد درجه ابتکار[۲] را تعریف می‌کنیم و در آن تلاش می‌کنیم که فاکتور انشعاب را در انتخاب‌های بعدی کمتر کنیم و این ایده را از طریق انتخاب متغیری که بیش‌ترین محدودیت را برای متغیرهای بدون مقدار ایجاد می‌کند عملی می‌کنیم. در شکل ۵-۱ SA متغیری با بیش‌ترین درجه است(۵). دیگر متغیرها دارای درجه ۳،۲ (به جز T که درجه آن صفر است) هستند. درحقیقت بعد از انتخاب SA با به‌کارگیری درجه‌ابتکار می‌توان مسائل را بدون حتی یک گام اشتباه حل کرد. یعنی می‌توانید در هر بار انتخاب رنگ سازگار را انتخاب و بدون عقب‌گرد به پاسخ مسئله دست پیدا کنید. ابتکار «کمترین مقادیر باقی‌مانده» اغلب روش به‌مراتب قوی‌تری است اما «درجه ابتکار» می‌تواند در مواقع بحرانی گره گشا باشد.

۱- Minimum Remaining Values
۲- Degree Heuristic

وقتی یک متغیر انتخاب می‌شود،الگوریتم باید درمورد ترتیب امتحان کردن مقادیر تصمیم‌گیری کند. در بعضی مواقع انتخاب «کمترین - محدودکننده- مقدار[۱]» می‌تواند موثرباشد. در این روش الگوریتم مقداری را انتخاب می‌کند که از همسایه‌هایش کمترین تعداد انتخاب را حذف کند. برای مثال فرض کنید در شکل ۵- ۱ مقداردهی‌های WA = قرمز و NT = سبز انجام شده و نوبت به Q رسیده است. رنگ آبی انتخاب خوبی نیست زیرا آخرین مقدار باقی مانده مجاز برای همسایه Q (SA) را حذف می‌کند. ابتکار «کمترین-محدودکننده-مقدار» مقدار قرمز را به آبی ترجیح می‌دهد. در کل، این نوع از ابتکار سعی می‌کند بیشترین انعطاف‌پذیری را برای مقداردهی متغیرهای بعدی به‌وجود بیاورد. این موضوع برای مسائل حل نشدنی نیز صادق است.

۵-۲-۲- انتشار اطلاعات در محدودیت‌ها

تا این‌جا الگوریتم جستجو، محدودیت‌ها را تنها برروی یک متغیر و تنها زمانی که توسط تابع SELECT-UNASSIGNED-VARIABLE انتخاب می‌شد، در نظر می‌گرفت. اما می‌توان با بررسی بعضی از محدودیت‌ها در ابتدای جستجو ویا حتی پیش از شروع آن، فضای جستجو را تا حد زیادی کاهش داد.

بررسی پیشرو

یکی از راه‌های استفاده بهتر از محدودیت‌ها حین جستجو، «بررسی پیشرو[۲]» نامیده می‌شود. در این روش بعد از مقداردهی یک مقدار به متغیر X، فرآیند «بررسی پیشرو» متغیرهای بدون مقدار مثل Y که از طریق یک محدودیت به X متصل شده‌اند را پیدا کرده و مقادیری را که با مقدار X ناسازگار است از دامنه Y حذف می‌کند. شکل شماره ۵-۶ مسیر پیشرفت جستجوی رنگ‌آمیزی نقشه را با روش بررسی پیشرو نشان می‌دهد. در این مثال دو نکته مهم حائز اهمیت است. اول اینکه بعد از مقداردهی‌های WA= قرمز و Q = سبز دامنه‌های NT و SA به یک مقدار منفرد کاهش پیدا می‌کنند و ما از انشعاب روی این دو متغیر با استفاده از اطلاعات منتشر شده‌ی WA و Q جلوگیری کردیم. حال ابتکار MRV که همواره در کنار روش "بررسی پیشرو" استفاده می‌شود به طور خودکار SA و سپس NT را انتخاب می‌کند (در واقع می‌توانیم به "بررسی پیشرو" به عنوان یک روش کارا برای محاسبه اطلاعات مورد نیاز ابتکار MRV نگاه کنیم). نکته دومی که باید به آن توجه کنیم این است که بعد ازمقداردهی V= آبی دامنه SA خالی می‌شود. حال بررسی پیشرو درمی‌یابد که مقداردهی {V=آبی،Q= سبز،WA= قرمز} با محدودیت های مسئله ناسازگار است بنابراین عقبگرد می‌کند.

۱- Least-Constraining-Value
۲- Forward-Checking

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=red	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q=green	(R)	B	(G)	R B	R G B	B	R G B
After V=blue	(R)	B	(G)	R	(B)		R G B

شکل ۵-۶ مراحل پیشرفت مسأله رنگ‌آمیزی ایالت‌های استرالیا با روش Forward Checking. ابتدا WA=red سپس الگوریتم FC ، مقدار red را از دامنه‌ی همسایه‌های WA (یعنی NT و SA) حذف می‌کند. سپس Q=green ، که مقدار green از همسایه‌های Q (NT ، SA و NSW) حذف می‌شود. سپس V = blue ، که مقدار blue از همسایه‌های V (NSW ، SA) حذف می‌شود که در اینحالت هیچ مقداری برای دامنه SA باقی نمی‌ماند.

انتشار محدودیت

اگرچه بررسی پیشرو بسیاری از ناسازگاری‌ها را می‌یابد اما همچنان بسیاری از ناسازگاری‌ها وجود دارند که بررسی پیشرو آن‌ها را کشف نمی‌کند. برای مثال به ردیف سوم از شکل ۵-۶ دقت کنید. هنگامی‌که WA قرمز و Q سبز است دو متغیر NT و SA مجبور به پذیرش رنگ آبی هستند درحالی که این دو متغیر همسایه‌اند و نمی‌توانند مقدار واحدی را بپذیرند. از آنجا که بررسی پیشرو به اندازه کافی آینده‌های دور را بررسی نمی‌کند در کشف این ناسازگاری ناتوان است. «انتشار محدودیت» یک اصطلاح عمومی برای انتشار اثر یک محدودیت برروی یک متغیر به دیگر متغیرهاست. در این مثال باید این اثر را از WA و Q به NT و SA انتشار دهیم (همانطور که بررسی پیشرو بدرستی این کار را انجام داد) و سپس این عمل را برای NT و SA انجام دهیم تا ناسازگاری مشخص گردد. البته این کار باید بسرعت انجام شود: زیرا خوب نیست که ما با استفاده از این روش اندازه جستجو را کاهش داده اما زمان انتشار این محدودیت‌ها از یک جستجوی معمولی بیش‌تر شود.

ایده «سازگاری یال[1]» یک روش بسیار سریع برای انتشار محدودیت‌ها است که بسیار قوی‌تر از «بررسی پیشرو» عمل می‌کند. در اینجا منظور از یال، یال‌های جهت‌دار گراف محدودیت (مثل یال کشیده شده از SA به NSW) هستند. با فرض دانستن دامنه‌ها، یال گذرنده از SA به سمت NSW سازگار است اگر به ازای هر مقدار x در دامنه SA مقداری مثل y در دامنه NSW وجود داشته باشد که با x سازگار باشد. در ردیف سوم شکل ۵-۶ دامنه‌های فعلی SA و NSW به ترتیب {آبی} و {قرمز، آبی} است. برای SA = آبی یک مقداردهی سازگار به‌صورت NSW= قرمز وجود دارد بنابراین یال کشیده شده از SA به سمت NSW سازگار است. از سوی دیگر یال معکوس آن یعنی از NSW به SA سازگار نیست: زیرا به ازای مقداردهی NSW= آبی هیچ مقداردهی سازگاری برای SA وجود ندارد. این یال با حذف مقدار آبی از دامنه NSW سازگار می‌شود.

همچنین می‌توانیم «سازگاری یال» را روی یال SA به NT در همان فرآیند جستجو اعمال کنیم. ردیف سوم جدول ۵-۶ نشان می‌دهد که این دو متغیر هر دو دارای دامنه {آبی}هستند. نتیجه اینکه آبی باید از دامنه SA حذف شود بنابراین دامنه آن خالی می‌شود. بنابراین ناسازگاری که توسط بررسی پیشرو کشف نشده بود با اعمال «سازگاری یال» بسیار زود شناسایی شد. فرآیند بررسی «سازگاری یال» می‌تواند به عنوان یک پیش‌پردازش،

[1]- Arc consistency

قبل از شروع فرآیند جستجو و یا به صورت یک فرآیند انتشار (همچون بررسی پیشرو) بعد از هرمقداردهی به اجرا درآید (این الگوریتم گاهی MAC[۱] به معنی نگهداری سازگاری یال نامیده می‌شود). درهرصورت این فرآیند تا زمانی که هیچ ناسازگاری وجود نداشته باشد به صورت مکرر اعمال می‌شود. زیرا هنگامی که یک مقدار از دامنه یک متغیر برای از بین بردن ناسازگاری حذف می‌شود ممکن است ناسازگاری دیگری در یال‌هایی که به آن متغیر اشاره میکنند روی دهد. الگوریتم کامل سازگاری یال (AC-۳) یال‌هایی را که نیاز به چک کردن برای ناسازگاری دارند در یک صف نگهداری می‌کند(شکل ۵-۷ را مشاهده کنید). هر یال (X_i,X_j) به نوبت از صف برداشته و چک می‌شود، اگر نیاز به حذف مقداری از دامنه Xiباشد آنگاه هر یال (Xk,Xi) که به Xi اشاره می‌کند باید دوباره در صف اضافه و چک شود. پیچیدگی زمانی "سازگاری یال" را می‌توان اینگونه تحلیل کرد: یک CSP دوگانه حداکثر $O(n^۲)$ یال دارد، از طرفی هریال (X_k,X_i) حداکثر d بار در صف اضافه می‌شود(زیرا X_i حداکثر d مقدار برای حذف دارد و با توجه به اینکه بررسی سازگاری یک یال در زمان $O(d^۲)$ انجام می‌شود بنابراین زمان کل در بدترین حالت برابر $O(n^۲d^۳)$ می‌شود. اگرچه این روش بسیار پرهزینه‌تر از «بررسی پیشرو» است اما این هزینه اضافی اغلب به‌صرفه است.

از آنجا که ۳SAT حالت خاصی از CSP است انتظار نداریم که الگوریتمی با پیچیدگی زمانی چندجمله‌ای برای بررسی سازگاری یک CSP بیابیم. لذا الگوریتم «سازگاری یال» همه‌ی ناسازگاری‌ها را تشخیص نمی‌دهد. برای مثال در شکل ۵-۱مقداردهی {WA= قرمز ، NSW= قرمز} ناسازگار است اما AC-۳ این ناسازگاری را نمی‌یابد. شکل‌های قوی‌تری از این انتشار با استفاده از روش k- سازگاری تعریف می‌شود. یک CSP، k- سازگار است اگر به ازای هر مجموعه از k-۱ متغیر و به‌ازای هر مقداردهی سازگار به آن متغیرها، همواره بتوان یک مقدار سازگار به متغیر k ام نسبت داد. برای مثال ۱-سازگاری به معنی این است که هر متغیر با خودش سازگار است(این نوع از سازگاری همچنین سازگاری گره نیز نامیده می‌شود). ۲- سازگاری دقیقا همان سازگاری یال است. ۳- سازگاری به این معنی است که هر دو متغیر مجاوری بتوانند به متغیر سوم همسایه‌اش اضافه و گسترش یابند.

یک گراف قویا k- سازگار است اگر k- سازگار و (k-۱)- سازگار، (k-۲) – سازگار و .... ۱- سازگار باشد. حال فرض کنید یک مسئله CSP با n گره داریم و آنرا قویا n- سازگار کرده‌ایم (قویا k- سازگار برای k=n) در این حالت می‌توانیم مسئله را بدون عقبگرد حل کنیم. زیرا ابتدا مقداری سازگار برای $X_۱$ می‌یابیم. از طرفی مطمئنیم که برای متغیر $X_۲$ نیز می‌توانیم مقداری انتخاب کنیم زیرا گراف ۲- سازگار است، و همین طور برای $X_۳$، زیرا گراف ۳- سازگار است و همین طور تا آخر. به‌ازای هر متغیر X_i ما تنها نیاز به جستجو در d مقدار در دامنه برای یافتن یک مقدار سازگار با $X_۱,\ldots.X_{i-۱}$ داریم. و مطمئنا پاسخ مسئله را در زمان O(nd) می‌یابیم. اگرچه هر الگوریتمی که بخواهد n-سازگاری را برقرار نماید در بدترین حالت دارای پیچیدگی زمانی نمایی برحسب n است.

function AC-۳ (csp) **returns** the CSP, possibly with reduced domains

inputs: csp, a binary CSP with variables $\{X_۱, X_۲, \ldots, X_n\}$

local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty **do**

۱- Maintaining arc consistency

$(X_i, V_j) \leftarrow$Remove-First (queue)
if Remove- Inconsistent-Values (X_i, V_j) **then**
FOR EACH X_k IN Neighbors $[X_i]$ **DO**
ADD (X_k, X_i) TO QUEUE

---

function REMOVE-INCONSISTENT-VALUES (X_i, X_j) **returns** true iff we remove a value
removed ← false
for each x **in** Domain $[X_i]$ **do**
if no value y in Domain $[X_j]$ allows (x,y) to satisfy the constraint between X_i and X_j
then delete x from Domain $[X_i]$; removed ← true
return removed

شکل ۵-۷ الگوریتم سازگاری یال AC-۳. بعد از اعمال AC-۳ یا تمامی یال‌ها سازگارند و یا دامنه یکی از متغیرها خالی می‌شود که در اینصورت نشان می‌دهد که این CSP نمی‌تواند سازگاری یال داشته باشد و بنابراین حل نمی‌شود.

البته «سازگاری یال» و n- سازگاری فاصله بسیار زیادی از هم دارند: بررسی سازگاری‌های قوی‌تر زمان بیش‌تر نیاز دارد اما درعوض اثر بیش‌تری در کاهش فاکتور انشعاب و زمان کشف مقداردهی‌های ناسازگار دارد. همچنین می‌توان کوچکترین عدد k را محاسبه کرد که به‌ازای آن اجرای الگوریتم k- سازگاری ما را مطمئن می‌کند که مسئله بدون عقبگرد قابل حل است. اگرچه اغلب این ایده غیر عملی است (تمرین ۵-۴ را مشاهده نمایید). در عمل تعیین مناسب‌ترین سطح برای بررسی سازگاری امری تجربی است.

مدیریت محدودیت‌های خاص

انواع مختلفی از محدودیت‌ها در مسائل واقعی وجود دارند که می‌توانند توسط الگوریتم‌های خاصی مدیریت شوند که از روش‌های عمومی گفته شده بسیار کاراترند. به عنوان مثال محدودیت Alldiff که باید تمامی متغیرهای موجود درآن دارای مقادیری متمایز باشند (برای مثال در معمای رمزنگاری) را در نظر بگیرید. یک راه ساده برای کشف ناسازگاری محدودیت Alldiff به صورت زیر است: اگر محدودیت دارای m متغیر باشد، و n مقدار متمایز ممکن وجود داشته باشد و $m > n$ باشد، آنگاه محدودیت ارضاء نمی‌شود.

بنابراین می‌توان الگوریتم ساده زیر را نتیجه گرفت: ابتدا همه‌ی متغیرهای با دامنه تک عضوی را حذف و مقدار آن متغیر را از دامنه دیگر متغیرها حذف کنید. این کار را تا زمانی که متغیرهایی با دامنه تک عضوی وجود دارد ادامه دهید. در هر لحظه که یک دامنه خالی به‌وجود آمد ویا متغیرهای بیش‌تری از مقدار دامنه باقی ماند، آنگاه ناسازگاری کشف شده است.

با استفاده از این روش می‌توان ناسازگاری موجود در مقداردهی {WA= قرمز، NSW= قرمز} در شکل ۵-۱ را یافت. توجه کنید که متغیرهای SA ، NT و Q به صورت موثری با استفاده از محدودیت Alldiff به هم مرتبط شده‌اند زیرا هر جفت باید رنگ متفاوتی داشته باشد. بعد از اعمال AC-۳ روی این مقداردهی دامنه

تمامی متغیرها به {سبز، آبی} کاهش می‌یابد. پس ما سه متغیر و دو رنگ داریم بنابراین محدودیت Alldiff نقض شده است. بنابراین گاهی‌اوقات در محدودیت‌های مرتبه بالا اعمال یک رویه[۱] «بررسی سازگاری» بسیار کاراتر از اعمال الگوریتم «سازگاری یال» برروی مجموعه معادل محدودیت‌های دوگانه آن می‌باشد.

شاید یکی از مهم‌ترین انواع محدودیت‌های مرتبه بالا، «محدودیت منابع[۲]» باشد که گاهی اوقات محدودیت atmost نیز نامیده می‌شود. برای مثال فرض کنید $PA_1,\ldots,PA_4$ تعداد کارکنان اختصاص داده شده برای انجام چهار ماموریت باشد. محدودیت atmost(۱۰, PA_1,PA_2,PA_3,PA_4) به این شکل تعریف می‌شود: حداکثر تعداد کارکنان اختصاص داده شده برای انجام کل کارها برابر ۱۰ می‌باشد. یعنی با بررسی حاصلجمع مینیمم مقدار هریک از دامنه‌های فعلی می‌توان ناسازگاری را شناسایی کرد. برای مثال اگر متغیرها دارای دامنه {۳,۴,۵,۶} باشند آنگاه این محدودیت هیچگاه ارضاء نمی‌شود. همچنین در صورتی که ماکزیمم مقدار یک دامنه با مینیمم مقدار دامنه‌های دیگر سازگار نباشد، مقدار ماکزیمم را حذف کرده و به‌این ترتیب به سازگاری نزدیک‌تر می‌شویم. بنابراین اگر هریک از متغیرها دارای دامنه {۲,۳,۴,۵,۶} باشند آنگاه می‌توانیم مقادیر ۵ و۶ را از کلیه دامنه‌ها حذف کنیم. در مسئله‌های «منبع- محدود» بزرگ با مقادیرصحیح(مثل مسائل لوجیستیکی که شامل صدها نفر و هزاران وسیله نقلیه است) این امکان وجود ندارد که دامنه هر متغیر را با یک مجموعه بزرگ از اعداد صحیح نمایش و بتدریج آن مجموعه را با روش‌های بررسی سازگاری کاهش دهیم. در عوض دامنه‌ها با تعریف کران‌های بالا و پایین نمایش داده شده و توسط روش «انتشار کران‌ها» مدیریت می‌شوند. برای مثال فرض کنید دو پرواز ۲۷۱ و ۲۷۲ وجود داشته باشد که ظرفیت هریک از آن‌ها به ترتیب ۱۶۵ و ۳۸۵ است. دامنه اولیه برای تعداد مسافر در هریک از پروازها به شکل زیر است:

$$Flight۲۷۱ \in [۰,۱۶۵]\ ,\ Flight۲۷۲ \in [۰,۳۸۵]$$

حال فرض کنید محدودیت دیگری داریم که در آن مجموع مسافران دو هواپیما باید ۴۲۰ باشد:

$$[۴۲۰,۴۲۰] \in Flight۲۷۱+Flight۲۷۲$$

با انتشار محدودیت کران‌ها دامنه‌ها به صورت زیر کاهش می‌یابند:

$$Flight۲۷۱ \in [۳۵,۱۶۵]\ ,\ Flight۲۷۲ \in [۲۵۵,۳۸۵]$$

می‌گوییم یک CSP، کران- سازگار[۳] است اگر به ازای هر دو متغیر X و Y و برای هر دو کران بالا و پایین X، مقداری برای Y وجود داشته‌باشد که محدودیت بین X وY را ارضاء کند. این نوع از انتشار کران‌ها به صورت گسترده در مسائل کاربردی محدودیت به کار می‌روند.

عقبگرد هوشمند: نگاه به عقب

الگوریتم BACKTRACKING-SEARCH شکل ۵-۳ هنگامی که شاخه‌ای از جستجو با شکست مواجه می‌شود سیاست بسیار ساده‌ای را دنبال می‌کند: به متغیر پیشین رفته و مقدار جدیدی را برای آن امتحان می-

---

۱- Procedure
۲- Resource constraint
۳- Bounds-Consistent

کند. به این روش عقبگرد زمانی[1] گویند زیرا در آن تصمیمات اخیر مجددا بازنگری می‌شوند. در این بخش با روش‌های بهتری از عقبگرد آشنا خواهیم شد.

فرض کنید ما الگوریتم «عقبگرد ساده» را در شکل ۵-۱ با ترتیب ثابت متغیرهای Q، NSW، V، T، SA، WA، NT اعمال کنیم حال دقت کنید که چه اتفاقی روی می‌دهد. فرض‌کنید مقداردهی {Q=قرمز،NSW=سبز، V=آبی، T=قرمز} انجام شده است. هنگامی که متغیر پسین(SA) را امتحان می‌کنیم درمی‌یابیم که به‌ازای همه مقادیر محدودیت نقض می‌شود. بنابراین به T برگشته و رنگ جدیدی را برای تاسمانیا امتحان می‌کنیم. بدیهی است که این کار احمقانه است زیرا انتخاب رنگ جدیدی برای تاسمانیا نمی‌تواند راه‌حل مناسبی برای حل مشکل SA (یا همان استرالیای جنوبی) باشد.

روش عقبگرد هوشمندانه‌تر، بازگشت به یک متغیر از مجموعه متغیرهایی است که علت اصلی شکست در جستجو بوده‌اند. به این مجموعه، «مجموعه تداخل»[2] گویند. در این مثال مجموعه تداخل برای SA، {Q,NSW,V} می‌باشد. درحقیقت مجموعه تداخل برای متغیر X مجموعه‌ای از متغیرهاست که پیش از این مقداردهی شده و از طریق محدودیت‌ها به X مرتبط شده‌اند.

روش «پرش به عقب[3]» آنقدر عقبگرد می‌کند تا به جدیدترین متغیر اضافه شده در مجموعه تداخل برخورد کند. در مثال بالا الگوریتم «پرش به عقب» از روی تاسمانیا پریده و مقدار جدیدی را برای V امتحان می‌کند. این الگوریتم با ایجاد تغییر در الگوریتم BACKTRACKING-SEARCH به راحتی قابل پیاده‌سازی است. بدین ترتیب که الگوریتم، «مجموعه تداخل» را هنگام بررسی مقادیر مجاز برای مقداردهی به متغیرها می‌سازد. اگر در شاخه‌ای از جستجو، هیچ مقدار مجازی پیدا نشد آنگاه الگوریتم جدیدترین عنصراضافه شده به مجموعه تداخل و نشانه‌ای به منزله شکست الگوریتم در این شاخه از جستجو را برمی‌گرداند.

با نگاهی تیزبینانه می‌توان دریافت که الگوریتم «بررسی پیشرو» می‌تواند مجموعه تداخل را بدون هیچگونه عمل اضافی بدست آورد: هنگامی‌که «بررسی پیشرو» مقداری را از دامنه Y (به دلیل مقداردهی به X) حذف می‌کند آنگاه باید X را به مجموعه تداخل Y اضافه کند. همچنین هنگامی که آخرین مقدار از دامنه Y حذف شد، متغیرهای موجود در مجموعه تداخل Y به مجموعه تداخل X اضافه می‌شوند. بنابراین هنگامی که به Y می‌رسیم در صورت شکست، می‌دانیم که به کجا باید عقبگرد کنیم.

اما با نگاهی دقیق به نکته عجیبی پی می‌بریم: پرش به عقب تنها زمانی اتفاق می‌افتد که همه‌ی مقادیر درون دامنه با مقداردهی فعلی تداخل داشته باشند. اما «بررسی پیشرو» این تداخل را شناسایی و از رسیدن جستجو به چنین گره‌ای جلوگیری می‌کند. در واقع، می‌توان نشان داد که هر شاخه‌ای از جستجو که توسط الگوریتم «پرش به عقب» هرس شود توسط الگوریتم «بررسی پیشرو» نیز هرس می‌شود. لذا الگوریتم پرش به عقب ساده درون جستجوی بررسی پیشرو (در واقع جستجوهایی که بررسی سازگاری را بشکل قوی‌تری انجام می‌دهند مثل MAC) تکراری و زائد است.

علی‌رغم مطالب گفته شده در پاراگراف قبلی ایده اصلی الگوریتم «پرش به عقب» همچنان یک ایده مناسب محسوب می‌شود: برگشت به جایی که علت اصلی شکست در جستجو بوده است. الگوریتم «پرش به عقب» درست

۱- Chronological Backtracking
۲- Conflict Set
۳- Back Jumping

هنگامی که دامنه یک متغیر خالی شود متوجه شکست می‌شود در صورتی که در بسیاری از موارد شکست یک شاخه از جستجو از مدت‌ها قبل از این اتفاق قطعی به‌نظر می‌رسد. مجددا مقداردهی {WA= قرمز، NSW= قرمز} (که بنا به بحث‌های قبلی ناسازگار بود) را در نظر بگیرید. فرض کنید قرمز=T را اعمال کرده و سپس مقداردهی‌های SA،V،Q،NT را انجام دهیم. می‌دانیم که هیچ نوع مقداردهی برای چهار متغیر آخر بدرستی عمل نمی‌کند بنابراین سرانجام به این نتیجه خواهیم رسید که برای مقداردهی NT مقدار مجازی نخواهیم داشت. حال سوال این است که به کجا عقبگرد کنیم؟ دراین حالت پرش به عقب به درستی عمل نمی‌کند زیرا NT دارای مقادیر سازگار با مقداردهی‌های قبلی است (NT مجموعه کاملی از متغیرهای قبلی که منجر به شکست آن شده‌اند را ندارد). همچنین می‌دانیم که چهار متغیر V،Q،NT وSA به دلیل وجود مجموعه‌ای از متغیرهای قبلی(متغیرهایی که مستقیما با این چهار متغیر تداخل دارند) شکست می‌خورند. این مسئله ما را به نگاهی دقیق‌تر به مفهوم «مجموعه تداخل» برای متغیری مثل NT وامی‌دارد: مجموعه‌ای از متغیرهای پیشین که منجر به عدم وجود راه‌حلی سازگار برای NT (و همه متغیرهای پس از آن) شدند. در مثال بالا این مجموعه شامل WA وNSW می‌باشد بنابراین الگوریتم باید از تاسمانیا صرف نظر کرده و به NSW عقبگرد کند. الگوریتم پرش به عقبی که از چنین مجموعه تداخلی استفاده می‌کند «تداخل- پرش به عقب هدایت شده[1]» نامیده می‌شود.

حال باید به روش‌های محاسبه چنین مجموعه تداخلی بپردازیم. روش کار بسیار ساده است. متوقف شدن یک شاخه از جستجو همواره به دلیل خالی شدن دامنه یک متغیر اتفاق می‌افتد که آن متغیر دارای مجموعه تداخلی استاندارد است. در مثال ما SA با شکست مواجه شده و دارای مجموعه تداخل {WA,NT,Q} است. ما به Q پرش به عقب کرده و Q مجموعه تداخل SA را به مجموعه تداخل خود({NT,NSW}) اضافه می‌کند (به جز خود Q) بنابراین مجموعه تداخل جدید به صورت {WA,NT,NSW} می‌شود. از آنجا که با توجه به مقداردهی قبلی {WA,NT,NSW} از Q راه‌حل رو به‌جلویی وجود ندارد بنابراین ما به NT (جدیدترین متغیر اضافه شده) عقبگرد می‌کنیم. NT مجموعه {WA, NT, NSW}−{NT} را به مجموعه تداخل خود ({WA}) اضافه می‌کند و مجموعه {WA, NSW} نتیجه می‌شود (همانطور که در پاراگراف قبلی گفته شد). حال همانطور که توقع داشتیم الگوریتم به NSW پرش به عقب می‌کند.

به طور خلاصه فرض کنید Xj متغیر جاری باشد، و فرض کنید conf(Xj) مجموعه تداخل آن باشد. اگر همه مقادیر ممکن برای Xj با شکست مواجه شوند آنگاه به متغیر Xi که جدیدترین متغیر اضافه شده به conf(Xj) است پرش به عقب کرده و مجموعه زیر پدید می‌آید:

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$

الگوریتم «تداخل-پرش به عقب هدایت شده» ما را دقیقا به نقطه مناسب در درخت جستجو می‌برد اما ما را از اشتباه در شاخه‌های دیگر درخت بازنمی‌دارد. «محدودیت‌های یادگیرنده»، CSP ها را با اضافه کردن محدودیت‌هایی که از این تداخل‌ها استنتاج می‌کنند، اصلاح می‌کند.

[1]- Conflict-Directed Back Jumping

۵-۳- جستجوی محلی برای مسائل ارضاء محدودیت

الگوریتم‌های جستجوی محلی (به بخش ۴-۳ مراجعه کنید) در حل بسیاری از مسائل CSP به راه‌حل بسیار موثری می‌رسند. آن‌ها از فرمول‌بندی کامل حالت استفاده می‌کنند: حالت اولیه مقداری دلخواه را به هریک از متغیرها نسبت می‌دهد و تابع پسین نیز در هر مرحله مقدار یک متغیر را تغییر می‌دهد. برای مثال در مسئله ۸-وزیر حالت اولیه ممکن است ترتیب تصادفی از ۸ وزیر در ۸ ستون شطرنج باشد و تابع پسین وزیری را برداشته و در خانه‌ای در همان ستون قرار می‌دهد. ویا در نوع دیگری از فرمول‌بندی می‌توان هر وزیر را در یک ستون قرار داده وجایگشتی از ۸ ردیف را به آن‌ها اختصاص دهیم و برای تولید حالت پسین شماره ردیف وزیرها را با هم عوض کنیم. ما تاکنون مثال‌هایی از جستجوی محلی برای حل مسائل CSP دیده‌ایم: مثل کاربرد آن در حل مسئله n-وزیر از طریق تپه‌نوردی.

در انتخاب یک مقدار برای یک متغیر بدیهی‌ترین ابتکار انتخاب مقداری است که حداقل تعداد تداخل را با متغیرهای دیگر ایجاد می‌کند. این الگوریتم و کاربرد آن در مسئله ۸-وزیر و ارزیابی کارایی آن به‌ترتیب در شکل‌های ۵-۸، ۵-۹ و۵-۵ نشان داده شده است.

الگوریتم «حداقل- تداخل» به‌طور شگفت‌آوری در بسیاری از مسائل CSP موثر است (مخصوصا زمانی که حالت اولیه مناسبی داده می‌شود) که ارزیابی کارایی آن در ستون آخر شکل ۵-۵ نشان داده شده است. تعجب‌آور است که در مسئله n- وزیر(اگر چیدمان اولیه وزیرها را در نظر نگیریم) زمان اجرایی الگوریتم «حداقل- تداخل» تقریبا مستقل از اندازه مسئله است. این الگوریتم حتی مسئله یک میلیون- وزیر را در ۵۰ مرحله حل می‌کند (بعد از مقداردهی اولیه). این مشاهده جالب توجه، در سال ۱۹۹۰ انگیزه‌های بسیار زیادی را برای انجام تحقیقات در زمینه جستجوی محلی و تشخیص تفاوت‌های بین مسائل آسان و سخت ایجاد کرد. به طور کلی مسئله n- وزیر برای الگوریتم جستجوی محلی آسان محسوب می‌شود زیرا راه‌حل‌های مسئله به طور انبوه در فضای حالت پراکنده شده‌اند. الگوریتم«حداقل- تداخل» برای مسائل سخت نیز خوب عمل می‌کند. به عنوان مثال این الگوریتم در برنامه‌ریزی مشاهدات تلسکوپ فضایی هابل استفاده شد و برنامه‌ریزی برای مشاهدات یک هفته‌ای را در ۱۰ دقیقه انجام داد (در صورتی که پیش از آن این کار سه هفته زمان می‌برد).

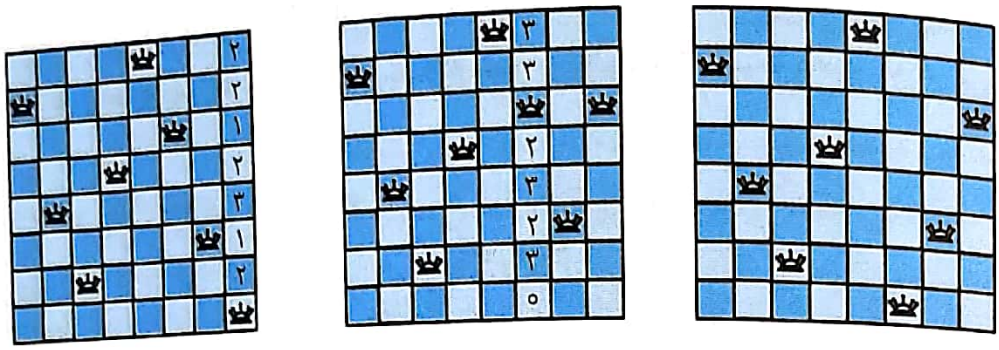
```
function Min-Conflicts(csp, max-steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
            max-steps, the number of steps allowed before giving up

    current ← an initial complete assignment for csp
    for i = ۱ to max-steps do
        if current is a solution for csp then return current
        var ← a randomly chosen, conflicted variable from Variables[csp]
        value ← the value v for var that minimizes Conflicts(var,v, current, csp)
        set var = value in current
    return failure
```

شکل ۵-۸ الگوریتم حداقل- تداخل برای حل CSP توسط جستجوی محلی. حالت اولیه یا به صورت تصادفی انتخاب می‌شود و یا براساس فرآیند مقداردهی حریصانه برای هر متغیر در نوبت خود مقداری با حداقل تداخل

انجام می‌شود. تابع CONFLICTS تعداد محدودیت‌هایی را که توسط یک مقدار خاص نقض می‌شود را محاسبه می‌کند.

فایده دیگر جستجوی محلی این است که هنگامی که مسئله تغییر پیدا می‌کند می‌تواند در مسائل «برخط» نیز به کار گرفته شود. این مسئله به‌خصوص در مسائل برنامه‌ریزی بسیار حائز اهمیت است. برنامه‌ریزی یک هفته یک هواپیما خود شامل هزاران پرواز و دهها هزارمقداردهی کارکنان است، از طرفی هوای نامناسب می‌تواند تمامی برنامه‌ریزی‌ها را غیرعملی و ناکارآمد جلوه دهد. می‌خواهیم برنامه را به شکلی بازسازی کنیم که حداقل تغییرات را اعمال کنیم. این عمل به راحتی از طریق یک الگوریتم جستجوی محلی و با در نظر گرفتن برنامه-ریزی فعلی به عنوان حالت اولیه قابل انجام است. این در حالی است که جستجوی عقبگرد با مجموعه جدیدی از محدودیت‌ها معمولا به زمان بیش‌تری نیازمند است و ممکن است جوابی با تفاوت‌های بسیار زیاد با برنامه‌ریزی فعلی بدست آورد.



شکل ۵-۹ یک راه‌حل دو مرحله‌ای با استفاده از الگوریتم حداقل- تداخل برای مسئله ۸- وزیر. در هر مرحله، یک وزیر برای مقدار دهی مجدد در ستون خود انتخاب می‌شود. تعداد تداخل‌ها (در این مسئله، تعداد وزیرهایی که به یکدیگر حمله می‌کنند) در هر خانه نمایش داده شده است. الگوریتم وزیر مورد نظر را به خانه‌ای که کمترین تعداد تداخل را دارد انتقال می‌دهد.

۵-۴- ساختار مسائل

در این بخش به روش‌هایی می‌پردازیم که در آن‌ها ساختار مسئله(همچون نمایش CSP به صورت گراف محدودیت) کمک بسیاری در پیدا کردن سریع راه‌حل مسئله می‌کنند. اکثر روش‌های مورد بحث، علاوه بر CSP در بسیاری از مسائل دیگر نیز قابل استفاده‌اند (مثل استدلال بر اساس احتمالات). یکی از روش‌های حل‌مسئله در مسائل دنیای واقعی تجزیه آن به تعداد زیادی زیرمسئله است. مجددا به شکل ۵-۱ (b) برای درک ساختار مسئله دقت کنید، یک مسئله در آن بسیار برجسته است: تاسمانیا به جزیره اصلی متصل نیست. به طور شهودی نیز بسیار بدیهی است که رنگ‌آمیزی تاسمانیا و جزیره اصلی دو زیر مسئله مستقل محسوب می‌شوند. بدین معنی که هر ترکیبی از پاسخ مسئله رنگ آمیزی جزیره اصلی و تاسمانیا می‌تواند پاسخی برای مسئله اصلی باشد. استقلال زیرمسئله‌ها را براحتی می‌توان با نگاه به مولفه‌های همبند گراف محدودیت تشخیص داد. هریک از مولفه‌ها متناظر با یک زیرمسئله CSP_i است. اگر مقداردهی S_i پاسخی برای CSP_i باشد آنگاه $\bigcup_i S_i$ پاسخی برای $\bigcup_i CSP_i$ خواهد بود. اما چرا این موضوع حائز اهمیت است؟ فرض کنید هریک از CSP ها دارای c (مقداری ثابت) متغیر از مقدار کل متغیرها n باشند. آنگاه n/c زیرمسئله وجود دارد که برای حل هریک از آن‌ها حداکثر زمان d^c لازم است. بنابراین زمان کل برابر $O(d^c n/c)$ خواهد بود (یعنی به‌صورت خطی برروی n). این درحالی است که پاسخ مسئله بدون تجزیه آن به زیر مسائل در زمان $O(d^n)$ قابل حل بود (یعنی نمایی برحسب

n). تقسیم یک CSP دوگانه با $n = ۸۰$ به چهار زیرمسئله با $c = ۲۰$، زمان الگوریتم در بدترین حالت را از چندین سال به زیر یک ثانیه کاهش می‌دهد.

زیرمسائل کاملا مستقل اگرچه بسیار رضایت‌بخش هستند اما کمیاب‌اند. در اغلب موارد زیرمسائلCSP، همبند هستند. گراف محدودیت یک CSP در ساده‌ترین حالت، یک درخت است(هر دو متغیر حداکثر با یک مسیر با یکدیگر ارتباط دارند) که در شکل ۵-۱۰(a) در قالب مثالی نشان داده شده است. می‌توان نشان داد که هر *CSP* با ساختار درختی در زمان خطی برحسب تعداد متغیرها قابل حل است. این الگوریتم شامل مراحل زیر است:

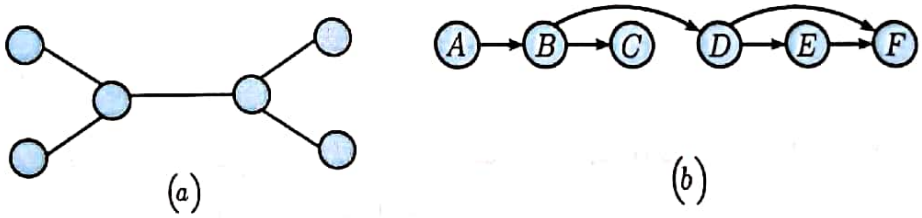
۱. ابتدا متغیری را به‌عنوان ریشه انتخاب می‌کنیم. سپس متغیرها را از ریشه تا برگ طوری مرتب می‌کنیم که والد هر گره قبل از فرزند خود در درخت باشد(به شکل ۵-۱۰(b) دقت کنید) متغیرها را به‌ترتیب به صورت $X_1,\ldots X_n$ برچسب‌گذاری می‌کنیم. حال همه‌ی متغیرها(به جز ریشه) دقیقا یک والد دارند.

۲. برای j از n تا ۲، الگوریتم «سازگاری یال» را برای یال (X_i,X_j) اعمال کنید که در آن X_i والدX_j است(اگر لازم بود مقدارهایی را از $DOMAIN[X_i]$ حذف کنید.)

۳. برای j از ۱ تا n، مقداری سازگار با مقدار نسبت داده شده به X_i به X_j نسبت دهید که در آن X_i والد X_j است.

در اینجا باید به دو نکته کلیدی توجه کنیم. اولا اینکه بعد از مرحله ۲ CSP یال- سازگار است بنابراین مقداردهی متغیرها در مرحله ۳ نیاز به عقبگرد ندارد. ثانیا اینکه با بررسی سازگاری یال در جهت برعکس در مرحله ۲ الگوریتم مطمئن می‌شود که مقادیر حذف شده نمی‌تواند سازگاری یال‌هایی که تا به حال پردازش شده‌اند را به خطر بیاندازد. کل الگوریتم در زمان $O(nd^۲)$ قابل اجرا است.

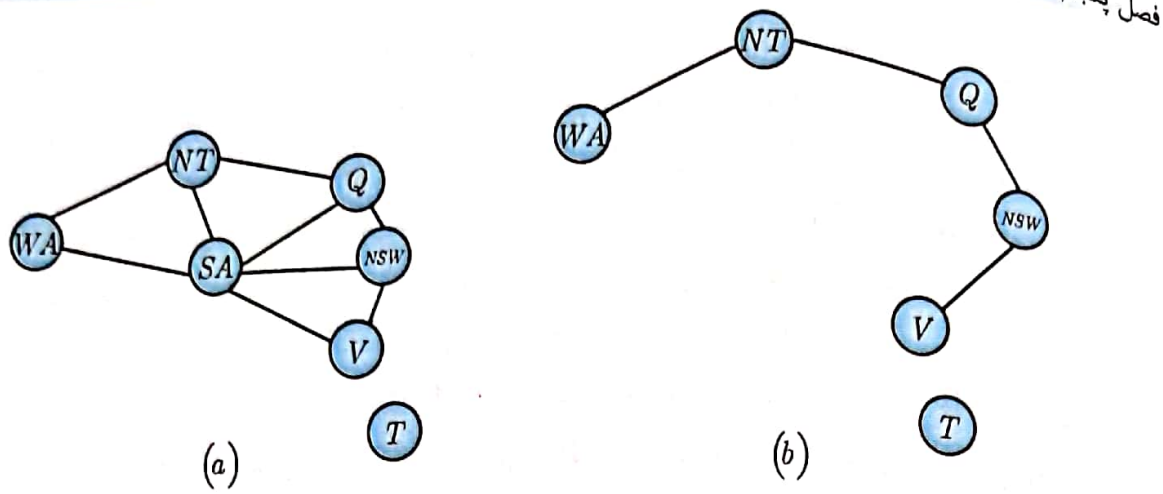
حال که ما الگوریتم کارایی برای درخت یافتیم، بدنبال راهی برای تبدیل گراف محدودیت به درخت می‌گردیم. دو راه اولیه برای این کار وجود دارد یکی براساس حذف گره‌ها و دیگری براساس متلاشی کردن گره‌ها است.



شکل ۵-۱۰ (a) گراف محدودیت یک درخت با ساختار CSP. (b) یک ترتیب خطی از متغیرهای سازگار با درختی که ریشه‌اش A است.

در روش اول مقادیری را به بعضی از متغیرها نسبت داده به نحوی که بقیه متغیرها تشکیل یک درخت دهند. به گراف محدودیت برای استرالیا توجه کنید (که درشکل ۵-۱۱(a) مجددا نشان داده شده است). اگر بتوانیم سوث استرالیا (SA) را حذف کنیم، آنگاه گراف تبدیل به درخت می‌شود(مثل قسمت (b)). خوشبختانه می‌توانیم این کار را با در نظر گرفتن یک مقدار ثابت برای SA و حذف مقادیر ناسازگار با آن از دامنه متغیرهای دیگر انجام دهیم.

بعد از حذف SA و محدودیت‌های آن، پاسخ‌های مسئلهCSP باید با مقدار انتخاب شده برای SA سازگار باشد (توجه داشته باشید که این موضوع در مورد CSP های دوگانه صادق است اما برای محدودیت‌های مرتبه بالاتر بسیار پیچیده‌تر خواهد بود). بنابراین می‌توانیم درخت حاصله را از طریق الگوریتم فوق‌الذکر حل کرده و پاسخ کل مسئله را بدست آوریم. اگرچه، در حالت کلی ممکن است مقدار انتخاب شده برای SA نادرست باشد و شاید نیاز باشد همه‌ی مقادیر ممکن را روی آن امتحان کنیم.



شکل ۵-۱۱ (a) گراف محدودیت اصلی شکل ۵-۱. (b) گراف محدودیت بعد از حذف SA.

الگوریتم کلی به صورت زیر است:

۱. زیر مجموعه‌ای مثل S را از VARIABLE[csp] انتخاب کنید به طوری که گراف محدودیت بعد از حذف S تبدیل به یک درخت شود. به مجموعه S «حلقه برشی» گوییم.

۲. به ازای هر مقداردهی ممکن به متغیرها در S که تمامی محدودیت‌ها را در S ارضاء می‌کند:

(a) از دامنه متغیرهای باقی مانده مقادیری را که با مقادیر نسبت داده شده در S ناسازگار است حذف کنید.

(b) اگر CSP باقی‌مانده پاسخی دارد، آن پاسخ را بهمراه مقداردهی انجام شده در S به عنوان خروجی برگردانید.

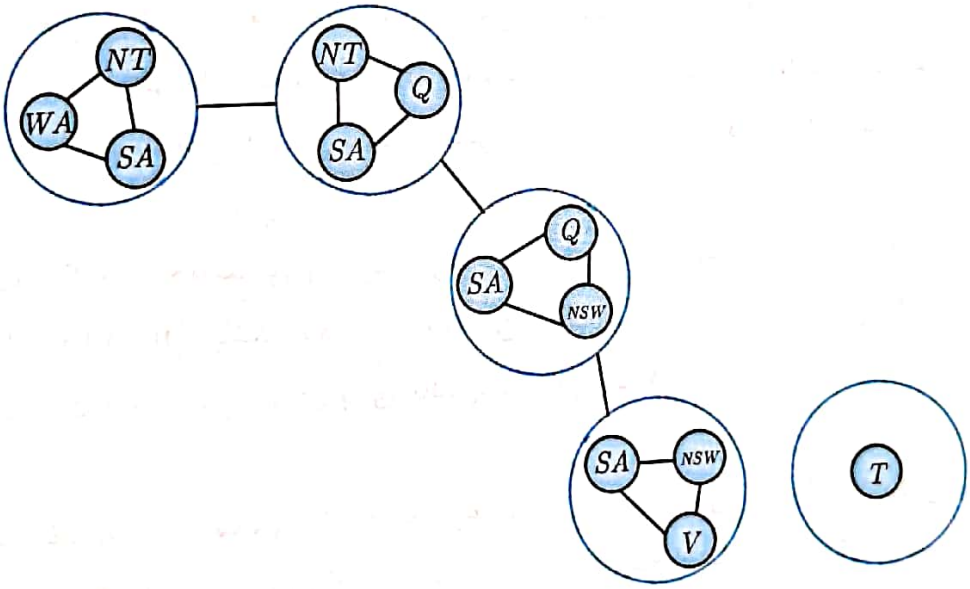
اگر اندازه حلقه برشی c باشد، آنگاه کل زمان اجرای الگوریتم برابر $O(dc.(n-c)d^2)$ می‌شود. اگر گراف «تقریباً درخت» باشد آنگاه c کوچک خواهد بود و صرفه جویی روی عقبگردها بسیار زیاد خواهد بود. در بدترین حالت c ممکن است به اندازه $(n-2)$ باشد. یافتن کوچکترین حلقه برشی از نوع مسائل NP- سخت است اما الگوریتم‌های تقریبی کارآمد بسیار زیادی برای یافتن آن وجود دارد. الگوریتم کلی این مسئله «شرط برش» نامیده می‌شود.

روش دوم براساس تجزیه گراف محدودیت به مجموعه‌ای از زیرمسئله‌های مرتبط و تشکیل یک درخت است. هریک از این زیرمسئله‌ها به صورت مستقل حل شده و سپس پاسخ آن‌ها با یکدیگر ترکیب می‌شود. همچون بسیاری از الگوریتم‌های تقسیم و حل[1] اگر زیرمسئله‌ها خیلی بزرگ نباشند این الگوریتم به خوبی عمل می‌کند. شکل ۵-۱۲ تجزیه درخت مسئله رنگ آمیزی نقشه به پنج زیرمسئله را نشان می‌دهد. تجزیه درخت باید دارای سه ویژگی زیر باشد:

- همه‌ی متغیرهای مسئله اصلی باید حداقل در یکی از زیرمسئله‌ها ظاهرشوند.
- اگر دو متغیر توسط یکی از محدودیت‌ها در مسئله اصلی به هم متصل شده باشند آنگاه حتما باید حداقل در یکی از زیر مسئله‌ها با هم ظاهر شوند.
- اگر متغیری در دو زیرمسئله در درخت ظاهر شود آنگاه باید در همه‌ی زیر مسئله‌ها در طول مسیر ارتباط این دو زیر مسئله نیز ظاهر شود.

[1]- Divide and conquer

دو شرط اول تضمین می‌کند که همه‌ی متغیرها و محدودیت‌ها در تجزیه ظاهر شده‌اند. شرط سوم کمی فنی‌تر بنظر می‌رسد و به این محدودیت اشاره می‌کند که همه متغیرها باید در تمامی زیرمسئله‌ها دارای مقدار یکسانی باشد (لینک‌های ارتباط‌دهنده زیر مسئله این محدودیت را اعمال می‌کنند). برای مثال SA در هر چهار زیر مسئله مرتبط شکل ۵-۱۲ ظاهر شده است. از شکل ۵-۱۱ به‌راحتی می‌توان دریافت که این نوع تجزیه بسیار منطقی به نظر می‌رسد. هرکدام از زیرمسئله‌ها را به صورت مستقل حل می‌کنیم. اگر هریک از زیرمسئله‌ها فاقد پاسخ باشند نتیجه می‌گیریم که مسئله کلی پاسخی ندارد. اگر همه زیر مسئله ها را حل کنیم آنگاه سعی می‌کنیم جواب کلی مسئله را بیابیم. ابتدا هر زیر مسئله را به صورت یک «متغیر ماوراء» در نظر میگیریم که دامنه آن مجموعه تمامی پاسخ‌های زیرمسئله است. برای مثال چپ‌ترین زیرمسئله در شکل ۵-۱۲ مسئله رنگ آمیزی نقشه با سه متغیر است که شش پاسخ دارد که یکی از آن‌ها {WA= قرمز،SA= آبی،NT= سبز} است. آنگاه نحوه اتصال این زیر مسئله‌ها را به صورت یک محدودیت در نظر می‌گیریم و آن را با الگوریتم ارائه شده در بالا حل می‌کنیم. محدودیت بین زیرمسئله‌ها به این صورت تعریف می‌شود که پاسخ زیرمسئله‌ها باید برروی متغیرهای مشترک توافق داشته باشند. برای مثال فرض کنید راه‌حل {WA=قرمز، SA=آبی، NT= سبز} برای زیرمسئله اول داده شده است بنابراین تنها راه حل سازگار برای زیرمسئله بعدی {SA= آبی، NT= سبز، Q= قرمز} خواهد بود.



شکل ۵- ۱۲ درخت تجزیه گراف محدودیت شکل ۵-۱۱(a).

هر گراف محدودیت را می‌توان به طرق مختلف تجزیه کرد، اما مناسب ترین تجزیه، تجزیه‌ای است که در آن زیرمسائل کوچکتری تشکیل شود. عرض درخت گراف تجزیه شده، یک واحد کمتر از اندازه بزرگترین زیرمسئله است. عرض درخت خود گراف برابر مینیمم عرض درخت در میان تمامی درخت‌های تجزیه شده است. اگر یک گراف دارای عرض w باشد (و درخت تجزیه شده تشکیل شده را داشته باشیم) آنگاه مسئله در زمان $O(nd^{w+1})$ قابل حل است. لذا CSPهایی با گراف محدودیتی با عرض درخت کران‌دار در زمان چندجمله‌ای قابل حل هستند. متاسفانه یافتن تجزیه‌ای با کمترین عرض درخت NP- سخت است اما روش‌های ابتکاری وجود دارند که در عمل خوب جواب می‌دهند.

۵-۵- خلاصه

- مسائل ارضاء محدودیت (CSP) شامل متغیرهایی می‌شوند که محدودیت‌هایی برروی آن‌ها اعمال می‌شود. بسیاری از مسائل در دنیای واقعی به‌صورت CSP بیان می‌شوند. ساختار یک CSP توسط گراف محدودیت نمایش داده می‌شود.
- جستجوی عقبگرد ساده، نوعی از الگوریتم عمق-اول است که عموما برای حل مسائل CSP به کار می‌رود.
- مینیمم مقادیر باقی‌مانده، و ابتکار درجه، روش‌هایی مستقل از دامنه هستند که در آن‌ها درباره انتخاب متغیر بعدی در جستجوی عقبگرد تصمیم‌گیری می‌شود. ابتکارکمترین-محدودکننده-مقدار به ما در ترتیب مقداردهی متغیرها کمک می‌کند.
- با انتشار نتیجه مقداردهی‌های ناقص فعلی، الگوریتم عقبگرد فاکتور انشعاب مسئله را به شدت کاهش می‌دهد. الگوریتم بررسی پیشرو، ساده‌ترین راه برای انجام این عمل است. اجراء سازگاری یال تکنیک قوی‌تری است ولی زمان اجرای آن بسیار پرهزینه‌تر است.
- عقبگرد زمانی اتفاق می‌افتد که هیچ مقداردهی مجازی برای یک متغیر وجود نداشته باشد. تداخل-عقبگرد هدایت شده به طور مستقیم به علت اصلی مشکل عقبگرد می‌کند.
- جستجوی محلی که از ابتکار حداقل- تداخل استفاده می‌کند برروی مسائل ارضاء محدودیت با موفقیت اعمال شده است.
- پیچیدگی حل یک مسئله CSP به شدت به ساختار گراف محدودیت آن وابسته است. مسائلی با ساختار درخت در زمان خطی قابل حل هستند. روش شرط برش می‌تواند یک CSP را در حالت کلی به یک مسئله با ساختار درختی تبدیل کند. این روش در صورتی که برش کوچکی پیدا شود بسیار کارا است.
- روش‌های «تجزیه درخت» CSP را به زیر مسئله‌هایی از درخت تبدیل می‌کند و در صورتی که «عرض درخت» گراف محدودیت کوچک باشد این روش بسیار کارا است.

۵-۶- تمرین‌ها

۵-۱ واژه‌های زیر را تعریف کنید: مسئله ارضای محدودیت، محدودیت، جستجوی عقبگرد، سازگاری یال، پرش به عقب، حداقل تداخل.

۵-۲ برای مسئله رنگ‌آمیزی گراف در شکل ۵-۱ چند راه‌حل وجود دارد؟

۵-۳ توضیح دهید که چرا در جستجوی CSP انتخاب بیش‌ترین محدودشونده و انتخاب کمترین مقدار محدودکننده ابتکار خوبی محسوب می‌شود؟

۵-۴ به مسئله ساختن پازل کلمات متقاطع توجه کنید: که در آن کلمات را درون شبکه‌ای مستطیلی شکل قرار می‌دهیم. این شبکه (که خود قسمتی از مسئله است) مشخص می‌کند که کدام خانه‌ها خالی و کدام سایه زده شده است. فرض کنید لیستی از کلمات (مثلا یک دیکشنری) داده شده است و وظیفه ما این است که با استفاده از زیرمجموعه‌ای از کلمات از این لیست خانه‌های خالی را پر کنیم. این مسئله را به طور دقیق به دو شکل فرمول‌بندی کنید:

الف) به عنوان یک مسئله جستجو. یک الگوریتم جستجوی مناسب انتخاب کرده، و تابع ابتکاری مربوطه را تعیین کنید (اگر احساس می‌کنید به آن نیاز دارید). کدام یک بهتر است، پرکردن جاهای خالی به صورت کلمه به کلمه (در هر مرحله یک کلمه را امتحان می‌کنیم) و یا حرف به حرف (در هر مرحله یک حرف را امتحان می‌کنیم)؟

ب) به عنوان یک مسئله ارضاء محدودیت. متغیرها باید کلمات باشند یا حروف؟ کدام فرمول‌بندی از نظر شما بهتر است؟ چرا؟

۵-۵ هریک از موارد زیر را به صورت مسئله ارضاء محدودیت فرمول‌بندی کنید:

الف) کف‌پوش مستطیلی: پوشش کف پوش مستطیل شکل با استفاده از خانه‌های مستطیلی کوچک‌تر به طوری که هم‌پوشانی نداشته باشند.

ب) زمان‌بندی کلاس: تعداد ثابتی استاد و کلاس وجود دارد، لیستی از کلاس‌ها و جدول زمانی ممکن برای کلاس‌ها باید پیشنهاد شود. هر استاد مجموعه‌ای از کلاس‌ها را می‌تواند تدریس کند.

۵-۶ مسئله رمزنگاری شکل ۵-۲ را با استفاده از الگوریتم‌های عقبگرد، بررسی پیشرو، و ابتکارات MRV و کمترین- محدودکننده- مقدار حل کنید.

۵-۷ از الگوریتم AC-۳ استفاده کنید و نشان دهید که الگوریتم سازگاری یال قادر است ناسازگاری مقداردهی ناقص {WA= قرمز، V= آبی} شکل ۵-۱ را تشخیص دهد.

۵-۸ در بدترین حالت، پیچیدگی اجرای الگوریتم AC-۳ در درختی با ساختار CSP چقدر است؟

۵-۹ AC-۳ هرگاه مقداری از دامنه X_i حذف شود، مجددا یال (X_k,X_i) را در صف قرار می‌دهد (حتی اگر همه مقادیر X_k با بعضی از مقادیر باقی‌مانده X_i سازگار باشد). فرض کنید، برای هر یال (X_k,X_i)، تعداد مقادیر باقی‌مانده X_i که با هریک از مقادیر X_k سازگار- یال است را بشماریم. توضیح دهید که چگونه می‌توان این مقادیر را به گونه‌ای کارا به‌روز رسانی کنیم، بنابراین نشان دهید که سازگاری یال در زمان کلی $O(n^2d^2)$ قابل اجراست.

۵-۱۰ نشان دهید که چگونه یک محدودیت سه‌تایی مثل $A+B=C$ می‌تواند به محدودیت‌های دوتایی و یک متغیر کمکی تبدیل شود. می‌توانید دامنه‌ها را به صورت متناهی در نظر بگیرید(راهنمایی: متغیر جدیدی را در نظر بگیرید که مقادیر آن جفت‌های مقادیر دیگر است، و محدودیت‌هایی مثل «X عنصر اول جفت Y است» را در نظر بگیرید). سپس نشان دهید چگونه محدودیت‌هایی با بیش از سه متغیر می‌توانند به طور مشابهی عمل کنند. در انتها نشان دهید که چگونه می‌توان محدودیت‌های یگانه را با تغییر دامنه‌های متغیرهای دیگر حذف کرد که در این صورت، اثبات این موضوع که هر CSP می‌تواند به صورت CSP هایی با محدودیت‌های دوگانه تبدیل شود کامل می‌شود.

۵-۱۱ فرض کنید گرافی دارای یک حلقه برشی باشد که دارای حداکثر مقدار k گره در آن باشد. الگوریتم ساده‌ای برای پیدا کردن حلقه برشی مینیمم ارائه دهید که زمان اجرای آن برای یک CSP با n متغیر بیش‌تر از $O(n^k)$ نباشد. روش‌های پیدا کردن حلقه برش مینیمم تقریبی را در زمان خطی بر حسب اندازه حلقه برش جستجو کنید. آیا وجود چنین الگوریتمی پیدا کردن حلقه برش را در عمل ممکن می‌سازد؟

۵-۱۲ معمای منطق زیر را در نظر بگیرید: در پنج خانه، که هر کدام رنگ متفاوتی دارند، ۵ نفر با ملیت‌های مختلف زندگی می‌کنند، هرکدام از آن‌ها یک مارک خاص سیگار، نوشیدنی خاص و حیوان خاصی را دوست دارند. با در نظر گرفتن این واقعیت‌ها سوال این است که زبرا کجا زندگی می‌کند و در کدام خانه آب می‌نوشد؟

- مرد انگلیسی در خانه قرمز زندگی می‌کند.
- اسپانیایی صاحب سگ است.
- نروژی در اولین خانه‌ی سمت چپ زندگی می‌کند.
- مردی که سیگار چسترفیلد می‌کشد در خانه‌ی بعد از خانه‌ی مرد روباه‌دار زندگی می‌کند.
- نروژی در خانه‌ی بعد از خانه‌ی آبی زندگی می‌کند.
- کسی که سیگار ونیستون می‌کشد مالک حلزون است.
- کسی که سیگار لاکی استرایک می‌کشد، آب پرتغال می‌نوشد.
- اوکراینی چای می‌نوشد.
- ژاپنی سیگار پارلیانتس می‌کشد.
- کولز در خانه‌ی بعد از خانه‌ای که اسب نگهداری می‌کند سیگار می‌کشد.
- کولز در خانه‌ی زرد سیگار می‌کشد.
- قهوه در خانه‌ی سبز نوشیده می‌شود.
- خانه‌ی سبز بلافاصله سمت راست (راست شما) خانه‌ی شیری رنگ است.
- شیر در خانه‌ی وسطی نوشیده می‌شود.

شکل‌های متفاوت بازنمایی این مسئله را به عنوان CSP بحث کنید. چرا یک بازنمایی را به دیگری ترجیح می‌دهید.

تست‌های طبقه‌بندی‌شده فصل پنجم

۱- در مسائل ارضاء محدودیت‌ها (Constraint satisfaction problems(CSP یک تابع مکاشفه‌ای مناسب برای انتخاب متغیر بعدی برای مقداردهی کدام است؟ (کامپیوتر ۸۴)

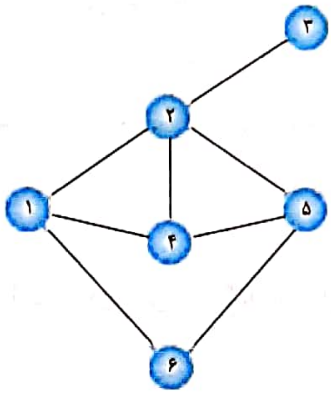
۱) انتخاب متغیری با بزرگترین دامنه مقادیر مجاز

۲) انتخاب متغیری با کوچکترین دامنه مقادیر مجاز

۳) انتخاب متغیری که کمترین میزان مقادیر مجاز را از دامنه سایر متغیرها حذف کند.

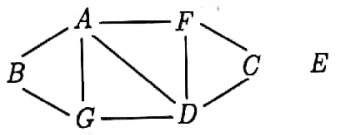
۴) انتخاب متغیری که بیش‌ترین میزان مقادیر مجاز را از دامنه سایر متغیرها حذف می‌کند.

۲- در صورتی که بخواهیم با استفاده از روش ارضای محدودیت گراف مقابل را با سه رنگ، رنگ‌آمیزی نمائیم پس از رنگ‌آمیزی رئوس ۱ و ۲ بهتر است کدام راس بعدا رنگ‌آمیزی شود؟ (کامپیوتر ۸۷)



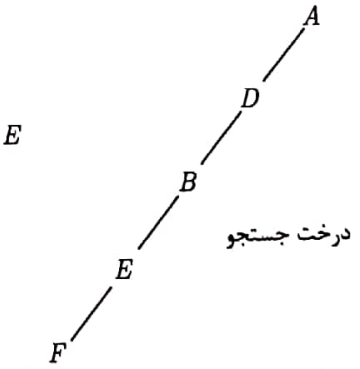
۱) ۵ ۲) ۶ ۳) ۳ ۴) ۴

۳- یک مسئله CSP دارای گراف محدودیت مقابل است. فرض کنید فضای حالت با یک جستجوی عمق اول پیمایش شود و با برخورد به یک شکست در مقداردهی به F در لحظه‌ای که درخت جستجوی آن نشان داده شده نیاز به عقب‌گرد باشد. در عقب‌گرد براساس مجموعه تناقض (Conflict set) به کدام گره باز خواهیم گشت؟ (کامپیوتر ۸۸)



گراف محدودیت

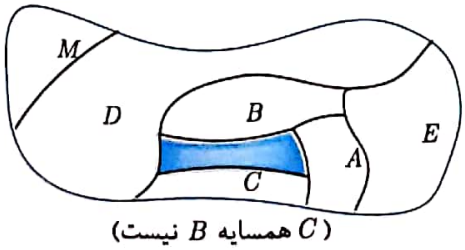
E



درخت جستجو

۱) A ۲) B ۳) D ۴) E

۴- در یک مسئله رنگ‌آمیزی نقشه به شکل زیر می‌خواهیم هیچ دو کشور همسایه‌ای هم‌رنگ نباشند و کشورهای A، B، C، D، E و M به یکی از سه رنگ قرمز، سبز و آبی رنگ شوند. فرض کنید در میانه یک جستجوی ارضاء محدودیت (CSP) برای این مسئله، به کشور A رنگ قرمز و به کشور C رنگ آبی را نسبت داده باشیم. حال اگر بخواهیم به کشور M رنگ قرمز را نسبت دهیم. کدام تست زیر بروز تناقض و نیاز به عقب‌گرد را پیش‌بینی می‌کند؟ (کامپیوتر ۸۹)



(C همسایه B نیست)

۱) forward checking

۲) path consistency

۳) node consistency

۴) تناقضی کشف نخواهد شد.

پاسخنامه تشریحی تست‌های فصل پنجم

۱) گزینه ۲ درست است.

به این تابع مکاشفه‌ای، MRV گویند.

۲) گزینه ۴ درست است.

برای انتخاب راس بعدی، باید راسی را انتخاب کنیم که کمترین دامنه مجاز را داشته باشد. با توجه به اینکه بعد از رنگ‌آمیزی هریک از راس‌های ۱ و۲ دو رنگ از دامنه راس ۴ حذف شده است بنابراین دامنه این راس تنها یک رنگ دارد و بنابراین کمترین دامنه مجاز را دارد.

۳) گزینه ۳ درست است.

نکته ۱: در حالت کلی، جستجوی عقب‌گرد، از الگوریتم عمق- اول برای مقداردهی به متغیرها استفاده می‌کند و درصورت بروز مشکل عقب‌گرد می‌کند.

نکته ۲: نوع دیگری از جستجوی عقب‌گرد وجود دارد که همچنان از الگوریتم عمق-اول برای مقداردهی به متغیرها استفاده می‌کند اما به هنگام بروز مشکل از مجموعه‌ای به نام Conflict set استفاده کرده و عمل بازگشت به عقب را به صورت هوشمندانه‌تری انجام می‌دهد.

نکته ۳: به طور کلی، مجموعه Conflict set برای متغیر X مجموعه‌ای از متغیرهای مقداردهی شده است، که در گراف محدودیت به آن متغیر متصل شده‌اند.

نکته ٤: در روش فوق‌الذکر در صورت بروز مشکل، الگوریتم به جدیدترین گره مقداردهی شده در مجموعه Conflict set برمی‌گردد.

با توجه به توضیحات داده شده، واضح است که الگوریتم به گره D برگشت به عقب می‌کند. زیرا گره‌های B و E اصولا در مجموعه Conflict set گره F نیستند و از طرفی از بین دو گره A و D چون گره D نزدیک‌ترین گره مقداردهی شده است، بنابراین گره D انتخاب می‌شود.

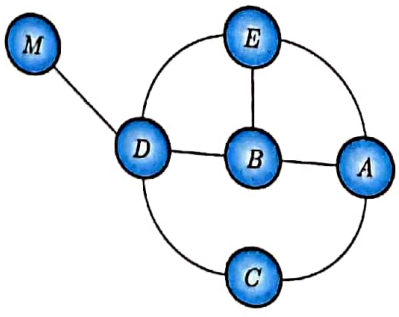
۴) گزینه ۲ درست است.

گزینه‌ی ۱ نادرست است. از آنجا که دامنه‌ی هیچ یک از متغیرها خالی نیست، بنابراین Forward checking ناسازگاری را تشخیص نمی‌دهد.

نکته ۱: ۳- سازگاری بدین معنی است که به‌ازای هر مقداردهی سازگار ۲ متغیر همسایه، متغیر سومی وجود داشته باشد که قابل مقداردهی بوده و با مقداردهی‌های قبلی سازگار باشد.

۳- سازگاری و بالاتر (یعنی ۴- سازگاری، ۵- سازگاری، ... سازگاری ـ مسیری گفته می‌شود

گراف محدودیت بدین‌ترتیب است:



با توجه به توضیحات بالا، به‌ازای مقداردهی سازگار دو متغیر همسایه، باید مقداردهی سازگار دیگری برای متغیر سوم وجود داشته باشد. با توجه به گراف محدودیت ملاحظه می‌کنید که اگر $A =$ قرمز و $C =$ آبی و $M =$ قرمز باشد آنگاه مثلث $B - D - E$ داریم که هر سه با هم محدودیت دارند ولی حداکثر ۲ رنگ در دامنه آنها معتبر است (دامنه D فقط یک رنگ دارد). با ۳− سازگاری می‌توان این ناسازگاری را تشخیص داد.

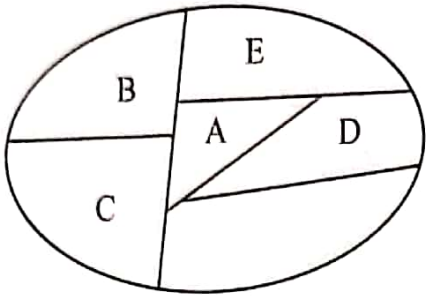
گزینه‌ی ۳ نادرست است.

نکته ۲: **۱- سازگاری بدین معنی است که هر متغیر به تنهایی سازگار است یعنی دامنه آنها تهی نباشد. به این نوع سازگاری Node consistency نیز می‌گویند. در حالی‌که حالت جاری گراف محدودیت متغیرها به‌تنهایی سازگارند.**

گزینه ۴ نادرست است. با توجه به توضیحات بالا، ناسازگاری وجود دارد.

تست‌های تألیفی

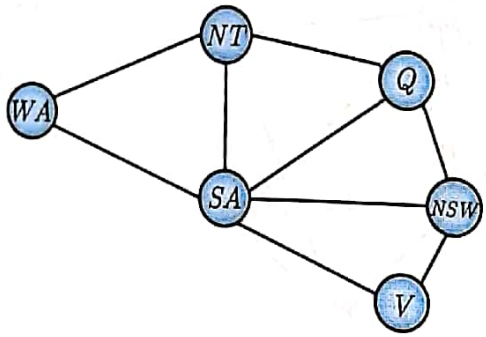
۱- مسئله رنگ آمیزی با k رنگ برای نواحی شکل زیر در نظر بگیرید. مقدار k حداقل بایستی چند باشد تا بدون backtracking بتوان مساله را حل کرد: (A و F همسایه هستند)



۱) ۲
۲) ۳
۳) ۴
۴) ۵

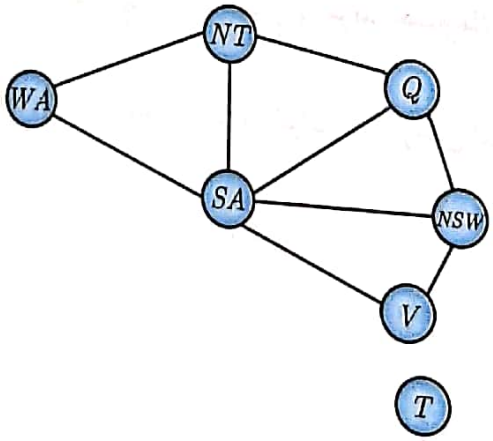
۲- مساله رنگ آمیزی زیر با سه رنگ را در نظر بگیرید. و فرض کنید که مقدار رنگ WA و NSW هر دو قرمز است. بهترین روشی که می‌توان این ناسازگاری را پیدا کرد، چیست؟

۱) AllDiff Heuristic
۲) Resource Constraint heuristic
۳) Arc Consistency
۴) Forward Checking

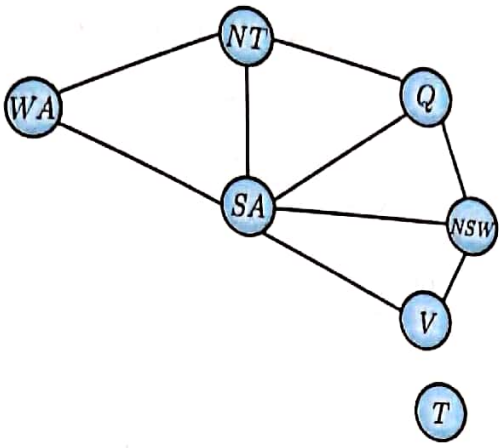


۳- می‌خواهیم گراف زیر را با سه رنگ رنگ‌آمیزی نماییم. هیچ دو رنگ مشابهی نمی‌توانند کنار یکدیگر باشند. پس از مقدار دهی NSW و SA کدام یک از گره‌ها باید مقداردهی شوند؟

۱) WA
۲) V
۳) Q
۴) NSW

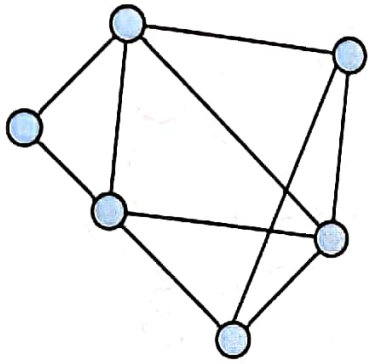


۴- می‌خواهیم گراف زیر را با سه رنگ رنگ‌آمیزی نماییم. هیچ دو رنگ مشابهی نمی‌توانند کنار یکدیگر باشند. اگر مقدار NT = red و V = green باشد، آنگاه این ناسازگاری با کدامیک از روشهای زیر قابل تشخیص است:



۱) Forward checking
۲) Arc-consistency
۳) Path-consistency
۴) موارد ۱ و ۲

۵- مسئله رنگ آمیزی گراف زیر بطوریکه هیچ دو گره ی همسایه همرنگ نباشند، را در نظر بگیرید. برای آنکه مسئله حداقل یک پاسخ درست داشته باشد به چند رنگ نیاز است. برای آنکه برای حل مسئله به backtrack نیاز نباشد، حداقل چند رنگ مورد نیاز است:



۱) ۳و ۴
۲) ۳و ۳
۳) ۴ و ۴
۴) ۴و ۵

۶- چند گزینه زیر درست است:

مسائل ارضای محدودیت که گراف آنها درختی است، را میتوان با هزینه خطی نسبت به تعداد متغیرها حل کرد.

مسائل ارضای محدودیت که گراف آنها درختی است، را میتوان با هزینه خطی نسبت به تعداد مقادیر دامنه‌ی متغیرها حل کرد.

مسائل ارضای محدودیت که گراف آنها درختی نیست، را میتوان با هزینه چندجمله ای با استفاده از ادغام متغیرها به گراف درختی تبدیل کرد.

می‌توان الگوریتم AC۳ را برای مسائلی ارضای محدودیت که گراف آنها درختی است، را میتوان با هزینه خطی نسبت به تعداد متغیرها اجرا کرد.

۱) ۱ گزینه
۲) ۲ گزینه
۳) ۳ گزینه
۴) ۴ گزینه

پاسخ تشریحی تست‌های تألیفی

۱) با رسم گراف محدودیت مربوط به مساله، متوجه می‌شویم که عرض گراف، ۲ است . (عرض گراف بدینصورت حاصل میشود که از چیدن گره ها در یک خط به ترتیبی که گره با بالاترین درجه در سمت چپ قرار گیرد و حداکثر تعداد یالهایی که برای یک گره به سمت چپ آن گره حرکت کرده است، محاسبه میشود. به این عدد گراف می‌گویند.) برای آنکه به $Backtrack$ نیاز نباشد بایستی k حداقل یک واحد بیشتر باشد. یعنی گزینه ۲ درست است.

۲) گزینه ۱ درست است

با Heuristic اول و این موضوع که سه گره SA, Q, NT هر سه دارای فقط دو رنگ هستند، طبق اصل لانه کبوتری چنین چیزی ممکن نیست.

۳) گزینه ۳ درست است.

پس از مقداردهی به NSW, SA، دامنه ی هر دو گره Q , V برابر یک است . که کمترین مقدار است. ولی Q دارای درجه بیشتر است ، بنابراین طبق MRV ، هر دو گره V, Q برابر هستند ولی در حالت مساوی ، طبق most constraint variable (Degree heuristic)، Q اولویت دارد

۴) گزینه ۳ درست است.

در حالت مذکور، سه گره SA, Q, NSW که بایکدیگر تداخل دارند، دارای دامنه به طول ۲ هستند ولی همه‌ی یالهای آنها دارای arc- Consistency هستند. در واقع این ناسازگاری نه با forward checking و نه با arc-consistency قابل تشخیص نیست و نیاز به کنترل ناسازگاریهای درجه بالاتر از یال یعنی (– Path consistency) است.

۵) گزینه ۱ درست است.

با توجه به آنکه عرض گراف ۳ است. (عرض گراف بدینصورت حاصل میشود که از چیدن گره ها در یک خط به ترتیبی که گره با بالاترین درجه در سمت چپ قرار گیرد و حداکثر تعداد یالهایی که برای یک گره به سمت چپ آن گره حرکت کرده است، محاسبه میشود. به این عدد عرض گراف می‌گویند.) برای آنکه مسئله بدون $Backtrack$ حل شود بایستی یک عدد بزرگتر از عرض انتخاب شود یعنی ۴ رنگ. ولی خود مسئله واضح است که با ۳ رنگ حل میشود.

۶) گزینه ۲ درست است.

مسائل ارضای محدودیت که گراف آنها درختی است، را میتوان با AC۳ با هزینه $O(n.d^2)$ حل کرد ، پس گزینه اول و چهارم درست است. گزینه های دیگر غلط است.

فصل ششم

بازی‌ها

۶-۱ بازی‌ها

در فصل ۲ محیط‌های چند عاملی[1] را معرفی کردیم. محیط‌هایی که در آن‌ها عامل، «کنش» عامل‌های دیگر و تاثیر آن‌ها برروی خود را تحت نظر می‌گیرد. از طرفی غیرقابل پیش‌بینی بودن این عامل‌ها، احتمالات و رویدادهای غیرمترقبه‌ای را به فرآیند حل مسئله می‌افزاید(همانطور که در فصل ۳ توضیح داده شد). همچنین تفاوت‌های میان محیط‌های چندعامله «همکاری[2]» و«رقابتی[3]» در فصل ۲ به تفضیل شرح داده شد. محیط‌های رقابتی که در آن‌ها اهداف عامل‌ها در تضاد با یکدیگر هستند منجر به مسائل جستجوی رقابتی[4] (گاهی به آن «بازی» نیز گویند) می‌شود.

نظریه بازی‌ها در علم ریاضی (که شاخه‌ای از علم اقتصاد نیز محسوب می‌شود) به محیط‌های چندعاملی (همکاری یا رقابتی) به دید یک بازی نگاه می‌کند البته با فرض اینکه هریک از عامل‌ها اثر «معنی‌داری» بر دیگران داشته باشد. در هوش مصنوعی اغلب بحث بر روی نوع خاصی از بازی‌ها است که به صورت دونفره، نوبتی، «مجموع- صفر با اطلاعات کامل» هستند. به عبارت دیگر محیط‌هایی قطعی و قابل مشاهده که در آن‌ها کنش‌های دوعامل به صورت یکی در میان اجرا شده و مقادیر «تابع سودمندی[5]» آن‌ها همواره در انتهای بازی مساوی و متضاد یکدیگرند. به‌عنوان مثال اگر بازیکنی در بازی شطرنج برنده شود مقدار ۱+ و طرف مقابل لزوماً بازنده و مقدار ۱- می‌گیرد. این تضاد بین توابع سودمندی عامل‌ها منجر به جو رقابتی برای آن‌ها می‌شود. البته ما در این فصل درباره بازی‌های «چندنفره»، بازی‌های «مجموع- غیرصفر[6]»، و همچنین بازی‌های «غیرقطعی» به طور خلاصه بحث می‌کنیم.

از آغاز تمدن بشریت بازی‌ها با قوه‌ی هوش بشر پیوند خورده‌اند. بازی‌ها به‌ذات مشکل‌اند و به همین دلیل یکی از جالب‌ترین موضوعات برای تحقیقات در هوش مصنوعی محسوب می‌شوند. در بازی‌ها نمایش «حالت‌ها» بسیار ساده است وعامل‌ها کنش‌های محدودی برای انتخاب دارند که نتایج حاصل از هر یک از این کنش‌ها برطبق قوانین مشخصی تعیین شده‌اند. بازی‌های فیزیکی مانند «گوی و میدان» و «هاکی روی یخ» تعاریف بسیار پیچیده‌تر، تعداد کنش‌های بیشتر، وحتی قوانین نادقیق‌تری دارند. به جز ربات فوتبالیست تاکنون بازی‌های فیزیکی دیگر چندان مورد توجه جامعه هوش‌مصنوعی قرار نگرفته‌اند.

---

[1]- Multi-agent
[2]- Cooperative
[3]- Competitive
[4]- Adversarial search
[5]- Utility function
[6]- Non Zero -sum

بازی‌ها یکی از اولین موضوعاتی بودند که مورد توجه هوش مصنوعی قرار گرفتند. در سال ۱۹۵۰ (تقریباً زمانی که تازه کامپیوترها قابل برنامه‌نویسی شده‌بودند) بازی شطرنج توسط Konrad Zuse (مخترع اولین کامپیوتر قابل برنامه‌نویسی و اولین زبان برنامه‌نویسی)، Claude Shannon (مخترع نظریه اطلاعات)، Norbert Wiener (سازنده‌ی تئوری کنترل مدرن) و Alan Turing بررسی شد. از آن زمان به بعد جریانی در جهت پیشرفت بازی شطرنج اتفاق افتاد تا آنجا که در بازی‌هایی مثل شطرنج و تخته‌نرد ماشین از انسان پیشی گرفت و قهرمانان این رشته‌ها را (نه همیشه ولی در مواردی) شکست داد این در حالی است که در بسیاری از بازی‌های دیگر نیز انسان و ماشین رقابت سختی با یکدیگر دارند.

بازی‌ها (برخلاف مسائل ساده‌ی فصل ۳) بدلیل ماهیت مشکلی که دارند بسیار جذاب‌اند. برای مثال، شطرنج به‌طور میانگین دارای فاکتور انشعاب ۳۵ است و معمولاً در هر بازی ۵۰ حرکت توسط هریک از بازیکنان انجام می‌شود؛ بنابراین درخت جستجوی آن تقریباً 35^{100} و یا 10^{154} گره دارد (اگرچه گراف جستجوی آن تنها 10^{40} گره متفاوت دارد). بازی‌ها (همچون جهان واقعیت) نیاز به قدرت تصمیم‌گیری دارند، حتی در شرایطی که پیدا کردن انتخاب بهینه غیرممکن باشد. گرفتن تصمیمی غیربهینه در بازی به‌شدت ما را دچار خسران می‌کند. لذا هزینه‌ی پیاده سازی جستجوی *Aای که نیمه- بهینه است دو برابر هزینه‌ی اجرای کامل این الگوریتم است. بنابراین، برنامه شطرنجی که به صورت نیمه- بهینه است (در صورت ثابت بودن بقیه‌ی شرایط) در زمان مشخص شده‌ی خود به شدت ضعیف عمل می‌کند. بنابراین تحقیقات روی بازی‌ها ایده‌های بسیار زیادی را در رابطه با چگونگی بهترین استفاده از زمان مطرح کرده‌است.

در این فصل ابتدا تعریفی از حرکت بهینه ارائه والگوریتمی برای یافتن آن طرح می‌کنیم. سپس تکنیک‌های انتخاب یک حرکت مناسب در«زمان محدود» را بررسی می‌کنیم. «هرس کردن» تکنیکی است که به ما اجازه می‌دهد از قسمت‌هایی از درخت جستجو که اثری بر نتیجه‌ی نهایی ما ندارند صرف نظر کنیم و توابع ابتکاری ارزیابی به ما کمک می‌کند که تقریب درستی از سودمندی یک حالت بدون جستجوی کامل درخت داشته باشیم. بخش ۶-۵ درباره‌ی بازی‌هایی همچون تخته نرد که عنصر شانس در آن‌ها دخیل است بحث می‌کند. سرانجام نگاهی به تقابل میان برنامه‌های بازی و انسان و پیشرفت آن در آینده می‌اندازیم.

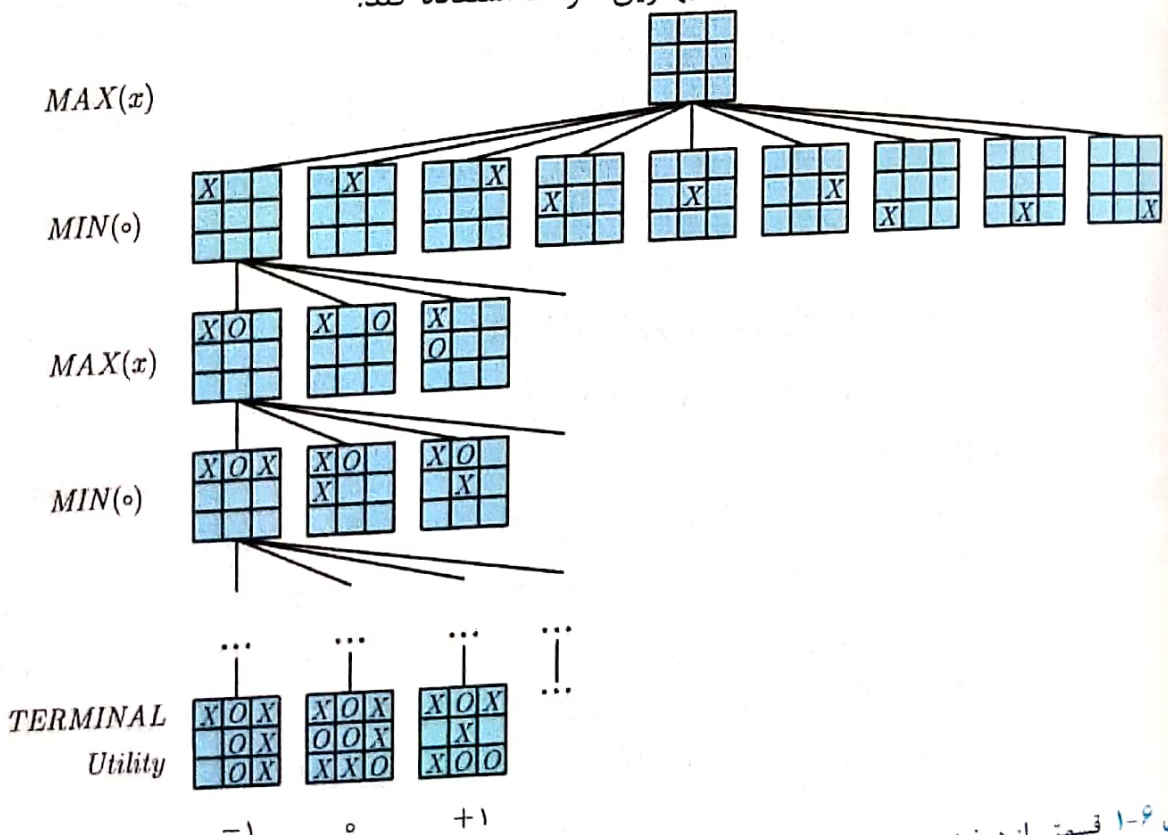
۶-۲- تصمیمات بهینه در بازی‌ها

ما در این بخش در مورد بازی‌هایی بحث می‌کنیم که دو بازیکن به نام‌های MAX و MIN (در آینده به دلایل این نام‌گذاری اشاره خواهیم نمود) در آن‌ها شرکت می‌کنند. بازی با حرکت MAX شروع شده و سپس دو بازیکن تا پایان بازی به صورت یکی در میان بازی میکنند. در پایان بازی امتیازات به فرد برنده داده شده و بازنده جریمه می‌شود. بازی را می‌توان به صورت یک مسئله‌ی جستجو تعریف کرد و با عناصر زیر نمایش داد:

- حالت اولیه: وضعیت اولیه‌ی صفحه بازی و مشخص کردن نوبت بازیکنان
- تابع پسین: لیستی از جفت‌های (حرکت، حالت) را برمی‌گرداند که هریک از آن‌ها نشان‌دهنده‌ی یک حرکت مجاز و «حالت» حاصله از آن است.
- تست پایانی: تعیین می‌کند که چه زمانی بازی پایان یافته‌است. حالتی که در آن بازی تمام می‌شود «حالت پایانی» نام دارد.

- تابع سودمندی (تابع هدف و یا تابع امتیاز نیز نامیده می‌شود): عددی را برای حالت پایانی در نظر می‌گیرد. در شطرنج نتیجه‌ی بازی یکی از سه حالت برد، باخت و یا تساوی با مقادیر ۱+، ۱- و یا ۰ است. در بعضی از بازی‌ها نتایج در گستره‌ی وسیع‌تری قرار می‌گیرند. امتیازات در تخته‌نرد در گستره‌ی ۱۹۲+ تا ۱۹۲- می‌باشد. این فصل اکثراً به بازی‌های مجموع- صفر می‌پردازیم اگرچه به مسائل مجموع- غیرصفر نیز اشارات کوچکی می‌کنیم.

«درخت بازی» از حالت اولیه و حرکات مجاز هر کدام از بازیکنان تشکیل می‌شود. شکل ۶-۱ قسمتی از درخت بازی، برای بازی تیک- تاک- تو (X-O یا ضربدر و پوچ) را نشان می‌دهد. در حالت اولیه MAX دارای ۹ حرکت مجاز است. بازی به صورت یکی در میان بین MAX (که X را در صفحه قرار می‌دهد) و MIN (که O را در صفحه قرار می‌دهد) ادامه می‌یابد تا زمانی که به برگ‌هایی برسیم که در آن‌ها حالت پایانی اتفاق افتاده است. در این مثال، حالت پایانی زمانی اتفاق می‌افتد که یک بازیکن سه مهره در یک ردیف داشته باشد و یا اینکه تمامی خانه‌ها پر شده باشند. عددی که بر روی هر یک از برگ‌ها درج شده‌است، نشان‌دهنده‌ی مقدار سودمندی برای هر یک از حالت‌های پایانی از دید بازیکن MAX است. بنابراین مقادیر بیش‌تر به‌نفع MAX و به ضرر MIN است (نام‌گذاری آن‌ها نیز بر همین اساس بوده‌است). این وظیفه‌ی MAX است که از درخت جستجو (مخصوصاً مقدار سودمندی حالت‌های پایانی) برای انتخاب بهترین حرکت استفاده کند.



شکل ۶-۱ قسمتی از درخت جستجو برای بازی تیک-تاک-تو. بالاترین گره، حالت اولیه است. بازیکن MAX اولین حرکت را انجام می‌دهد و یک X در یکی از خانه‌های خالی می‌گذارد. ما در این شکل قسمتی از درخت جستجو را نمایش داده‌ایم که در آن حرکت‌های MIN و MAX به صورت یکی در میان انجام می‌شوند تا اینکه سرانجام به حالت‌های پایانی می‌رسند (که در آن مقادیر سودمندی براساس قوانین بازی داده می‌شود).

۶-۲-۱- استراتژی‌های بهینه

در مسائل جستجو، پاسخ بهینه به‌صورت دنباله‌ای از حرکت‌هاست که در انتها به هدفی (یک حالت پایانی که نشان‌دهنده‌ی برد است) منجر می‌شوند. از طرفی MIN نیز در بازی نقش ایفا می‌کند. بنابراین MAX باید یک

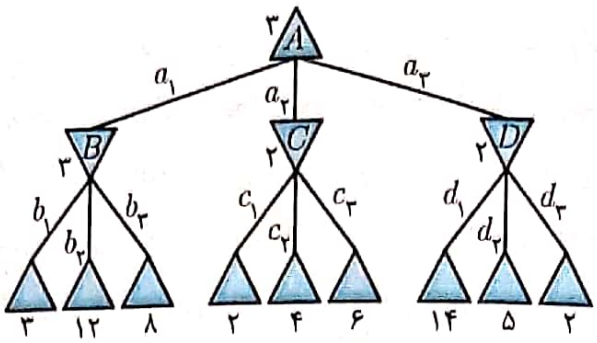
«استراتژی احتمالی» را برگزیند که براساس آن ابتدا (منظور در حالت اولیه) حرکت MAX تعیین می‌شود. سپس حرکت بعدی MAX از حالتی شروع می‌شود که از پاسخ MIN به حرکت MAX به وجود آمده‌است و همین‌طور این روند ادامه پیدا می‌کند. به‌عبارت دیگر، استفاده از استراتژی بهینه در بازی منجر به نتیجه‌ای بهتر و یا حداقل مساوی استراتژی‌های دیگر می‌شود (البته با این فرض که حریف به صورت ایده‌آل و بدون اشتباه بازی کند.) حال می‌خواهیم چگونگی یافتن این استراتژی‌های بهینه را به شما نشان دهیم (حتی در بازی‌هایی بسیار پیچیده‌تر از تیک- تاک- تو که محاسبات آن برای MAX غیرممکن است).

ازآنجا که ترسیم کل درخت بازی حتی در بازی‌های ساده‌ای همچون تیک-تاک-تو بسیار مشکل است، بازی کوچک نشان داده شده در شکل ۶-۲ را درنظر می‌گیریم. حرکت‌های مجاز برای MAX در ریشه با a_1، a_2 و a_3 برچسب گذاری شده‌اند. پاسخ‌های مجاز برای MIN (برای پاسخ به a_1) نیز b_1، b_2 و b_3 هستند (و همین طور تا آخر). این بازی بعد از انتخاب یک حرکت توسط هر یک از بازیکنان پایان می‌یابد (در اصطلاح، می‌گوییم این درخت به اندازه‌ی یک حرکت عمق دارد و شامل دو نیم- حرکت است که هریک از آن‌ها یک ply نام دارند). مقدار سودمندی حالت‌های پایانی در بازه ۲ تا ۱۴ قرار می‌گیرند.

با فرض داشتن درخت بازی، بهینه‌ترین استراتژی برای بازی، انتخاب مقدار minimax گره‌ها است که به صورت MINIMAX(n) نوشته می‌شود. مقدار minimax یک گره، متناظر با مقدار سودمندی حالت مورد نظر برای MAX است البته با این فرض که هر دو بازیکن تا انتهای بازی به صورت بهینه بازی کنند. بدیهی است که مقدار minimax حالت‌های پایانی همان مقدار سودمندی است. به‌علاوه از بین گزینه‌های موجود همواره MAX به دنبال حرکتی با بیش‌ترین مقدار و MIN به دنبال حرکتی با کم‌ترین مقدار می‌گردد. بنابراین می‌توان نوشت:

$$\text{MINIMAX-VALUE}(n)=\begin{cases}\text{UTILITY}(n) & \text{اگر گره n یک حالت پایانی باشد}\\ \max_{s\in\text{Successor}(n)}\text{MINIMAX-VALUE}(s) & \text{اگر گره n MAX باشد}\\ \min_{s\in\text{Successor}(n)}\text{MINIMAX-VALUE}(s) & \text{اگر گره n MIN باشد}\end{cases}$$

حال اجازه دهید این تعاریف را بر روی درخت بازی شکل ۶-۲ اعمال کنیم. گره‌های پایانی، با مقادیر سودمندی برچسب‌گذاری شده‌اند. اولین گره MIN (که با B برچسب‌گذاری شده) دارای گره‌های پسین با مقادیر ۳، ۱۲ و ۸ است بنابراین مقدار minimax آن برابر ۳ خواهد بود. به‌همین ترتیب دو گره MIN دیگر دارای مقدار minimax ۲ هستند. ریشه، گره‌ای از نوع MAX است که گره‌های پسین آن دارای مقادیر minimax، ۳، ۲ و ۲هستند. پس ریشه دارای مقدار minimax ۳ است. براساس مقادیر minimax موجود در ریشه بدین شکل تصمیم‌گیری می‌کنیم: کنش a_1 بهترین انتخاب برای MAX است زیرا منجر به گره پسینی با بیش‌ترین مقدار minimax می‌شود. در این نوع تصمیم‌گیری برای MAX فرض بر این است که گره MIN نیز همواره به صورت بهینه بازی می‌کند. اما اگر MIN به صورت بهینه بازی نکرد چطور؟ آنگاه به‌راحتی می‌توان نشان داد که MAX نتیجه‌ی بهتری خواهد گرفت (تمرین شماره‌ی ۶-۲). اگرچه ممکن است استراتژی‌های دیگری در مقابل حریف‌های غیربهینه وجود داشته‌باشند که بهتر از استراتژی minimax عمل کنند، اما همه‌ی این استراتژی‌ها قطعاً در مقابل حریف‌های بهینه بد عمل می‌کنند.



شکل ۶-۲ یک درخت بازی ply-۲. گره‌های ▲ گره‌های MAX هستند، که در آن‌ها نوبت MAX در بازی است، و گره‌های ▼ گره‌های MIN هستند. گره‌های پایانی نشان‌دهنده مقادیر سودمندی برای MAX هستند و گره‌های دیگر با مقادیر minimax خود برچسب‌گذاری شده‌اند. بهترین حرکت MAX در ریشه، a۱ است زیرا منجر به گره پسینی با بیش‌ترین مقدار minimax می‌شود و بهترین جواب MIN، b۱ است، زیرا منجر به گره پسینی با کمترین مقدار minimax می‌شود.

۶-۲-۲- الگوریتم minimax

الگوریتم minimax (شکل ۶-۳) از حالت جاری شروع به تصمیم‌گیری می‌کند. این الگوریتم به‌راحتی با انجام محاسبات برروی مقادیر minimax حالت‌های پسین، به صورت بازگشتی، تساوی‌های فوق‌الذکر را پیاده‌سازی می‌کند. این راه‌حل بازگشتی کل برگ‌های درخت را تا پایین رفته و مقادیر minimax را به سمت بالا برمی‌گرداند. برای مثال در شکل ۶-۲ الگوریتم ابتدا به سمت سه گره پایینی سمت چپ رفته و از تابع سودمندی برای کشف مقادیر آن‌ها استفاده می‌کند (به ترتیب ۳، ۱۲، ۸). سپس الگوریتم کمترین مقدار آن‌ها را (که همان ۳ است) انتخاب و آن را به سمت گره B بازگشت می‌دهد. فرآیند مشابهی مقادیر ۲ و ۲ را به گره‌های C و D برمی‌گردانند. در آخر بیش‌ترین مقدار را از بین سه مقدار ۳، ۲ و ۲ انتخاب و به سمت گره ریشه می‌فرستد.

```
function Minimax-Decision(state) return an action
    inputs: state, current state in game

    v ← Max-Value(state)
    return the action in Successors(state) with value v
```

```
function Max-Value(state) return a utility value
    if Terminal-Test(state) then return Utility(state)
    v← -∞
    for a, s in Successors(state) do
        v ← Max(v,Min-Value(s))
    return v
```

```
function Min-Value(state) returns a utility value
    if Terminal-Test(state) then return Utility(state)
    v← ∞
    for a, s in Successors(state) do
        v← Min(v, Max-Value(s))
    return v
```

شکل ۶-۳ الگوریتمی برای محاسبه تصمیم‌گیری minimax. این الگوریتم کنشی را که متناظر با بهترین حرکت مجاز است برمی‌گرداند (یعنی حرکتی که منجر به نتیجه‌ای با بهترین مقدار سودمندی می‌شود) البته با این فرض که حریف همواره تلاش می‌کند که مقدار سودمندی را مینیمم کند. توابع MIN-VALUE و MAX-VALUE کل درخت را تا برگ‌ها پیمایش و مقادیر حالت‌های بازگشتی را تعیین می‌کنند.

الگوریتم minimax یک جستجوی کامل عمق-اول را به‌روی درخت بازی پیاده‌سازی می‌کند. اگر بیش‌ترین عمق درخت برابر m، و در هر مرحله b حرکت مجاز موجود باشد، آنگاه پیچیدگی زمانی الگوریتم minimax برابر $O(b^m)$ خواهد بود. پیچیدگی فضای حافظه آن برای الگوریتمی که تمامی گره‌های پسین را در هر مرحله تولید می‌کند برابر O(bm) است و برای الگوریتمی که در هر مرحله تنها یک گره پسین تولید می‌کند برابر O(m) می‌شود. در بازی‌های واقعی، پرداخت این هزینه‌ی زمانی غیرعملی به‌نظر می‌رسد اما به عنوان پایه‌ای برای تحلیل بازی‌ها و الگوریتم‌های کاربردی استفاده می‌شوند.

۶-۲-۳- تصمیم‌گیری بهینه در بازی‌های چند نفره

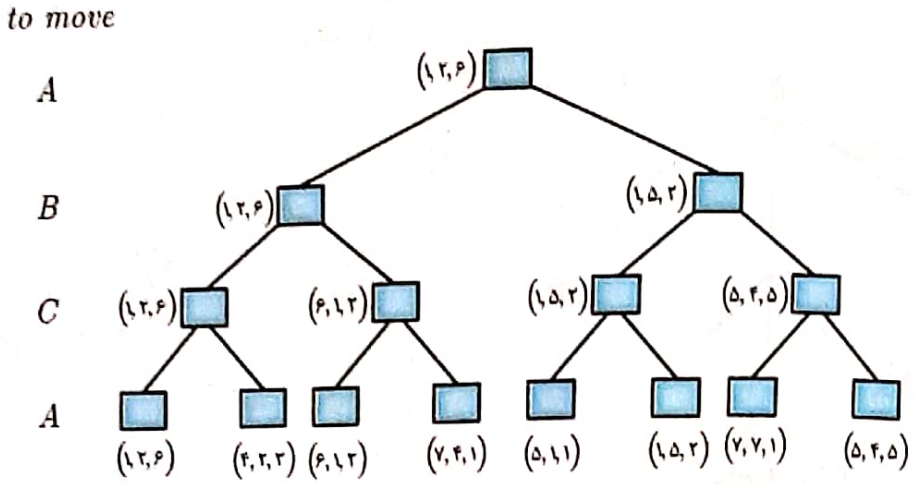
در بسیاری از بازی‌ها بیش از دو نفر حضور دارند. بگذارید ایده‌ی minimax را بر روی بازی‌های چند نفره امتحان کنیم. اگرچه این موضوع از دیدگاه فنی بسیار بدیهی به نظر می‌رسد اما پاره‌ای از مطالب جذاب در آن نهفته است.

ابتدا باید به جای یک مقدار واحد برای هریک از گره‌ها، برداری از مقادیر را در نظر بگیریم. به‌عنوان مثال، در یک بازی سه نفره با بازیکنان A، B و C برداری همچون $<V_a, V_b, V_c>$ دارای مقداری مرتبط با هریک از گره‌هاست. در حالات پایانی، این بردار مقدار سودمندی حالت مورد نظر را از نقطه‌نظر هر یک از بازیکنان به‌طور جداگانه نمایش می‌دهد (در بازی‌های دونفره (مجموع- صفر) دو مؤلفه‌ی بردار به یک مقدار کاهش می‌یابد زیرا این دو مؤلفه متضاد یکدیگرند). ساده‌ترین راه برای پیاده‌سازی این موضوع استفاده از تابع UTILITY است که در آن برداری از مقادیر سودمندی را به عنوان خروجی برمی‌گرداند.

حال گره‌هایی غیر از گره‌های پایانی را درنظر بگیرید. به گره مشخص شده X در شکل ۶-۴ دقت کنید. در این حالت، بازیکن C چه حرکتی را انتخاب می‌کند؟ این دو انتخاب ما را به دو حالت پایانی متفاوت با بردار سودمندی $<V_a=۱,V_b=۲,V_c=۶>$ و $<V_a=۴,V_b=۲,V_C=۳>$ می‌برند. از آنجا که ۶ بزرگتر از ۳ است C باید حرکت اول را انتخاب کند. این بدین معنی است که اگر حین جستجو به گره X برسیم، بازی منجر به حالت پایانی با مقادیر سودمندی $<V_a=۱,V_b=۲,V_c=۶>$ می‌شود. لذا مقدار بازگشتی X همین بردار خواهد بود. به طور کلی مقدار بازگشتی گره n، یک «بردار سودمندی» است که در آن گره پسین دارای بیش‌ترین مقدار برای آن بازیکن در انتخاب n باشد. افرادی که در بازی‌های چند نفره بازی می‌کنند (هم‌چون دیپلماسی در سیاست) از این موضوع اگاه هستند که اتفاقات بسیار بیش‌تری از یک بازی دو نفره در جریان است. در بازی‌های چند نفره گاهی به صورت رسمی یا غیررسمی میان بازیکنان ائتلاف[1] شکل می‌گیرد. در طول زمان بازی معمولاً ائتلاف‌های زیادی تشکیل و دوباره شکسته می‌شود. درک ما از چنین رفتارهایی چگونه است؟ آیا ائتلاف‌ها جزئی از استراتژی بهینه برای هر یک از بازیکنان در یک بازی چند نفره محسوب می‌شود؟ پاسخ مثبت است. برای مثال فرض کنید A و B ضعیف هستند و C در موقعیتی قوی‌تری نسبت به آنان قرار دارد. بنابراین برای A و B این تصمیم بهینه محسوب می‌شود که به جای حمله به همدیگر ابتدا هر دو به C حمله کرده تا از نابودی خود توسط C جلوگیری کنند. به این ترتیب همکاری میان بازیکنانی که قبلاً به صورت مجرد عمل می‌کردند به وجود می‌آید. البته به محض اینکه C بعد از این یورش هماهنگ ضعیف شد ائتلاف ارزش خود را از دست

[1]- Alliance

می‌دهد و هرکدام از A یا B ممکن است توافق انجام شده را نقض کنند. گاهی اوقات ائتلاف‌هایی که به صورت رسمی انجام می‌شود تحت هر شرایطی پایدار باقی می‌ماند. در بسیاری از موارد نیز شکستن ائتلاف موجب بدنامی می‌شود بنابراین بازیکنان باید بین سود حاصل از شکستن ائتلاف و ضرر بلندمدت حاصل از شناخته شدن به عنوان یک فرد غیرقابل اعتماد، یکی را انتخاب کنند.

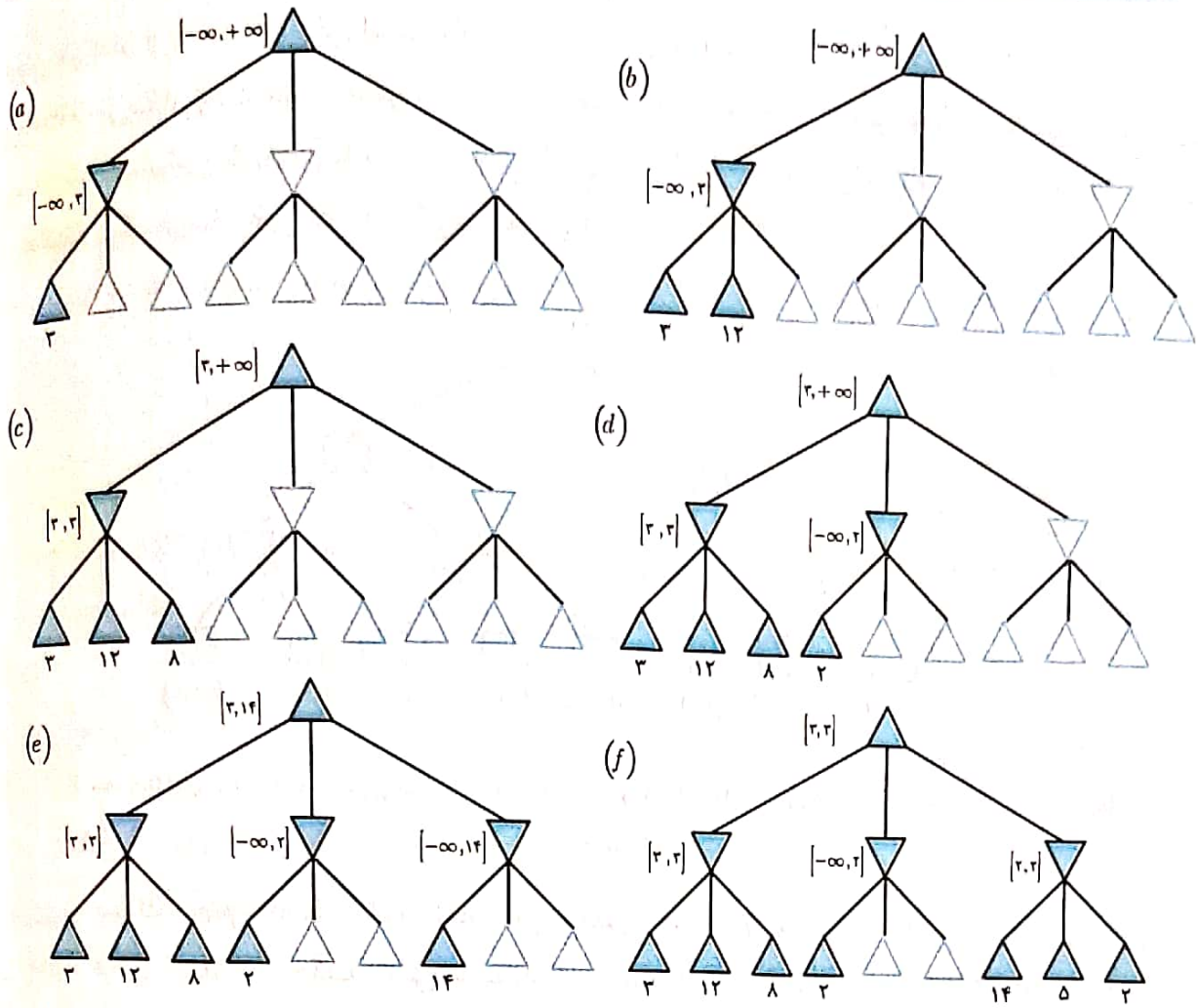


شکل ۶-۴ سه ply اول از یک درخت بازی سه نفره (A، B، C). هر گره از دید هر بازیکن برچسب‌گذاری شده است. بهترین حرکت در ریشه مشخص شده‌است.

اگر بازی به صورت مجموع- صفر نباشد آنگاه حتی ممکن است در بازی‌های دو نفره نیز همکاری اتفاق بیفتد. برای مثال، فرض کنید یک حالت پایانی با مقدار سودمندی $<V_a=1000,V_b=1000>$ وجود دارد و ۱۰۰۰ بالاترین مقدار سودمندی برای هر یک از بازیکنان است. بنابراین استراتژی بهینه برای هریک از بازیکنان این خواهد بود که هر کاری برای رسیدن به این حالت انجام دهند (بدین معنی که دو بازیکن به‌طور خودکار برای رسیدن به هدف مشترک خود همکاری می‌کنند).

۶-۳- هرس آلفا- بتا

مشکل اصلی الگوریتم minimax این است که تعداد حالت‌های بازی، به صورت نمایی برحسب تعداد حرکت‌ها رشد می‌کند. اگرچه نمی‌توانیم این هزینه‌ی نمایی را از بین ببریم اما می‌توانیم آن را به نصف کاهش دهیم. ایده این است که تصمیم minimax را بدون جستجوی تمامی گره‌های درخت بازی محاسبه کنیم. به‌عبارت دیگر می‌توانیم از ایده هرس کردن (که در فصل ۴ به آن اشاره شد) برای حذف قسمت‌های بزرگی از درخت استفاده و از جستجوی این قسمت‌ها صرف نظر کنیم. تکنیک بسیار مشهوری که ما از آن استفاده می‌کنیم هرس آلفا- بتا است. هنگامی که این روش بر روی درخت استاندارد minimax اعمال می‌شود، دقیقاً مثل الگوریتم minimax عمل می‌کند با این تفاوت که شاخه‌هایی را که در تصمیم نهایی ما تأثیرگذار نیستند هرس می‌کند. دوباره درخت بازی شکل ۶-۲ را که به صورت دو-ply است در نظر بگیرید. بگذارید فرآیند محاسبه تصمیم بهینه در این درخت را دوباره بررسی کنیم. این بار به کارهایی که در هر مرحله انجام می‌دهیم دقت کنید. مراحل در شکل ۶-۵ توضیح داده شده‌اند. نتیجه اینکه تصمیم minimax را می‌توانیم بدون ارزیابی دو تا از برگ‌ها بیابیم.

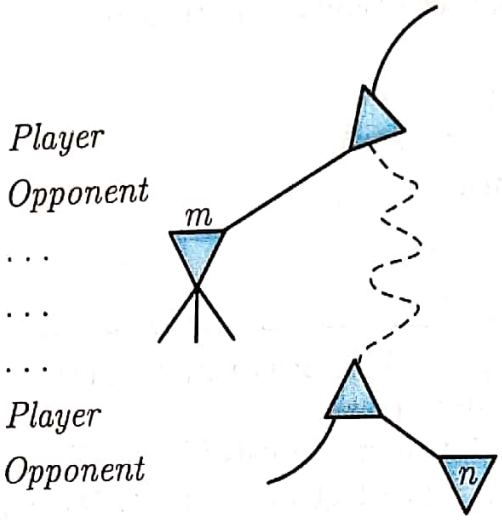


شکل ۶-۵ مراحل محاسبه تصمیم‌گیری بهینه برای درخت بازی شکل ۶-۲. در هر نقطه، برای هر گره گستره‌ی مقادیر مجاز نشان داده شده‌است. (a) اولین برگ پایین B دارای مقدار ۳ است. لذا B (که یک گره از نوع MIN است) دارای حداکثر مقدار ۳ است. (b) دومین برگ پایین B دارای مقدار ۱۲ است. MIN از این حرکت صرف‌نظر می‌کند بنابراین حداکثر مقدار B همچنان ۳ است. (c) سومین برگ پایین B، دارای مقدار ۸ است. ما تا اینجا تمامی گره‌های پسین B را دیده‌ایم، بنابراین مقدار دقیق B همان ۳ است. حال می‌توانیم نتیجه بگیریم که مقدار ریشه حداقل ۳ است زیرا MAX در ریشه دارای انتخابی برابر ۳ است. (d) اولین برگ پایین C دارای مقدار ۲ است. لذا C (که یک گره از نوع MIN است) دارای حداکثر مقدار ۲ است. اما می‌دانیم که B دارای ارزش ۳ است بنابراین MAX هیچ‌گاه C را انتخاب نمی‌کند. بنابراین جستجو در گره‌های پسین C ارزشی ندارد. این دقیقاً مثالی از هرس آلفا- بتا است. (e) اولین گره پایین D، دارای مقدار ۱۴ است. بنابراین D دارای حداکثر مقدار ۱۴ است. این مقدار فعلاً بالاتر از بهترین انتخاب MAX (یعنی ۳) بوده است بنابراین باید گره‌های پسین D را نیز بررسی کنیم. به این نکته توجه داشته باشید که ما تا حالا برای تمامی گره‌های پسین ریشه، کرانی را تعیین کرده‌ایم بنابراین مقدار ریشه حداکثر برابر ۱۴ است. (f) دومین گره پسین D دارای مقدار ۵ است، بنابراین باید به جستجو ادامه دهیم. سومین گره پسین آن دارای مقدار ۲ است، بنابراین D حالا دارای مقدار دقیق ۲ است. تصمیم‌گیری MAX در ریشه، حرکت B را انتخاب می‌کند (همان مقدار ۳).

بگذارید با ساده کردن فرمول MINIMAX-VALUE از زاویه دیگری به مسئله نگاه کنیم. فرض کنید دو گره پسین بررسی نشده‌ی گره C در شکل ۶-۵ دارای مقادیر x و y و مینیمم آن‌ها z باشد. مقدار ریشه بدین شکل به‌دست می‌آید:

$$\begin{aligned}\text{MINIMAX-VALUE (root)} &= \max(\min(۳, ۱۲, ۸), \min(۲, x, y), \min(۱۴, ۵, ۲)) \\ &= \max(۳, \min(۲, x, y), ۲) \\ &= \max(۳, z, ۲) \text{ where } z \leq ۲ \\ &= ۳.\end{aligned}$$

به عبارت دیگر، مقدار ریشه و یا همان تصمیم minimax مستقل از مقدار دو برگ هرس شده x و y است. هرس آلفا- بتا را می‌توان بر روی درختی با هر عمق دلخواه اعمال کرد این در حالی است که گاهی ممکن است به جای یک برگ از درخت، کل زیردرخت آن هرس شود. **ایده‌ی اصلی به صورت زیر است:** گره‌ای مانند n را در شاخه‌ای از درخت در نظر بگیرید (شکل ۶-۶). بازیکن می‌تواند به چنین گره‌ای برود، حال اگر او گزینه‌ی بهتری مانند m در گره والد و یا در نقطه‌ای بالاتر از آن داشته باشد، آنگاه بازیکن هیچگاه در بازی خود به n نمی‌رود. بنابراین اگر برای رسیدن به این نتیجه به اندازه‌ی کافی اطلاعات درباره گره n داشته باشیم (با بررسی نوادگان آن) آنگاه آن را هرس می‌کنیم.



شکل ۶-۶ هرس آلفا- بتا: حالت کلی. اگر برای بازیکن m بهتر از n باشد، آنگاه هیچ‌وقت در بازی به سراغ n نمی‌رویم.

به یاد داشته باشید که جستجوی minimax یک نوع جستجوی عمق- اول است، بنابراین در هر لحظه، تنها گره‌های مربوط به یک مسیر از درخت را بررسی می‌کنیم. نام‌گذاری هرس آلفا- بتا از دو پارامتر زیر که کرانی برای مقادیر پشتیبان[1] (که در طول مسیر همواره وجود دارند) تعیین می‌کنند گرفته شده‌است:

α = مقدار بهترین گزینه‌ای (بیش‌ترین مقدار) است که تا به حال در طول مسیر برای بازیکن MAX پیدا کردیم.

β = مقدر بهترین گزینه‌ای (کم‌ترین مقدار) است که تا به حال در طول مسیر برای بازیکن MIN پیدا کردیم.

جستجوی آلفا-بتا مقادیر α و β را در طول مسیر به‌روزرسانی کرده و به محض اینکه مشخص شد مقدار گره فعلی از مقادیر α و β برای MAX وMIN بدتر است شاخه‌های باقی‌مانده از گره مورد نظر را هرس می‌کند (در واقع همان پایان فراخوانی بازگشتی است). شکل کامل این الگوریتم در شکل ۶-۷ نمایش داده شده است. پیشنهاد می‌کنیم خوانندگان این الگوریتم را بر روی درخت شکل ۶-۵ اعمال کرده و رفتار آن را بررسی کنند.

[1]- back-up

```
function Alpha-Beta-Search(state) returns an action
  inputs: state, current state in game

  v ← Max-Value(state, -∞,+∞)
  return the action in Successors(state) with value v
```

```
function MAX-VALUE(state, α,β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if Terminal-Test (state) then return Utility(state)
  v ← -∞
  for a, s in Successors(state) do
     v ←Max(v,Min-Value(s,α,β))
     if v ≥ β then return v
     α ← MAX (α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if Terminal-Test(state) then return Utility(state)
  v ←+∞
  for a, s in Successors(state) do
     v ←Min(v,Max-Value(s,α,β))
     if v ≤ α then return v
     β ← Min(β, v)
  return v
```

شکل ۶-۷ الگوریتم جستجوی آلفا- بتا. دقت داشته باشید که این قطعه کد دقیقاً شبیه به قطعه کد MINIMAX شکل۶-۳ است به‌جز در دو خطی که در آن MIN-VALUE و MAX-VALUE مقادیر α و β را نگهداری می‌کنند.

تأثیرگذاری الگوریتم آلفا- بتا به‌شدت به ترتیب امتحان کردن گره‌های پسین بستگی دارد. برای مثال در شکل ۶-۵ در قسمت (e) و (f) نمی‌توانیم هیچ یک از گره‌های پسین D را هرس کنیم، زیرا بدترین گره‌های پسین (از زاویه دید MIN) خیلی زود تولید شدند. اگر ابتدا سومین گره پسین تولید می‌شد، آنگاه قادر بودیم که دو گره دیگر را هرس کنیم. این موضوع این نکته را گوشزد می‌کند که شاید به‌صرفه باشد که تلاش کنیم ابتدا گره‌های پسینی را تولید کنیم که مفیدتر به‌نظر می‌رسند.

اگر فرض کنیم این موضوع قابل پیاده‌سازی است، آنگاه می‌توان نتیجه گرفت، الگوریتم آلفا-بتا به جای $O(b^m)$ گره، تنها $O(b^{m/2})$ گره را برای انتخاب بهترین حرکت minimax تولید می‌کند. یعنی فاکتور انشعاب مؤثر به جای b، $\sqrt{b}$ می‌شود (برای بازی شطرنج به جای ۳۵، ۶ خواهد بود). از سوی دیگر در یک زمان مشخص، الگوریتم آلفا-بتا، دو برابر فراتر از الگوریتم minimax را پیش‌بینی خواهد کرد. اگر گره‌های پسین به جای

انتخاب بهترین گره به صورت تصادفی تولید و امتحان شوند، آنگاه تعداد گره‌های تولید شده (برای b متوسط) تقریباً برابر $O(b^{3m/4})$ خواهد بود. برای شطرنج یک تابع اولویت مناسب (مثلاً ابتدا گرفتن اسیر، بعد تهدید کردن، بعد حرکت روبه‌جلو و بعد از آن حرکت روبه‌عقب) تعداد گره‌هایی معادل دو برابر مقدار $O(b^{m/2})$ تولید می‌کند. اضافه کردن الگوی اولویت–حرکت‌ها به صورت پویا (مثل انتخاب حرکتی که در آخرین بار بهترین نتیجه را از آن گرفتیم) ما را به کران تئوری فوق‌الذکر بسیار نزدیک می‌کند.

در فصل ۳، اشاره کردیم که حالت‌های تکراری در جستجوی درخت می‌تواند هزینه‌ی درخت را به صورت نمایی افزایش دهد. در بازی‌ها به‌دلیل وجود خاصیت جابجایی، حالت‌های تکراری به‌طور مداوم تکرار می‌شوند (ترکیب-های متفاوتی از دنباله‌ای از حرکت‌ها که منجر به نتیجه یکسان می‌شوند). برای مثال، اگر سفید یک حرکت a_1 داشته‌باشد و سیاه بتواند آن را با b_1 جواب دهد و در نقطه‌ی دیگری از صفحه یک حرکت بی‌ارتباط با a_1 بنام a_2 وجود داشته باشد که با b_2 جواب داده می‌شود، آنگاه دو دنباله $[a_1,b_1,a_2,b_2]$ و $[a_1,b_2,a_2,b_1]$ هر دو به موقعیت یکسانی منجر می‌شوند (همین‌طور در مورد ترکیب‌های دیگری که با a_2 شروع می‌شوند). بسیار ارزشمند است که مقدار ارزیابی هر یک از این موقعیت‌ها را بعد از یک‌بار محاسبه در یک جدول درهم‌سازی[1] ذخیره کنیم تا در مواقع نیاز مجدداً آن‌ها را محاسبه نکنیم. جدول درهم‌سازی موقعیت اشاره شده به جدول انتقال[2] معروف است. استفاده از یک جدول انتقال می‌تواند تأثیر شگفت‌آوری داشته‌باشد به گونه‌ای که گاهی اوقات در شطرنج عمق جستجوی قابل دسترس را دو برابر می‌کند. از طرف دیگر، اگر ما یک میلیون گره را در یک ثانیه پردازش کنیم، آنگاه نمی‌توان تمامی آن‌ها را در جدول انتقال نگهداری کرد. به همین دلیل استراتژی‌های متفاوتی برای انتخاب باارزش‌ترین آن‌ها امتحان شده‌است.

۶-۴- تصمیمات ناکامل و بی‌درنگ[3]

الگوریتم minimax کل فضای حالت را جستجو می‌کند، درحالی‌که الگوریتم آلفا-بتا به ما اجازه می‌دهد که قسمت‌های بزرگی از درخت را هرس کنیم. اما الگوریتم آلفا-بتا نیز برای رسیدن به حالت پایانی باید قسمتی از فضای جستجو را بررسی کند. جستجوی این عمق از درخت غیرعملی است، زیرا حرکت‌ها باید در زمان معقولی انجام شوند (حداکثر در چند دقیقه). شنون در مقاله خود در سال ۱۹۵۰ (برنامه‌نویسی کامپیوتر برای بازی شطرنج) پیشنهاد کرد که برنامه‌ها جستجوی خود را کمی زودتر قطع کنند و یک تابع ارزیابی ابتکاری را بر روی حالت‌های جستجو شده اعمال کنند و به این شکل گره‌های غیرپایانی را به برگ‌های پایانی تبدیل کنند. به عبارت دیگر ایده این است که الگوریتم minimax یا آلفا-بتا را به شکل مقابل تغییر دهیم: تابع سودمندی را به تابع ارزیابی ابتکاری (EVAL) تبدیل کنیم (که تقریبی از سودمندی موقعیت‌ها را به ما می‌دهد) و تست پایانی را به تست قطع[4] تبدیل کنیم که در مورد زمان اعمال EVAL تصمیم‌گیری می‌کند.

۱- Hash table
۲- Transposition
۳- Real-time
۴- Cutoff test

۶-۴-۱- توابع ارزیابی

تابع ارزیابی، تخمینی از سودمندی یک موقعیت مشخص در بازی به ما می‌دهد (همان‌طور که توابع ابتکاری در فصل ۴ تخمینی از فاصله تا هدف به ما می‌دادند). وقتی «شنون» ایده‌ی تابع ارزیابی را پیشنهاد کرد، این اولین باری نبود که این ایده مطرح می‌شد. قرن‌ها است که بازیکنان شطرنج (و هواداران دیگر تیم‌ها) راه‌های بسیاری را برای قضاوت درباره‌ی ارزش یک موقعیت ابداع کرده‌اند، (زیرا انسان توانایی انجام جستجوی به مراتب محدودتری را از برنامه‌های کامپیوتری دارد). بدیهی است که کارایی یک برنامه کاملاً وابسته به کیفیت تابع ارزیابی آن است. یک تابع ارزیابی نادقیق، عامل را به سمت موقعیت شکست سوق می‌دهد. حال این سؤال مطرح می‌شود که چگونه باید توابع ارزیابی مناسبی را ابداع کنیم؟

اولاً تابع ارزیابی باید ترتیب حالت‌های پایانی را به همان شکلی که تابع سودمندی واقعی مرتب می‌کرد قرار دهد درغیراین‌صورت عاملی که از آن استفاده می‌کند ممکن است (حتی اگر کل مسیر تا پایان بازی را نیز بداند) حرکت‌های غیربهینه را انتخاب کند. ثانیاً زمان محاسبات نباید خیلی طولانی شود (تابع ارزیابی می‌تواند تابع MINIMAX-DECISION را درون برنامه‌ی خود صدا زده و هزینه‌ی دقیق موقعیت مورد نظر را محاسبه کند، اما این کار هدف اصلی تابع ارزیابی که همان صرفه‌جویی در زمان است را زیر سؤال می‌برد). ثالثاً برای حالت‌های غیرپایانی تابع ارزیابی باید قویاً به «شانس برد» از آن حالت مرتبط باشد.

شاید خواننده از عبارت «شانس برد» تعجب کند زیرا در بازی مثل شطرنج عنصر شانس وجود ندارد (زیرا ما حالت فعلی را با قطعیت می‌شناسیم و در آن تاس نیز نمی‌اندازیم). اما خواننده باید به این نکته توجه کند، هنگامی که جستجو در حالت‌های غیرپایانی قطع می‌شود آنگاه الگوریتم درباره‌ی نتیجه‌ی نهایی آن حالت‌ها به عدم قطعیت می‌رسد. این نوع از عدم قطعیت نتیجه‌ی محدودیت‌های محاسباتی است نه محدودیت‌های اطلاعاتی. با فرض اینکه یک محدودیت محاسباتی برای تابع ارزیابی (به ازای یک حالت داده شده) داشته باشیم، بهترین کاری که تابع ارزیابی می‌تواند انجام دهد این است که نتیجه‌ی نهایی را حدس بزند.

بگذارید این ایده را کامل‌تر کنیم. بیش‌تر توابع ارزیابی بر اساس مشخصه[۱]های متفاوتی از یک حالت عمل می‌کنند، مثلاً در بازی شطرنج، تعداد سربازهایی که هر یک از بازیکنان از طرف مقابل خود گرفته‌اند یک مشخصه محسوب می‌شود. همه‌ی این مشخصه‌ها (در کنار یکدیگر) حالت‌های شطرنج را به گروه‌های مختلفی تقسیم می‌کند که به آن‌ها کلاس‌های هم‌ارزی گفته می‌شود (زیرا حالت‌های درون هر گروه دارای مشخصه‌های یکسانی هستند). هرکدام از گروه‌های داده شده، دارای حالت‌هایی هستند که منجر به برد، باخت، و یا مساوی می‌شوند. تابع ارزیابی نمی‌تواند این حالت‌ها را از هم تشخیص دهد، اما یک مقدار که نشان‌دهنده‌ی تناسب هریک از حالت‌ها با نتیجه‌ی مرتبط با آن است را به ما برمی‌گرداند. برای مثال، فرض کنید آزمایشات نشان می‌دهد که ۷۲٪ از حالت‌های یک گروه منجر به پیروزی (با مقدار سودمندی: ۱+)، ۲۰٪ منجر به شکست (با مقدار سودمندی: ۱-)، و ۸٪ منجر به تساوی (با مقدار سودمندی: ۰) شده‌اند. منطقی‌ترین نوع ارزیابی برای حالت‌های درون این گروه همان میانگین وزن‌دار یا مقدار مورد انتظار است:

$$(۰/۷۲\times(+۱))+(۰/۲۰\times(-۱))+(۰/۰۸\times۰)=۰/۵۲$$

۱- attribute

در اصل مقدار مورد انتظار برای هر یک از گروه‌ها محاسبه، و در تابع ارزیابی استفاده می‌شود. درست مانند حالت‌های پایانی، تا زمانی که ترتیب حالت‌ها یکسان است، نیازی نیست تابع ارزیابی مقادیر واقعی مورد انتظار را برگرداند.

این نوع تحلیل، برای تخمین احتمال برد کلیه‌ی حالت‌های ممکن، نیاز به تعداد زیادی گروه و در نتیجه مقدار زیادی آزمایش دارد. در عوض توابع ارزیابی مقادیر عددی جداگانه‌ای را از هر یک از مشخصه‌ها محاسبه و آن‌ها را ترکیب کرده تا مقدار کلی را پیدا کنند. برای مثال کتاب‌های مقدماتی شطرنج، به‌طور تقریبی یک «مقدار مادی[1]» به هر مهره نسبت می‌دهند: سرباز دارای ارزش ۱، اسب یا شطرنج ۳، رخ ۵، وزیر ۹. مشخصه‌های دیگری مثل «ساختار مناسب سربازها» و «امنیت پادشاه» ممکن است ارزشی نصف یک سرباز را داشته باشند. سپس مقادیر هر یک از مشخصه‌ها به‌راحتی با یکدیگر جمع شده و مقدار ارزیابی موقعیت مورد نظر به‌دست می‌آید. یک موقعیت با امنیتی معادل یک سرباز نشان‌دهنده‌ی نزدیک شدن ما به پیروزی و موقعیتی با امنیتی معادل سه سرباز، نشان‌دهنده‌ی پیروزی قطعی ماست. در ریاضی به این نوع از توابع ارزیابی، توابع خطی وزن‌دار گویند، زیرا می‌توان آن را اینگونه بیان کرد:

$$\mathrm{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

که در آن wها وزن‌ها و f_iها مشخصه‌های آن موقعیت هستند.. برای شطرنج f_i می‌تواند تعداد هر یک از مهره‌ها بر روی صفحه شطرنج و w می‌تواند مقادیر هر یک از مهره‌ها باشد (۱ برای سرباز، ۳ برای فیل).

اگرچه ایده‌ی حاصل‌جمع مقادیر مشخصه‌ها منطقی به نظر می‌رسد، اما این ایده در صورتی درست است که فرض کنیم: مقادیر هر یک از مشخصه‌ها مستقل از مقادیر مشخصه‌های دیگر است. برای مثال در مقداردهی ۳ به فیل، این واقعیت که فیل‌ها در انتهای بازی بسیار قدرتمندترند (زیرا فضای بیش‌تری برای مانور دارند) درنظر گرفته نشده‌است. به همین دلیل برنامه‌های فعلی برای شطرنج و بازی‌های دیگر از ترکیب غیرخطی مشخصه‌ها استفاده می‌کنند. برای مثال، ممکن است ارزش یک جفت فیل در کنار هم کمی بیش‌تر از دو برابر ارزش یک فیل تنها باشد و یا شاید ارزش یک فیل در انتهای بازی بیشتر از ابتدای بازی باشد.

به این نکته زیرکانه توجه داشته باشید که مشخصه‌ها و وزن‌ها جزئی از قوانین بازی شطرنج نیستند، بلکه از آزمایشات و تجربیات چند قرن بشریت نتیجه گرفته شده‌اند. فرض کنید تابع ارزیابی به صورت خطی باشد، که در آن مشخصه‌ها و وزن‌ها بهترین تقریب از ترتیب واقعی حالت‌ها را بر اساس مقادیر آن‌ها تعیین می‌کنند. بر این اساس، آزمایشات نشان می‌دهند که اگر «مقدار مادی» امنیت برای یک موقعیت خاص، بیش‌تر از یک واحد باشد احتمالاً در بازی برنده خواهیم شد و در شرایط مشابه اگر این مقدار برابر سه واحد باشد آنگاه ما با احتمال قریب به یقین برنده بازی هستیم. در بازی‌هایی که این نوع تجربیات و آزمایشات وجود ندارد وزن‌های تابع ارزیابی توسط تکنیک یادگیری ماشین بدست می‌آیند. اعمال این تکنیک‌ها بر روی شطرنج دوباره بر این ادعا تأکید می‌کند که ارزش یک فیل تقریباً سه برابر سرباز است.

---

[1]- Material Value

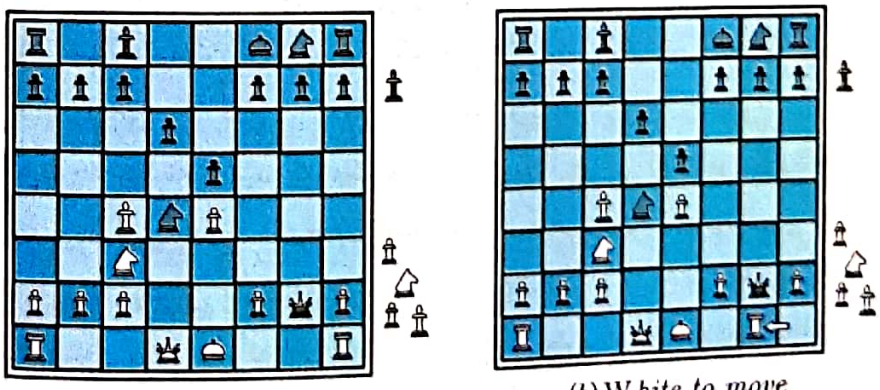
۶-۴-۲- قطع جستجو

مرحله‌ی بعدی برای اصلاح الگوریتم ALPHA-BETA-SEARCH این است که این الگوریتم بعد از این که زمان قطع جستجو فرا رسید، تابع ابتکاری EVAL را صدا کند. برای پیاده‌سازی این موضوع، دو خط موجود در شکل ۶-۷ را که به TERMINAL-TEST اشاره می‌کنند، با دستور زیر جایگزین می‌کنیم:

If CUTOFF-TEST (state, depth) then return EVAL (state)

از طرفی باید افزایش عمق فعلی را در هر بار فراخوان بازگشتی ثبت کنیم. بهترین راه برای کنترل تعداد جستجوها، تعیین یک کران ثابت برای عمق است، با این کار CUTOFF-TEST(state,depth) به ازای همه‌ی عمق‌هایی که بزرگتر از عمق ثابت d هستند مقدار true برمی‌گرداند (این تابع دقیقاً مشابه TERMINAL-TEST برای همه‌ی حالت‌های پایانی نیز مقدار true برمی‌گرداند). عمق d به‌گونه‌ای انتخاب می‌شود که زمان لازم برای اجرای آن از زمان تعیین شده توسط قوانین بازی بیش‌تر نشود.

روش قوی‌تر این است که الگوریتم عمیق‌شونده تکراری که در فصل ۳ توضیح داده شد را بر روی آن اعمال و هنگامی که زمان تعیین شده پایان یافت، حرکتی را که توسط عمیق‌ترین جستجوی کامل انتخاب شده برگردانیم. بااین‌حال، به‌دلیل تقریبی بودن تابع ارزیابی ممکن است این روش‌ها به خطا منجر شوند. تابع ارزیابی ساده‌ای که براساس «مقدار مادی[1]» برای شطرنج در نظر گرفته شده‌است را در نظر بگیرید. فرض کنید برنامه تا عمق تعیین شده جستجو کند، و به موقعیت شکل ۶-۸ (b) برسد، که در آن سیاه با یک اسب و دو سرباز پیش است. بنابراین تابع ارزیابی آن را به عنوان مقدار ابتکاری این حالت معرفی می‌کند، بنابراین نتیجه می‌گیرد که این حالت به نفع سیاه است. در صورتی که سفید با حرکت بعدی خود بدون اینکه مهره‌ای از آن در خطر باشد وزیر سیاه را مال خود می‌کند. لذا این موقعیت در حقیقت به نفع سفید است، اما این حقیقت را زمانی می‌توان دریافت که یک گام بیش‌تر رو به جلو برداریم.



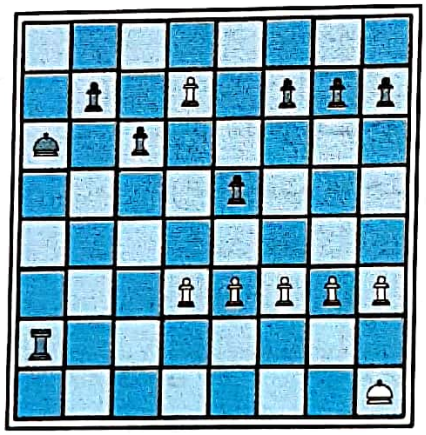
(a) White to move (b) White to move

شکل ۶-۸ دو موقعیت در شطرنج با اندکی تفاوت. در (a) سیاه دارای مزیت یک اسب و دو سرباز است و بازی را می‌برد. در (b) سیاه با از دست دادن وزیر، بازی را واگذار خواهد کرد.

[1]- مقدار مادی یعنی صرفاً براساس شمارش تعداد و ارزش مهره‌های باقیمانده مستقل از محل مهره‌ها است.

بنابراین روشن است که نیاز به تابع cutoff-test پیشرفته‌تری داریم. تابع ارزیابی باید تنها روی موقعیت‌های ساکن[۱] اعمال شود (یعنی موقعیت‌هایی که در آن‌ها پرش‌های ناگهانی روی مقدار ارزیابی اتفاق نمی‌افتد). برای مثال، در شطرنج موقعیت‌هایی که در آن‌ها مهره‌ای از صفحه شطرنج حذف می‌شود، برای تابع ارزیابی که در آن تنها ارزش مادی در نظر گرفته می‌شود، ساکن محسوب نمی‌شوند. می‌توان تا زمانی که به موقعیت‌های ساکن نرسیدیم موقعیت‌های غیرساکن را بسط دهیم. این جستجوی اضافی، «جستجوی سکون[۲]» گفته می‌شود که برای این کار تنها نوع خاصی از حرکت‌ها را مدنظر قرار می‌دهیم (مثل حرکت‌هایی که منجر به حذف یک مهره در شطرنج می‌شود) که موجب برطرف شدن عدم قطعیت درباره موقعیت مورد نظر می‌شود.

البته حذف اثر افق[۳] بسیار مشکل‌تر است. این اثر زمانی اتفاق می‌افتد که برنامه با حرکتی از حریف مواجه می‌شود که آسیبی جدی و غیرقابل جبران به آن وارد می‌کند. موقعیت شطرنج در شکل ۶-۹ را در نظر بگیرید. سیاه از نظر مادی (تعداد مهره‌ها در این مثال) پیش است، اما اگر سفید سرباز خود را از خانه‌ی هفتم به خانه‌ی هشتم ببرد، سرباز وزیر می‌شود، و پیروزی را برای سفید بسیار راحت می‌کند. سیاه می‌تواند این کار را با کیش دادن سفید از طریق رخ خود ۱۴ مرحله عقب بیاندازد اما این کار بی‌فایده است و سرانجام سرباز تبدیل به وزیر می‌شود. مشکل جستجو با عمق- ثابت این است که فکر می‌کند این کیش دادن‌ها می‌تواند از وزیر شدن سرباز جلوگیری کند بنابراین جستجو، حرکت تبدیل سرباز به وزیر را به سمت جستجوی افق سوق می‌دهد، یعنی جایی که جستجو نمی‌تواند آن را کشف کند.



Black to move

شکل ۶-۹ اثر افق. یک سری کیش دادن توسط رخ سیاه حرکت حتمی تبدیل سرباز به وزیر را در فراسوی افق قرار می‌دهد و موقعیت را به‌گونه‌ای تصویر می‌کند که سیاه برنده است در حالی که درحقیقت سفید بازی را می‌برد.

به دلیل پیشرفت سخت‌افزار، جستجوها با عمق بیش‌تری انجام می‌شوند، بنابراین انتظار داریم اثر افق کمتر اتفاق بیفتد (دنباله‌هایی با تأخیر بسیار زیاد کمیاب‌اند). بسط واحد[۴] بدون اینکه هزینه‌ی زیادی را به ما تحمیل کند در جلوگیری از اثر افق بسیار مؤثر است. بسط واحد، انتخاب حرکتی است که در یک موقعیت داده شده، لزوماً

۱- Quiescent
۲- Quiescence search
۳- Horizon effect
۴- Singular extension

بهترین حرکت باشد. از آن‌جا که فاکتور انشعاب بسط واحد ۱ است، می‌تواند بدون اینکه هزینه‌ی زیادی را به ما تحمیل کند با محدودیت استانداردی روی عمق اجرا شود. جستجوی سکون[۱] را می‌توان نوعی از جستجوی واحد در نظر گرفت. در شکل ۶-۹ جستجوی بسط واحد حرکت تبدیل سرباز به وزیر را می‌یابد البته با این فرض که کیش دادن و حرکات شاه سفید به عنوان حرکت‌هایی «لزوماً بهتر» از دیگر حرکت‌ها شناخته شوند.

تا این‌جا ما درباره‌ی قطع جستجو در یک سطح مشخص و عدم تأثیر هرس آلفا-بتا بر نتیجه‌ی نهایی آن صحبت کردیم. اما می‌توان «هرس پیشرو» را نیز امتحان کرد، بدین معنی که بعضی از حرکت‌ها در یک گره خاص به‌سرعت و بدون هیچ توجهی هرس می‌شوند. بدیهی است که اکثر انسان‌ها در بازی شطرنج تنها برروی تعداد اندکی از حرکت‌های ممکن از یک موقعیت خاص فکر می‌کنند (حتی گاهی اوقات به عمد). متأسفانه، این روش بسیار خطرناک است زیرا هیچ تضمینی وجود ندارد که بهترین حرکت هرس نشود. اگر این روش نزدیک ریشه اعمال شود، ممکن است فاجعه‌آمیز باشد، زیرا ممکن است برنامه بسیاری از حرکت‌های بدیهی را از دست بدهد. هرس پیشرو در موقعیت‌های خاصی از مسئله می‌تواند با اطمینان اعمال شود، مثلاً اگر دو حرکت متقارن و یا معادل باشند، آنگاه در نظر گرفتن یکی از آن دو گره کافی به نظر می‌رسد.

ترکیب تمامی این روش‌ها می‌تواند منجر به برنامه‌ای شود که شطرنج (یا بازی‌های دیگر) را بسیار کم-اشتباه بازی کند. بگذارید فرض کنیم که یک تابع ارزیابی، یک تست قطع منطقی با جستجوی سکون، و یک جدول انتقال بزرگ را برای شطرنج پیاده‌سازی کرده‌ایم. علاوه بر این فرض کنید، بعد از کلی تلاش، توانسته‌ایم یک میلیون گره در ثانیه را در یکی از جدیدترین کامپیوترها تولید و ارزیابی کنیم که در این صورت می‌توانیم در زمان استاندارد برای هر حرکت (۳ دقیقه) ۲۰۰ میلیون گره را جستجو کنیم. فاکتور انشعاب برای شطرنج تقریباً ۳۵ است و 35^5 تقریباً برابر ۵۰ میلیون می‌شود، بنابراین اگر از جستجوی minimax استفاده کنیم می‌توانیم تا ply ۵ جلوتر را بررسی کنیم. اگرچه این برنامه ناکارآمد نیست، اما چنین برنامه‌ای می‌تواند توسط یک بازیکن متوسط شطرنج ، که معمولاً قادر است برای شش یا هشت ply در آینده تصمیم‌گیری کند فریب داده شود. با استفاده از جستجوی آلفا- بتا می‌توانیم به ۱۰ ply برسیم که سطحی ماهرانه در بازی محسوب می‌شود. بخش ۶-۷ تکنیک‌های جدیدی از هرس کردن را معرفی می‌کند که عمق مؤثر جستجو را به ۱۴ ply افزایش می‌دهد. برای رسیدن به سطح قهرمانان گراندمستر ما نیاز به یک تابع ارزیابی بسیار قوی و پایگاه داده‌ای بزرگ از حرکت‌های بهینه‌ی ابتدا و انتهای بازی داریم. می‌توان از سوپرکامپیوترها برای این منظور استفاده کرد.

۶-۵- بازی‌هایی که در آن‌ها عنصر شانس دخیل است

در زندگی واقعی، گاهی حوادث غیرقابل پیش‌بینی خارجی، ما را تحت شرایط نامشخصی قرار می‌دهند. بسیاری از بازی‌ها نیز به دلیل وجود عنصری تصادفی مثل تاس، به یک واقعه غیرقابل پیش‌بینی تبدیل شده‌اند که البته از این نظر به واقعیت نزدیک‌ترند. بنابراین سودمند است که تأثیر این ویژگی را بر روی فرآیند تصمیم‌گیری بررسی کنیم.

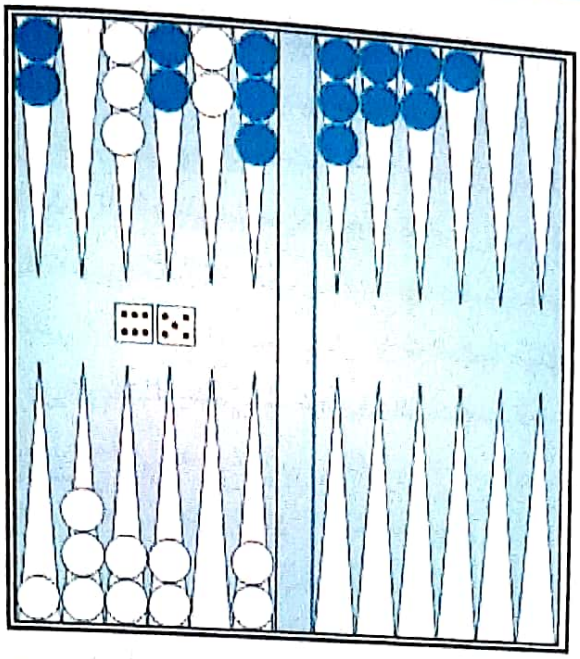
تخته‌نرد، بازی‌ای است که در آن دو عنصر شانس و مهارت با هم آمیخته شده‌اند. برای تعیین حرکت مجاز، هر بازیکن در نوبت خود تاس را پرتاب می‌کند. برای مثال در تخته‌نرد نشان داده شده در شکل ۶-۱۰، سفید تاس

۱- Quiescence search

انداخته و تاس‌ها اعداد ۶-۵ را نشان می‌دهند، بنابراین چهار حرکت مجاز وجود دارد. اگرچه سفید از حرکت‌های مجاز خود مطلع است اما از حرکت‌های مجاز سیاه بی‌خبر است زیرا نمی‌داند سیاه چگونه تاس می‌اندازد. یعنی، سفید نمی‌تواند هم‌چون بازی‌هایی مثل شطرنج و تیک- تاک- تو یک درخت بازی استاندارد بسازد. درخت بازی در تخته‌نرد علاوه بر گره‌های MAX و MIN باید شامل گره‌های شانس باشد. گره‌های شانس در شکل ۶-۱۱ به شکل دایره نشان داده شده‌اند. شاخه‌هایی که از هر یک از گره‌های شانس به وجود می‌آیند نشان‌دهنده‌ی انواع اعداد ممکن حاصل از انداختن تاس است که هر یک از آن‌ها با احتمال وقوع آن برچسب‌گذاری شده‌اند. از ریختن دو تاس ۳۶ حالت حاصل می‌شود (که همگی شانس یکسانی دارند)، اما چون ۶-۵ و ۵-۶ مثل یکدیگرند، تنها ۲۱ حالت مختلف وجود دارد. شش حالت جفت (۱-۱ تا ۶-۶) دارای احتمال $\frac{۱}{۳۶}$ و بقیه‌ی حالت‌ها دارای احتمال $\frac{۱}{۱۸}$ هستند.

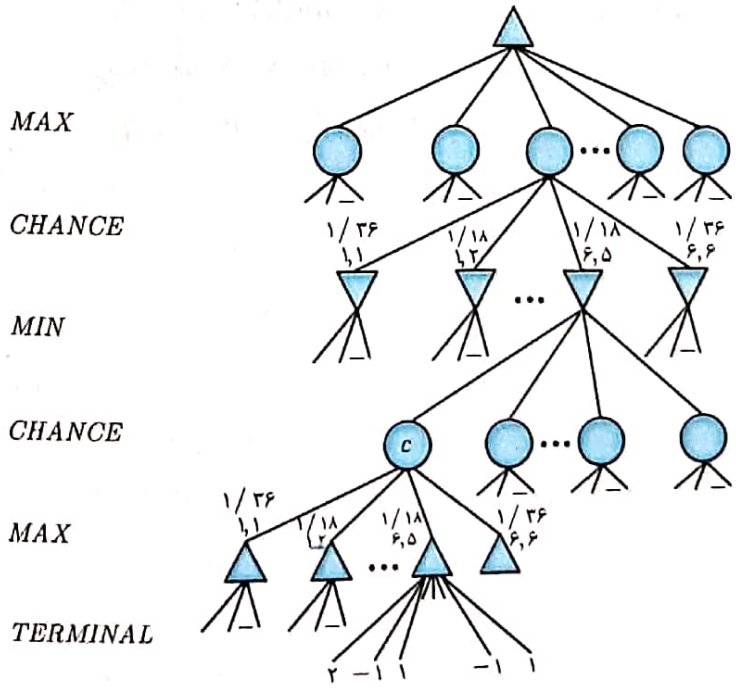
حال باید بیاموزیم که در این بازی‌ها چگونه تصمیمی صحیح اتخاذ نماییم. بدیهی است که هم‌چنان می‌خواهیم حرکتی را انجام دهیم که به بهترین موقعیت منجر می‌شود. در هر صورت، موقعیت‌های حاصل دارای مقادیر minimax دقیقی نیستند. بنابراین ما تنها می‌توانیم مقادیر مورد انتظار (که در آن همه‌ی حالت‌های ممکن برای انداختن تاس در نظر گرفته می‌شود) را محاسبه کنیم. بنابراین مفهوم «مقدار minimax» در بازی‌های قطعی را به صورت تعریف «مقدار minimax مورد انتظار» برای بازی‌هایی با گره‌های شانس بسط می‌دهیم. گره‌های پایانی و گره‌های MAX و MIN (برای حالت‌هایی که اعداد تاس‌ها مشخص شده‌اند) دقیقاً مثل گذشته عمل می‌کنند. گره‌های شانس از طریق محاسبه‌ی میانگین وزن‌دار مقادیر حاصله از همه‌ی حالات ممکن برای تاس ارزیابی می‌شوند، یعنی:

$$\text{EXPECTIMINIMAX} = \begin{cases} \text{UTILITY}(n) & \text{اگر گره n یک حالت پایانی باشد} \\ \max_{s \in \text{Successor}(n)} \text{EXPECTIMINIMAX}(s) & \text{اگر گره n یک max باشد} \\ \min_{s \in \text{Successor}(n)} \text{EXPECTIMINIMAX}(s) & \text{اگر گره n یک min باشد} \\ \sum_{s \in \text{Successor}(n)} P(s) \cdot \text{EXPECTIMINIMAX}(s) & \text{اگر گره n یک شانس باشد} \end{cases}$$



شکل ۶-۱۰ یک موقعیت معمول در بازی تخته نرد. هدف بازی این است که هر بازیکن مهره‌های خود را به بیرون از تخته ببرد. سفید از خانه‌ی ۲۵ به صورت ساعت‌گرد، و سیاه به‌صورت پادساعت گرد از خانه ۰ حرکت می‌کنند. یک مهره می‌تواند به هر خانه‌ای برود مگر اینکه بیش از یک مهره‌ی حریف در آنجا باشد: اگر تنها یک مهره از حریف در خانه‌ای باشد، این مهره دستگیر شده و باید کار خود را از ابتدا شروع کند. در موقعیت نشان داده شده، سفید تاس ۵-۶ ریخته، و باید از میان ۴ حرکت مجاز خود انتخاب کند: (۱۱–۱۰,۵–۵)، (۲۴–۱۱,۱۹–۵)، (۱۶–۱۰,۱۰–۵)، (۱۶–۱۱,۱۱–۵).

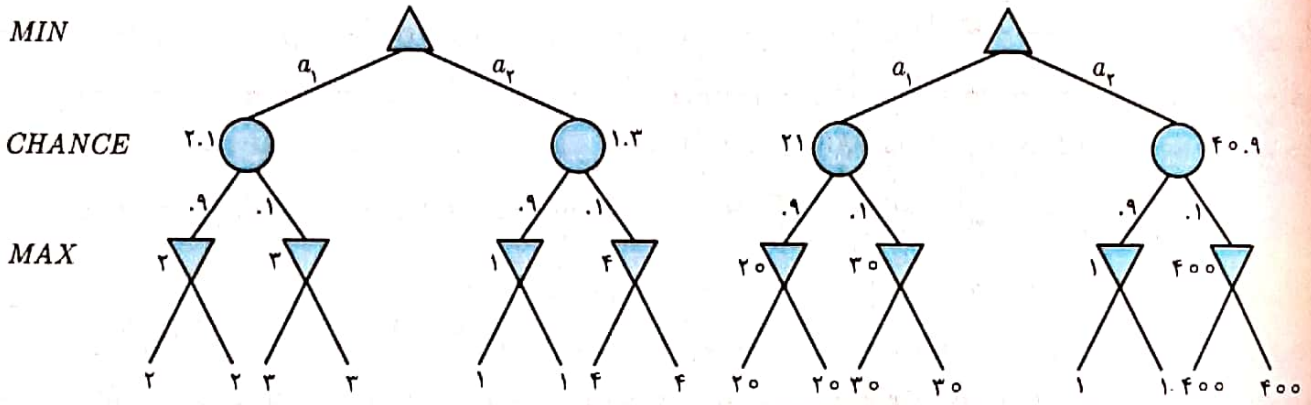
که در آن تابع پسین (برای گره شانس n)، حالت n را از طریق تولید گره پسین s (همه‌ی حالات حاصله‌ی ممکن از انداختن تاس‌ها) و P(s) (احتمال وقوع هر یک از این حالات) ارتقاء می‌دهد. این معادلات می‌توانند به‌صورت بازگشتی به سمت بالا و تا ریشه‌ی درخت حرکت کند (دقیقاً هم‌چون minimax). جزئیات الگوریتم به‌عنوان تمرین بر عهده‌ی خواننده گذاشته شده‌است.



شکل ۶-۱۱ درخت جستجو برای موقعیت‌های بازی تخته‌نرد.

۶-۵-۱- ارزیابی موقعیت‌ها در بازی‌هایی با گره‌های شانس

همچون minimax، برای تخمین «minimax مورد انتظار» باید جستجو را در نقطه‌ای قطع و تابع ارزیابی را بر روی تک تک برگ‌ها اعمال کنیم. ممکن است فکر کنید توابع ارزیابی در بازی‌هایی مثل تخته‌نرد هم‌چون توابع ارزیابی برای بازی‌هایی مثل شطرنج (توابعی که در آن به موقعیت‌های بهتر عدد بالاتری نسبت داده می‌شود) است. اما وجود گره‌های شانس، به ما هشدار می‌دهد که درباره‌ی تعریف مقادیر ارزیابی دقت کنیم. شکل ۶-۱۲ مراحل کار را نشان می‌دهد: درتابع ارزیابی که مقادیر [۱,۲,۳,۴] را به برگ‌ها نسبت می‌دهد، A_1 بهترین حرکت است و در صورتی که این مقادیر به صورت [۱,۲۰,۳۰,۴۰۰] باشد، آنگاه A_2 بهترین حرکت است. در نتیجه، با تغییر مقیاس بعضی از مقادیر ارزیابی، برنامه به گونه‌ای کاملاً متفاوت عمل می‌کند. برای از بین بردن این نقص، تابع ارزیابی باید یک تبدیل خطی مثبت از احتمال برد یک موقعیت خاص باشد (در حقیقت، همان «سودمندی مورد انتظار» آن موقعیت). این موضوع یکی از ویژگی‌های مهم و عمومی مسئله‌هایی است که در آن‌ها عدم قطعیت وجود دارد.



شکل ۶-۱۲ یک تبدیل حفظ‌کننده، ترتیب مقادیر برگ‌های بهترین حرکت را عوض می‌کند.

۶-۵-۲- پیچیدگی minimax مورد انتظار

اگر برنامه از قبل از نتایج تمامی تاس‌های انداخته شده تا پایان بازی مطلع باشد، حل بازی با تاس دقیقاً شبیه حل مسئله بدون تاس خواهد بود که minimax آن را در زمان $O(b^m)$ حل می‌کرد. از آنجایی که الگوریتم «minimax مورد انتظار» تمامی حالات به‌وجود آمده از انداختن تاس را در نظر می‌گیرد در زمان $O(b^m n^m)$ قابل اجراست (که در آن n تعداد حالات متفاوت از انداختن تاس است).

حتی اگر عمق جستجو به عمق کوچک d محدود شود، هزینه‌ی اضافی آن در مقایسه با minimax شرایط را برای بررسی و پیش‌بینی آینده‌ی دور در بازی‌های شانسی غیرعملی می‌کند. در تخته نرد n ، ۲۱ و b معمولاً ۲۰ است (اما در بسیاری از موارد (برای بازی‌هایی با تاس‌های دوتایی) این مقدار می‌تواند تا اندازه‌ای برابر ۴۰۰۰ نیز بزرگ باشد). در این موارد حداکثر قادر هستیم تا سه ply را بررسی و مدیریت کنیم.

می‌توان به شکل دیگری به مسئله نگاه کرد: مزیت الگوریتم آلفا-بتا این بود که از پیشامدهایی که در آینده هرگز اتفاق نمی‌افتند صرف نظر و بر روی رویدادهای محتمل‌تر تمرکز می‌کرد. اما در بازی‌های با تاس، هیچ دنباله‌ای از حرکت‌های محتمل‌تر وجود ندارد زیرا وقوع هر حرکتی نیازمند انداختن تاس است و بعد از آن تاس تصمیم

می‌گیرد که حرکت مورد نظر مجاز است یا خیر. این موضوع یک مشکل کلی در حل مسائلی است که در آن‌ها عدم قطعیت وجود دارد یعنی «احتمالات در مسائل به‌شدت زیاد می‌شوند، و برنامه‌ریزی‌های دقیق انجام شده را بی‌نتیجه می‌گذارند (زیرا ممکن است وقایع آنگونه که ما پیش‌بینی کردیم اتفاق نیافتند)».

بدون شک، خوانندگان به مواردی از درخت‌های بازی (که در آن‌ها گره‌های شانس وجود دارد) برخورد خواهند کرد که بتوان روی آن‌ها الگوریتم هرس آلفا-بتا را نیز اعمال کرد. بنابراین این کار نشدنی نیست. در واقع در صورتیکه برای عدد «سودمندی مورد انتظار» بتوان دامنه‌ی محدود تعریف کرد، می‌توان نشان داد که امکان هرس الفا – بتا وجود دارد. با کمی ذکاوت می‌توانیم گره‌های شانس را نیز هرس کنیم (اگرچه تحلیل گره‌های MIN و MAX هم‌چنان بدون تغییر باقی می‌ماند). گره شانس C در شکل ۶-۱۱ را در نظر بگیرید، دقت کنید که هنگامی که فرزندانش را امتحان و ارزیابی می‌کنیم، مقدار آن چه تغییری می‌کند. آیا می‌توان بدون بررسی فرزندان گره C کران بالایی برای آن در نظر گرفت؟ (به یاد داشته باشید این دقیقاً چیزی است که آلفا-بتا برای هرس زیردرخت مربوط به این گره نیاز دارد). در نگاه اول احتمالاً غیرممکن به نظر می‌آید، زیرا مقدار C برابر میانگین مقادیر فرزندانش است و تا زمانی که تمامی حالت‌های انداختن تاس را در نظر نگیریم، میانگین ممکن است هر مقداری را بپذیرد، زیرا فرزندان بررسی نشده می‌توانند هر مقداری داشته باشند. اما اگر کران‌هایی را برای مقادیر ممکن تابع سودمندی درنظر بگیریم، آنگاه می‌توانیم کران‌هایی را برای میانگین تعیین کنیم. برای مثال، اگر فرض کنیم تمامی مقادیر سودمندی، بین ۳+ و ۳- هستند، آنگاه مقدار برگ‌ها کران‌دار خواهد شد، در نتیجه می‌توانیم کران بالایی را برای گره‌های شانس، بدون در نظر گرفتن فرزندان آن تعیین کنیم.

۶-۶- پیشرفته‌ترین بازی‌های روز

خیلی‌ها فکر می‌کنند، بازی‌ها در هوش‌مصنوعی، همچون موتور مسابقه Grand prix در صنعت ماشین است: پیشرفت برنامه‌های بازیکن بسیار سریع است. ماشین‌های پیشرفته‌ای به وجود آمده‌اند که در آن‌ها بسیاری از تکنیک‌های پیشرفته مهندسی گردآوری شده‌اند، البته این ماشین‌ها کاربرد زیادی در بازار فروش ندارند. اگرچه بسیاری از محققان معتقدند که انجام بازی‌ها با استفاده از هوش‌مصنوعی جدا از جریان اصلی هوش‌مصنوعی است، اما همچنان این روند ادامه دارد و جریانی هیجان‌آور و نوآور را در جامعه هوش‌مصنوعی به‌وجود می‌آورد.

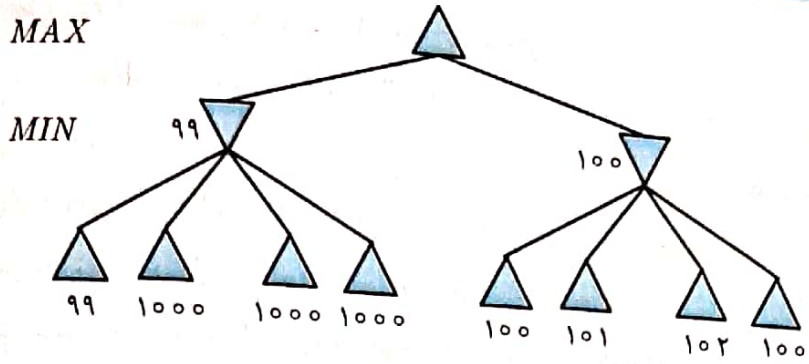
شطرنج: در سال ۱۹۵۷، هربرت سیمون، پیش‌بینی کرد که ظرف ۱۰ سال آینده کامپیوترها قهرمانان جهانی جامعه انسانی را شکست می‌دهند. چهل سال بعد، برنامه Deep Blue در یک مسابقه شش-گیم کاسپاروف را شکست داد. سیمون اشتباه کرد، اما تنها با ضریبی از ۴. کاسپاروف نوشت:

گیم دوم سرنوشت ساز بود، خاطره‌ی بدی از آن دارم... شاهد آن هستیم که شرایط در جهت برآورده کردن انتظارات ما از یک کامپیوتر برای پیش‌بینی نتیجه‌ی بسیاری از تصمیمات در آینده خوب پیش می‌رود. در این بازی، ماشین از رفتن به موقعیت‌هایی که نفع کوتاه‌مدت داشت خودداری کرد که رقابت تنگاتنگی را با انسان ایجاد کرده و انسان موقعیت خود را در خطر دیده است. (کاسپاروف، ۱۹۹۷). Deep Blue، توسط ماری کمپ بل، فنگ سینگ هد، جوزف هان در شرکت IBM گسترش یافت. ماشین برنده، یک ماشین موازی با ۳۰ پروسسور IBM RS/۶۰۰۰ بود که بر روی آن برنامه‌ی فوق‌الذکر قرار داشت. هم‌چنین ۴۸۰ پروسسور VLSI شطرنج وجود داشت که اجرای حرکات تولید شده (که شامل ترتیب حرکات نیز می‌شد) را برعهده داشتند. به-

طور متوسط Deep Blue ۱۲۶ میلیون گره در ثانیه را جستجو می‌کرد. که حداکثر سرعت آن در بازی ۳۳۰ میلیون گره در ثانیه بود. این برنامه در هر حرکت ۳۰ بیلیون موقعیت را تولید کرد که تقریباً با این تعداد گره به عمق ۱۴ می‌توان دسترسی پیدا کرد. قلب ماشین، "جستجوی آلفا- بتا به صورت عمیق‌کننده تکراری استاندارد" و استفاده از یک "جدول انتقال" بوده‌است. اما به نظر می‌آید دلیل اصلی پیروزی آن، توانایی آن در بسط گره‌ها تا عمقی بیش از حد تعیین شده بوده‌است. در بعضی از موارد جستجو به عمق ۴۰ ply رسید. تابع ارزیابی دارای بیش از ۸۰۰۰ مشخصه بود که بسیاری از آن‌ها الگوهای خاصی از مهره‌ها را توصیف می‌کرد. در این برنامه از ۴۰۰۰ موقعیت متفاوت از کتاب‌های مختلف و از ۷۰۰۰۰۰ بازی گراندمستر استفاده شد. این سیستم هم‌چنین از پایگاه‌داده‌ای بسیار بزرگ برای لحظات پایان بازی استفاده کرد که شامل تمامی موقعیت‌های ممکن با ۵ و ۶ مهره بود. این پایگاه‌داده بر روی گسترش عمق جستجو بسیار مؤثر بود و Deep Blue را قادر ساخت که در بسیاری از موارد که از مات کردن حریف بسیار دور بود بسیار بی‌نقص بازی کند.

۶-۷- بحث

از آنجا که محاسبه‌ی تصمیمات بهینه در برخی از موارد بسیار مشکل است، همه‌ی الگوریتم‌ها باید بتوانند فرضیات و تقریب‌های مناسبی را انجام دهند. روش استاندارد (که براساس minimax، توابع ارزیابی، آلفا- بتا است) تنها یکی از روش‌ها برای انجام این کار است. شاید چون این روش بسیار زود پیشنهاد شد، این روش استاندارد به‌طور گسترده توسعه یافت و بر روش‌های دیگر پیشی گرفت. بعضی معتقدند این موضوع باعث شد که بازی‌ها از جریان تحقیقات هوش‌مصنوعی فاصله بگیرند، زیرا این روش فضای زیادی برای ایجاد اندیشه‌های جدید در زمینه الگوریتم‌های تصمیم‌گیری ندارد. در این بخش ما به روش‌های دیگر می‌پردازیم. بگذارید ابتدا minimax را در نظر بگیریم. minimax حرکت بهینه را در درخت جستجوی داده شده، برمی‌گزیند، با این فرض که ارزیابی برگ‌ها به شکل دقیق انجام می‌شود. در واقعیت، ارزیابی‌ها، تقریبی خام از ارزش یک موقعیت هستند و خطاهای زیادی در آن‌ها صورت می‌پذیرد. در شکل ۶-۱۳ درخت ۲-ply یک بازی را نشان می‌دهد که در آن اعمال الگوریتم minimax نامناسب به نظر می‌رسد. minimax پیشنهاد می‌کند که شاخه‌ی دست راست را ادامه دهیم، در صورتی که احتمال اینکه مقدار واقعی دست چپ بزرگ‌تر باشد بیش‌تر است. انتخاب minimax بر این فرض استوار است که تمامی گره‌هایی که دارای مقادیر ۱۰۰، ۱۰۱، ۱۰۲ هستند بهتر از گره برچسب زده شده با ۹۹ هستند. در حالی که از آنجا که گره برچسب زده شده با ۹۹، یک هم‌زاد با برچسب ۱۰۰۰ دارد، احتمالاً مقدار واقعی بزرگ‌تری دارد. یک راه برای حل این مشکل، داشتن یک تابع ارزیابی است که نحوه‌ی توزیع احتمال بر روی مقادیر موجود را برگرداند. آنگاه می‌توانیم از طریق استفاده از تکنیک‌های آماری، نحوه‌ی توزیع احتمالات بر روی مقادیر والدین را محاسبه کنیم. متأسفانه، معمولاً مقادیر هم‌زادها بسیار هم-بسته‌اند بنابراین به محاسبات پرهزینه و اطلاعات زیادی نیازمندیم.



شکل ۶-۱۳ درخت بازی ۲-ply که اعمال الگوریتم minimax بر روی آن ممکن است مناسب نباشد.

در مرحله بعدی، به الگوریتم جستجویی که درخت را تولید می‌کند دقت می‌کنیم. هدف هر الگوریتمی این است که به سرعت محاسبات را انجام دهد و حرکت مناسبی را انتخاب کند. بدیهی‌ترین مشکل الگوریتم آلفا- بتا این است که تنها برای انتخاب یک حرکت مناسب طراحی نشده‌است بلکه برای محاسبه‌ی کران‌هایی برای مقادیر تمامی حرکت‌های مجاز طراحی شده‌است. برای درک اینکه این اطلاعات اضافی غیرضروری است، موقعیتی را در نظر بگیرید که در آن تنها یک حرکت مجاز وجود دارد. در این حالت جستجوی آلفا- بتا هم‌چنان حجم زیادی از درخت جستجو را ارزیابی و تولید می‌کند که عملاً کاربردی برای ما ندارد. اگرچه می‌توانیم تستی برای اجتناب از وقوع این حالت در الگوریتم قرار دهیم، اما این در حقیقت پوشاندن مشکل اصلی الگوریتم است زیرا بسیاری از محاسبات انجام شده در الگوریتم آلفا-بتا نامرتبط با هدف مسئله است. داشتن یک حرکت مجاز تفاوت چندانی با حالتی که در آن حرکت‌های زیادی وجود دارند اما تنها یکی از آن‌ها مناسب و بقیه بسیار نامناسب هستند ندارد. در چنین شرایط بدیهی، به جای اتلاف زمانی که ممکن است در آینده در موقعیت‌های پیچیده‌تری مورد نیاز باشد بهتر است بعد از جستجوی کوتاهی، سریعاً تصمیم‌گیری کنیم. این مسئله منجر به ایده‌ای به نام «سودمندی بسط یک گره» شده‌است. یک الگوریتم جستجوی مناسب باید گره‌هایی را برای بسط دادن انتخاب کند که دارای مقدار سودمندی بالا هستند (بدین معنی که گره‌هایی که احتمال بیش‌تری برای کشف یک حرکت مناسب دارند بسط داده شوند). اگر گره‌ای وجود ندارد که سودمندی بسط آن بالاتر از از هزینه آن باشد (در واحد زمان) آنگاه الگوریتم باید جستجو را متوقف کرده و یک حرکت را انتخاب کند. توجه داشته باشید که این عمل، تنها برای موقعیت‌های بدیهی صدق نمی‌کند بلکه در مواردی که حرکت‌ها بصورت متقارن (حرکت‌هایی که درآن‌ها هیچ‌گاه جستجو نشان نمی‌دهد که کدام حرکت بهتر از دیگری است) هستند نیز صدق می‌کند.

این نوع از استدلال که در آن درباره‌ی استفاده از یکی از روش‌های محاسباتی تصمیم‌گیری می‌شود، استدلال ماوراء[1] نامیده می‌شود (که در آن استدلال درباره چگونگی استدلال است). این نوع استدلال نه تنها روی بازی‌ها بلکه روی انواع مختلف استدلال‌ها اعمال می‌شود. تمامی محاسبات، در جهت جریانی برای تلاش برای تصمیم‌گیری بهتر انجام می‌شود، تمامی آن‌ها دارای هزینه هستند ، و این احتمال در تمامی آن‌ها وجود دارد که در مواردی، کیفیت تصمیم‌گیری را بهبود ببخشند. استدلال آلفا- بتا ساده‌ترین نوع از استدلال ماوراء را در خود دارد، تئوری که بر اساس آن از بررسی شاخه‌هایی از درخت می‌توان صرف‌نظر کرد بدون اینکه خسارتی به ما وارد شود.

---

[1]- Meta-reasoning

بگذارید ماهیت جستجوها را دوباره مورد بررسی قرار دهیم. الگوریتم‌هایی که برای جستجوی ابتکاری و بازی‌ها به وجود آمدند همگی با تولید یک دنباله از حالت‌ها به صورت گسسته، با شروع از یک حالت اولیه، و با اعمال تابع ارزیابی بر روی آن‌ها کار می‌کنند. بدیهی است که این همان روشی نیست که انسان خود در بازی‌ها از آن استفاده می‌کند. مثلاً در شطرنج، انسان هدف مشخصی را در ذهنش مجسم می‌کند، برای مثال گرفتن وزیر حریف، سپس نقشه‌های خود را بر این اساس طراحی می‌کند. این نوع از استدلال یا تصمیم‌گیری «هدف مدارانه» گاهی اوقات جستجوی ترکیبی را شکست می‌دهد. DAVID WILKINS PARADISE تنها برنامه‌ای است که در آن از «استدلال هدف‌مدارانه» به‌طور موفق برای بازی شطرنج استفاده شد: این برنامه قادر بود برخی از مسائل شطرنج را با ۱۸ ترکیب مختلف از حالات حل کند. تاکنون هیچ نتیجه‌ای برای چگونگی ترکیب دو نوع الگوریتم و تبدیل آن به یک سیستم کارا و نیرومند وجود ندارد.

۶-۸- خلاصه

در این فصل بازی‌های مختلفی را بررسی کردیم تا بفهمیم منظور از بازی به‌صورت بهینه چیست و دریابیم که چگونه می‌توان بدون اشکال بازی کرد. مهم‌ترین ایده‌های شرح داده شده بصورت زیر است:

- یک بازی می‌تواند توسط یک حالت اولیه (وضعیت ابتدایی صفحه)، کنش‌های مجاز برای هر حالت، تست پایانی (تستی که می‌گوید چه زمانی بازی به پایان رسیده‌است)، و یک تابع سودمندی که بر روی حالت‌های پایانی اعمال می‌شود تعریف شود.
- در بازی‌های دونفره مجموع- صفر با اطلاعات کامل، الگوریتم minimax می‌تواند با استفاده از جستجوی عمق- اول روی درخت بازی حرکت‌های بهینه را انتخاب کند.
- الگوریتم جستجوی آلفا- بتا هم‌چون الگوریتم minimax حرکت‌های بهینه را انتخاب می‌کند، اما به‌دلیل حذف زیردرخت‌های نامرتبط کارایی بیش‌تری دارد.
- گاهی اوقات بررسی تمام درخت غیرعملی است (حتی با الگوریتم آلفا- بتا) بنابراین نیاز داریم که جستجو را در نقطه‌ای قطع کرده و تابع ارزیابی (که سودمندی یک حالت را به ما نشان می‌دهد) را بر روی آن اعمال کنیم.
- بازی‌های شانسی، با نسخه‌ی بسط داده شده الگوریتم minimax قابل حل هستند که در آن گره‌های شانس با گرفتن میانگین وزن‌دار از سودمندی گره‌های فرزندان (که در آن‌ها وزن‌ها همان احتمال فرزندان هستند) ارزیابی می‌شوند.
- بازی به‌صورت بهینه در بازی‌هایی با اطلاعات ناقص، نیازمند استدلال درباره حالت‌های جاری و آینده هر بازیکن دارد. یک تقریب ساده می‌تواند از طریق گرفتن میانگین از مقادیر هر کنش در وضعیت‌هایی که اطلاعات ناقص وجود دارد بدست آید.

۶-۹- تمرین

۶-۱ در این مساله با استفاده از مثال بازی تیک-تاک- تو مفاهیم اصلی بازی‌ها مرور می‌شود. X_n را تعداد سطرها، ستون‌ها یا قطرهایی که دقیقاً دارای n رخداد X هستند و هیچ رخداد Oای ندارند تعریف می‌کنیم. به طور مشابه، O_n تعداد سطرها، ستون‌ها و قطرهای شامل دقیقاً n تا O است. با این حساب تابع سودمندی به هر موقعیت با $X_3 = 1$ مقدار ۱+ و به هر موقعیت با $O_3 = 1$ مقدار ۱- را نسبت می‌دهد. دیگر موقعیت‌های پایانی مقدار سودمندی ۰ دارند. ما از یک تابع ارزیابی خطی که به صورت زیر تعریف می‌شود استفاده خواهیم کرد:

$$Eval = 3X_2 + X_1 - (3O_2 + O_1)$$

الف) تقریباً چند بازی ممکن برای تیک-تاک-تو وجود دارد؟

ب) یک درخت بازی کامل که با صفحه‌ی خالی شروع می‌شود و تا عمق ۲، (یعنی یک X و یک O روی صفحه) فرو می‌رود را با در نظر گرفتن تقارن نمایش دهید.

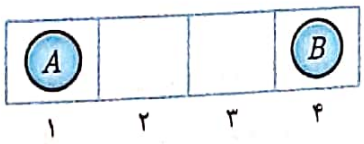
ج) بر روی درختتان تمامی وضعیت‌های سطح ۲ ارزیابی شده را مشخص کنید.

د) بر روی درختتان مقادیری که از سطوح پایین‌تر به سطوح بالاتر محاسبه می‌شوند را در سطح ۱ و ۰ با استفاده از الگوریتم minimax مشخص کنید و با استفاده از آن‌ها بهترین حرکت شروع را انتخاب کنید.

ه) اگر هرس آلفا- بتا به کار برده شود، در سطح ۲ گره‌هایی که نباید ارزیابی شوند را با کشیدن دایره مشخص کنید. با این فرض که گره‌ها به ترتیب بهینه برای هرس آلفا-بتا تولید می‌شوند.

۶-۲ این ادعا را ثابت کنید: در هر درخت بازی، سودمندی که MAX توسط الگوریتم تصمیم‌گیری minimax در مقابل بازی غیربهینه MIN بدست می‌آورد هچگاه کوچکتر از سودمندی آن در مقابل بازی بهینه MIN نخواهد بود. آیا می‌توانید درخت بازی را مثال بزنید، که در آن MAX با استفاده از یک استراتژی غیربهینه همچنان بهتر از بازی غیربهینه MIN عمل کند؟

۶-۳ به بازی دو نفره نشان داده شده در شکل ۶-۱۴ توجه کنید.



شکل ۶-۱۴ نقطه شروع یک بازی ساده. ابتدا بازیکن A حرکت می‌کند. به نوبت بازیکنان بازی می‌کنند. هر بازیکن می‌تواند مهره خود را در خانه خالی همسایه در جهت دلخواه قرار دهد. اگر حریف در خانه همسایه باشد، بازیکن می‌تواند از روی حریف پریده و به خانه خالی بعدی برود(برای مثال اگر A در خانه ۳ باشد و B در خانه ۲ باشد، آنگاه A می‌تواند به ۱ بازگشت کند). بازی زمانی خاتمه می‌یابد که یکی از بازیکنان به سمت مقابل صفحه بازی برسد. اگر بازیکن A ابتدا به خانه ۴ برسد آنگاه مقدار بازی برای A، ۱+ می‌شود. و اگر بازیکن B ابتدا به خانه ۱ آنگاه مقدار بازی برای A، ۱- خواهد بود.

الف) براساس مقررات زیر درخت بازی را رسم کنید:

هر حالت را به صورت (S_A, S_B) نمایش دهید که در آن S_A و S_B مکان‌های گرفته شده‌اند.

هر یک از حالت‌های پایانی را در یک خانه مربعی قرار دهید و مقدار بازی آن را درون دایره یادداشت کنید.

حالت‌های حلقه‌ای (حالت‌هایی که در مسیر ریشه ظاهر شده‌اند) را درون خانه مربعی دوخطی قرار دهید. از آنجایی که مشخص نیست چگونه به حالت‌های حلقه‌ای مقدار نسبت دهیم آن‌ها را با علامت ؟ مشخص کنید.

ب) حال هریک از گره‌ها را با مقدار minimax برگشتی آن مشخص کنید (در یک دایره). توضیح دهید که چگونه مقادیر ؟ را مدیریت می‌کنید.

ج) توضیح دهید چرا الگوریتم minimax استاندارد در درخت این بازی شکست می‌خورد، و به‌طور خلاصه توضیح دهید که چگونه می‌توان این اشکال را براساس پاسخ قسمت (ب) برطرف کرد؟ آیا الگوریتم اصلاح شده شما برای تمامی درخت‌هایی که دارای حلقه هستند پاسخ بهینه می‌دهد؟

د) این بازی که دارای ۴ خانه است می‌تواند در حالت کلی برای $n>2$ بسط داده شود. ثابت کنید اگر n زوج باشد آنگاه A برنده بازی است و اگر n فرد باشد A بازنده است.

۶-۴ برای بازی‌های زیر تولید کننده حرکت و تابع ارزیابی را پیاده‌سازی کنید. Kalah، Othell، Checkers، شطرنج. سپس یک عامل بازیکن که از آلفا- بتا استفاده می‌کند ساخته و آن را پیاده‌سازی کنید. تاثیر افزایش عمق جستجو، بهبود ترتیب حرکات و بهبود تابع ارزیابی را با یکدیگر مقایسه کنید. ضریب انشعاب موثرتان چقدر به حالت ایده‌آل ترتیب حرکات کامل، نزدیک است؟

۶-۵ یک اثبات صوری برای درستی هرس آلفا-بتا ارائه دهید. برای این منظور، شرایطی که در شکل ۶-۱۵ نشان داده شده۷ است را در نظر بگیرید. سوال این است که آیا باید گره‌ی n_j که یک گره‌ی ماکزیمم و از نسل n_1 است حذف شود؟ ایده‌ی اصلی این است که آن را هرس کنید اگر و تنها اگر بتوان نشان داد مقدار minimax، n_1 مستقل از مقدار n_j است.

الف) مقدار n_1 این گونه تعریف می‌شود:

$$n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2})$$

با نوشتن عبارتی مشابه برای مقدار n_2 ، عبارتی برای n_1 بر حسب n_j به دست آورید.

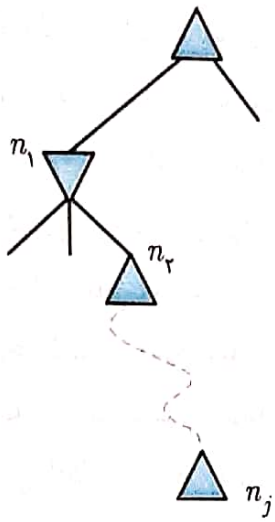
ب) فرض کنید l_i مینیمم (یا ماکزیمم) مقادیر گره‌های سمت چپی n_i در عمق i باشد. این‌ها گره‌هایی هستند که مقادیر ماکزیمم‌شان اکنون معلوم است. به طور مشابه فرض کنید r_i مینیمم (یا ماکزیمم) مقادیر گره‌های سمت راستی n_i در عمق i باشد. این گره‌ها هنوز پیمایش نشده‌اند. عبارتتان را برای n_1 دوباره بر حسب l_i و r_i بنویسید.

ج) اکنون عبارت را دوباره فرمول‌بندی کنید تا نشان دهید برای تغییر n_1 نباید n_j از حد مشخصی که از مقادیر l_i به دست می‌آید تجاوز کند.

د) این گام‌ها را برای حالتی که n_j یک گره کمینه است تکرار کنید.

۶-۶ ثابت کنید با تبدیل خطی مثبت مقادیر برگ‌ها(یعنی x تبدیل به $ax+b$ که در آن $a>0$ میشود)، انتخاب حرکت‌ها در درخت بدون تغییر باقی می‌ماند حتی اگر در آن گره‌های شانس نیز وجود داشته باشد.

۶-۷ روند زیر را برای انتخاب حرکت‌ها در بازی‌های با گره‌های شانسی در نظر بگیرید:
به تعداد مناسبی(مثلاً ۵۰) از دنباله‌ی اعداد تصادفی را تا عمقی مناسب (مثلاً ۸) تولید کنید.
با معلوم شدن مقادیر تاس‌ها، درخت بازی تبدیل به یک درخت بازی قطعی می‌شود. برای هر دنباله از اعداد تصادفی درخت بازی حاصل را با استفاده از آلفا- بتا حل کنید.
با استفاده از نتایج، مقدار هر حرکت را تخمین و بهترین آن را انتخاب کنید.
آیا این روند درست عمل می‌کند؟ چرا (نه)؟



شکل ۶-۱۵

۶-۸ یک محیط بازی چندنفره و بی‌درنگ را توصیف و پیاده‌سازی کنید که در آن زمان جزئی از حالات محیط باشد و به بازیکنان زمان‌های ثابتی داده شده باشد.

۶-۹ برای بازی‌های زیر حالت‌ها، تولیدکننده حرکت‌ها، توابع تست پایانی، توابع سودمندی، توابع ارزیابی را تعریف و پیاده‌سازی کنید: Monopoly، Scrabble، Bridge، Poker.

۶-۱۰ با دقت، اثر متقابل رخدادهای تصادفی و اطلاعات جزئی را در هر یک از بازی‌های تمرین ۶-۹ مورد توجه قرار دهید:
الف) برای کدامیک مدل استاندارد "minimax موردانتظار" مناسب است؟ الگوریتمی پیاده‌سازی و با ایجاد تغییرات مناسب برروی محیط بازی آن را برروی عامل بازی‌تان اجرا کنید.
ب) برای کدامیک طرحی که در تمرین ۶-۷ توضیح داده شد مناسب است؟
ج) درباره اینکه چگونه ممکن است با این واقعیت که در بعضی بازی‌ها، بازیکنان اطلاع یکسانی از وضعیت جاری نداشته باشند برخورد کنید، بحث کنید؟

۶-۱۱ در الگوریتم minimax فرض می‌شود که بازیکنان به نوبت بازی میکنند، اما در بازی‌های کارتی مثل bridge و whist برنده دست قبل در دست بعدی اول بازی می‌کند.
الف) الگوریتم را به گونه‌ای اصلاح کنید که برای این گونه بازی‌ها درست عمل کند. می‌توانید تابعی مثل WINNER(s) را در نظر بگیرید که بگوید کدام بازیکن دست قبلی را برده است.

۶-۱۲ برنامه‌ی Chinook checkers از پایگاه داده‌ی پایان بازی که مقادیر دقیق هر وضعیت در ۶ حرکت پایانی بازی را دارد استفاده‌ی گسترده‌ای می‌کند. چگونه چنین پایگاه داده‌ای می‌تواند به شکلی کارا تولید شود؟

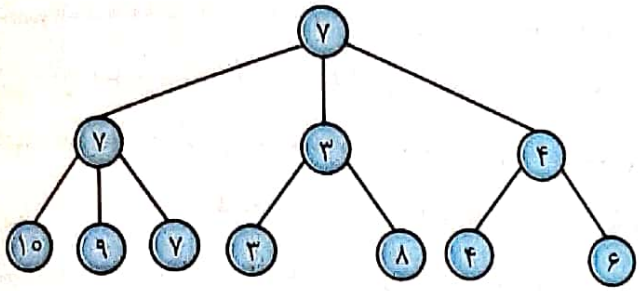
۶-۱۳ توضیح دهید روش استاندارد بازی کردن تا چه اندازه برای بازی‌هایی مثل تنیس، Pool و Croquet که در یک فضای حالت پیوسته‌ی فیزیکی اتفاق می‌افتند کاربرد دارد؟

۶-۱۴ تا کنون فرض کرده بودیم که قواعد هر بازی یک تابع سودمندی تعریف می‌کند که توسط جفت بازیکنان استفاده می‌شود و اینکه سودمندی x برای MAX به معنی سودمندی $-x$ برای MIN است. بازی‌های با این خصوصیت مجموع- صفر نامیده می‌شوند. توضیح دهید وقتی بازی‌های مجموع- غیرصفر- یعنی هر بازیکن تابع سودمندی خاص خودش را دارد- داریم چگونه الگوریتم‌های minimax و آلفا- بتا تغییر می‌کنند. می‌توانید فرض کنید هر بازیکن تابع سودمندی حریف را می‌داند.

۶-۱۵ فرض کنید برنامه شطرنجی دارید که می‌تواند در هر ثانیه یک میلیون گره را ارزیابی کند. برای نحوه نمایش فشرده بازی در حافظه جدول انتقال تصمیم‌گیری کنید. در جدولی با حافظه ۵۰۰ مگا بایت حدودا چند موجودیت قرار می‌دهید؟ آیا زمان جستجو ۳ دقیقه‌ای برای یک حرکت کافی است؟ برای یک ارزیابی در این زمان، چند جستجو در جدول صورت می‌گیرد؟فرض کنید جدول به اندازه‌ای بزرگ است که در حافظه جا نمی‌شود. در زمان معین برای پیگرد دیسک، چند ارزیابی می‌توان انجام داد؟

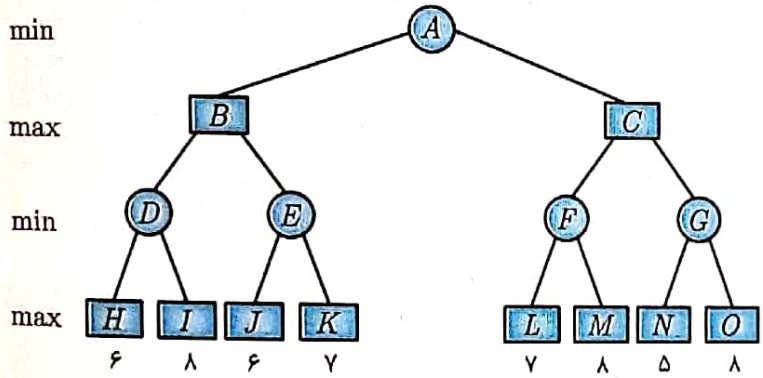
تست‌های طبقه‌بندی‌شده فصل ششم

۱- در درخت بازی زیر (i,j) نمایش ارزش عنصر jام از چپ به راست در سطح iام است. مثلا (۲، ۳) نمایش عنصر دوم از سطح سوم با ارزش ۹ می‌باشد چه عناصری شامل الفا- بتا پرونینگ می‌شوند؟ (کامپیوتر ۸۱)



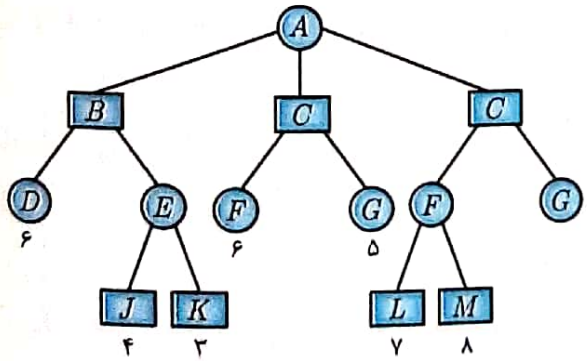
۱) (۳,۳)
۲) (۳,۵)(۳,۷)
۳) (۲,۳)(۲,۲)
۴) (۳,۶)(۳,۴)(۳,۲)(۳,۱)

۲- درخت مقابل در اثر جستجوی minimax ایجاد شده است. گره‌های دایره گره min وگره‌های مربع گره max هستند. اعداد زیر گره‌های برگ ارزش آنها را نشان می‌دهد. در صورت استفاده از روش آلفا و بتا کدام یک از گره‌های این درخت جستجو نخواهد شد؟ (کامپیوتر ۸۲)



۱) گره‌های K و M
۲) گره‌های K و M و O
۳) گره‌های K و G و O و N
۴) کلیه گره‌ها جستجو خواهند شد.

۳- اگر با استفاده از روش جستجوی Minimax درخت جستجوی مقابل پیمایش شود، با استفاده از روش هرس آلفا- بتا، کدام یک از گره‌های این درخت ملاقات نخواهد شد؟ (دوایر معرف گره‌ی min و مربع‌ها معرف گره max و اعداد کنار گره‌های برگ، معرف ارزش گره‌ها است) (کامپیوتر۸۳)

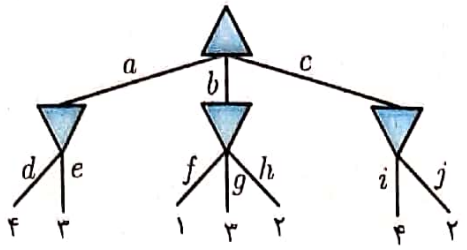


۱) {L,H,N}
۲) {L,N,J}
۳) {L,H,J}
۴) {H,N,J}

۴- بیشترین تأثیر افزودن عنصر شانس (مثل ریختن تاس) در بازی‌ها، بر روی درخت جستجوی تولید شده چیست؟

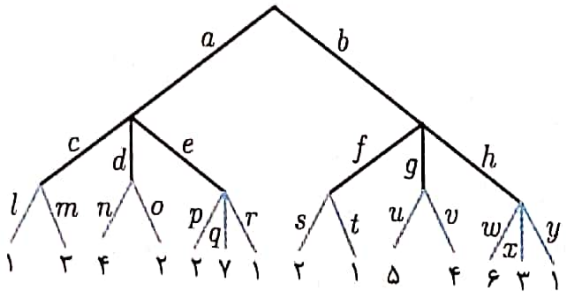
۱) هرس کردن شاخه‌ها مشکل‌تر می‌شود.
۲) روش‌هایی مثل minimax و آلفا ـ بتا نمی‌توانند با عنصر شانس کار کنند.
۳) در محاسبه‌ی تابع ارزیابی باید لبه‌های مرزی که بازتاب عنصر شانس هستند را اعمال نمود.
۴) برای هر حرکت بازیکن، سطح دیگری از گره‌ها تولید می‌شوند که احتمالات معرفی شده به‌وسیله عنصر شانس را دربرمی‌گیرند.

۵- در درخت زیر اعمال هرس آلفا- بتا همراه با ترتیب‌دهی (ordering) در هنگام استفاده از الگوریتم MiniMax منجر به حذف چه شاخه‌هایی خواهد شد؟ (حروف مربوط به شاخه‌ها هستند و روی شاخه‌ها ثابت می‌مانند و با جابه‌جا شدن اعداد، جابه‌جا نمی‌شوند.) (کامپیوتر ۸۵)



۱) h,g
۲) c,b
۳) j,h,g
۴) j,h,g,e

۶- در درخت بازی زیر اگر از هرس آلفا- بتا استفاده شود کدام شاخه‌ها حذف خواهند شد؟ (فرض می‌شود حذف هر شاخه غیر انتهایی به‌طور ضمنی حذف تمام زیر درخت تحت آن را به همراه دارد و ذکر شاخه‌های زیر درخت لازم نیست.) (کامپیوتر ۸۶)

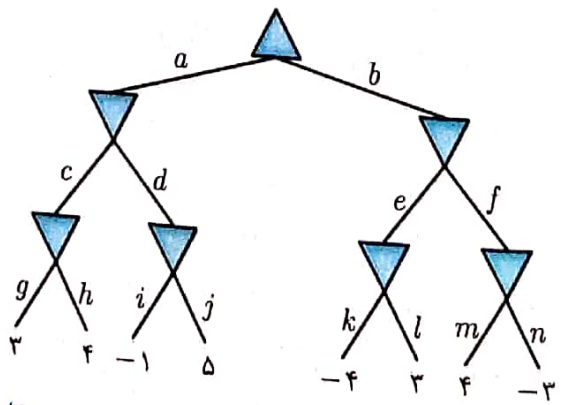


۱) e – g – h
۲) o – r – v – h
۳) o – q – r – v – x – y
۴) o – r – g – h

۷- اگر در یک بازی از الگوریتم MINIMAX استفاده شود، چه اتفاقی می‌افتد اگر بازیکن MIN در عمل گزینه‌ای را انتخاب کند که سودمندی بیش‌تری داشته باشد؟ (کامپیوتر ۸۷)

۱) سودمندی MIN بیشتر خواهد شد و MIN برنده خواهد شد.
۲) الگوریتم MINIMAX در این حالت قابل استفاده نیست و این الگوریتم با این فرض طراحی نشده است.
۳) درهر حالت سودمندی MAX کمتر از حالتی که MIN گزینه‌ای با سودمندی کمتر را انتخاب کند نیست.
۴) در هر حالت سودمندی MAX حداکثر برابر حالتی است که MIN گزینه‌ای با سودمندی کمتر را انتخاب کند.

۸- فرض کنید در یک بازی، هر بازیکن ابتدا یک سکه سالم پرتاب می‌کند، اگر شیر آمد شاخه سمت چپ و اگر خط آمد شاخه سمت راست درخت بازی مقابل را انجام می‌دهد. در درخت بازی داده شده به فرض اینکه امتیاز بازیکن‌ها در بازه [۵، ۵-] است، اگر هرس آلفا- بتا اجرا شود، کدام شاخه‌ها حذف خواهند شد؟ (کامپیوتر ۸۸)



۱) l-f
۲) l-n
۳) j-l-n
۴) هیچ هرسی رخ نخواهد داد.

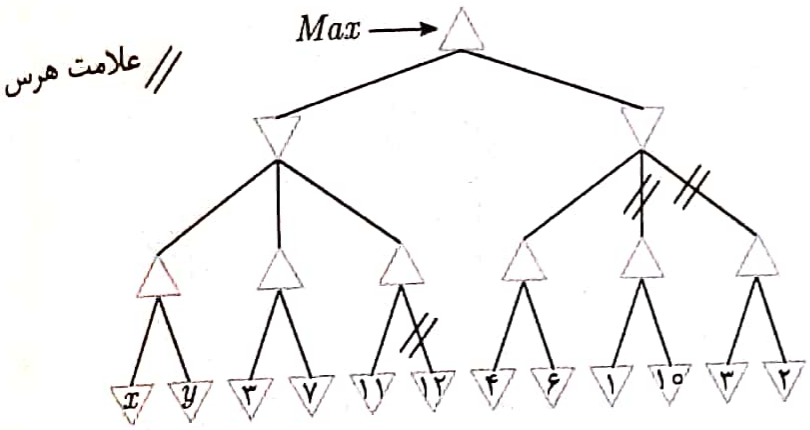
۹- در درخت بازی زیر کدام یک از گزینه‌ها محدوده مناسبی برای مقادیر مثبت x و y تعیین می‌کند به طوری‌که شاخه‌های علامت زده شده در هرس آلفا ـ بتا، هرس شوند؟

۱) به‌ازای هیچ مقداری از x و y شاخه‌های علامت‌زده شده حذف نخواهند شد.

۲) $\forall x, y \quad 0 < x \leq 3 \ , \ 0 < y \leq 3$

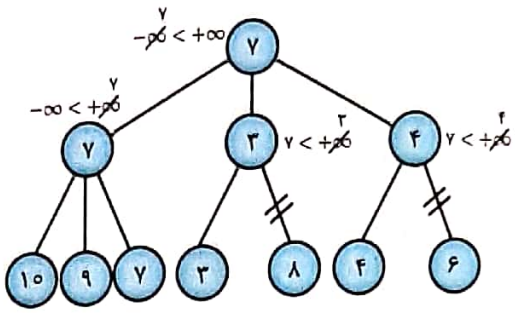
۳) $\forall x, y \quad 6 - x - y > 0$

۴) $\forall x, y \quad x < 5 \ , \ y > 7$

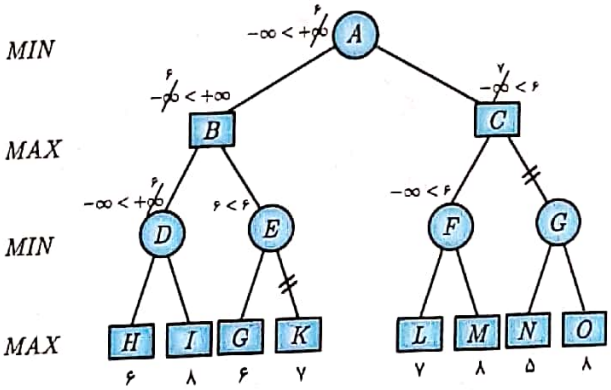


پاسخنامه تشریحی تست‌های فصل ششم

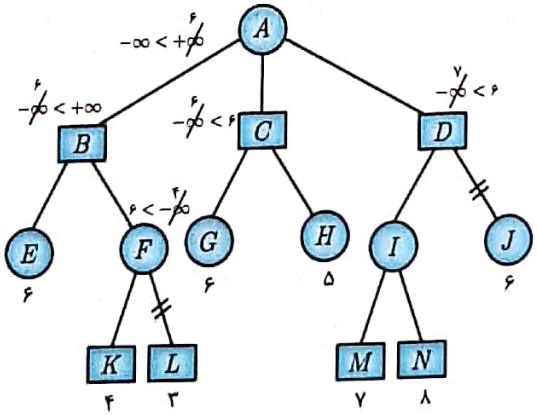
۱) گزینه ۲ درست است.



۲) گزینه ۳ درست است.



۳) گزینه ۳ درست است.

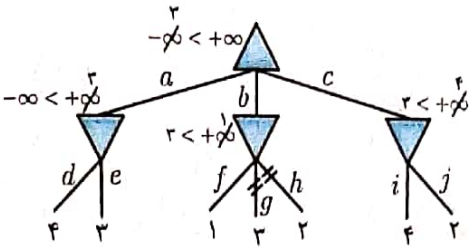


۴) گزینه ۱ و ۴ درست است.

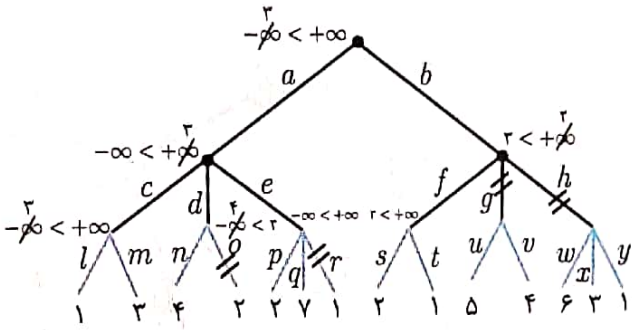
توضیح: هرس آلفا بتا به‌معنای آن است که بخش‌هایی از درخت حذف شود بدون آنکه نتیجه تغییر یابد. و عملاً گارانتی می‌شود که شاخه‌های حذف شده در نتیجه نهایی استفاده نمی‌شود. در حالت وجود گره شانس، عملاً utility هر میانگین وزن‌دار همه‌ی گره‌های زیرین است. که از ضرب میزان مقبولیت هر گره در احتمال آن و جمع آن برای همه‌ی گره‌های زیرین گره شانس به‌دست می‌آید. ولی در صورتی که میزان مقبولیت یک گره کم

شود، به معنای آن نیست که ممکن نیست استفاده شود، بلکه ممکن است با همان احتمال خیلی کم، نیز اتفاق بیافتد. بنابراین هیچ گارانتی در اینجا وجود ندارد و عملاً هرس آلفا ـ بتا مشکل‌تر خواهد بود. البته اگر برای عدد مقبولیت گره‌ها حداکثر و حداقل قائل شویم، این امکان وجود خواهد داشت. (تست ۵۶ سال ۸۸ گروه کامپیوتر رجوع شود) گزینه ۴ نیز درست است. گره‌های شانس در درخت بازی ایجاد می‌گردند.

۵) گزینه ۱ درست است.



۶) گزینه ۴ درست است.



۷) گزینه ۳ درست است.

می‌توان نشان داد که دراین حالت MAX نتیجه‌ی بهتری خواهد گرفت (تمرین شماره‌ی ۶-۲). اگرچه ممکن است استراتژی‌های دیگری در مقابل حریف‌های غیربهینه وجود داشته‌باشند که بهتر از استراتژی minimax عمل کنند، اما همه‌ی این استراتژی‌ها قطعاً در مقابل حریف‌های بهینه بد عمل می‌کنند.

۸) گزینه ۱ درست است.

۹) گزینه ۴ درست است.

فرض کنید که $x < y$ باشد و یعنی اینکه $max(x, y) = y$ باشد با اجرای هرس آلفا ـ بتا روی زیردرخت سمت چپ مشاهده می‌شود که هر مقداری که x و y داشته باشند. زیرشاخه ۱۲ هرس می‌شود. پس عملاً مقدار x و y در هرس شدن یا نشدن شاخه ۱۲ تأثیری ندارد. فقط چون نباید شاخه‌ی ۷ هرس شود، بایستی $\max(x, y) = y$ است از ۳ بیشتر باشد تا ۷ هرس نشود.

خروجی گره min (فرزند سمت چپ ریشه) برابر است با $\min(۷, y) = k$ که k می‌نامیم. حال چون در سمت راست، گره ۴ و ۶ هرس نشده است، پس حتماً $k < ۶$ است. بنابراین بایستی از بین گزینه‌ها آن را که شرط زیر را دارد انتخاب کنیم.

$$\left.\begin{array}{l} \phi < x < y \\ ۳ < y \\ ۶ < y \end{array}\right\} \Rightarrow$$

تنها گزینه‌ی «۴» است که این شرایط را امضا می‌کند

اما عوض کردن ترتیب انتخاب گره‌ها باعث تغییر در هرس آلفا- بتا می‌شود.

تست‌های تألیفی

۱- بازی زیر را در نظر بگیرید:

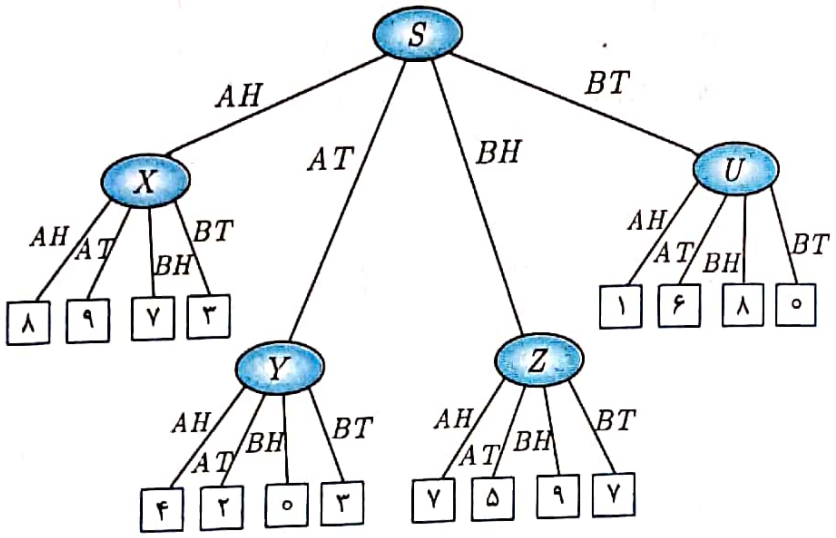
وقتی نوبت هر بازیکن باشد، بازیکن در ابتدا سکه مورد نظرش را انتخاب میکند. سکه A یا سکه B احتمال ظاهر شدن شیر (H) و خط (T) در دو سکه در زیر مشخص است.

سکه A: H ٪۷۵ ، T ٪۲۵

سکه B: H ٪۱۰ ، T ٪۹۰

و با توجه به اینکه آن سکه H یا T بیاید، حرکتش را انتخاب میکند.

درخت بازی زیر را در نظر بگیرید. فرض کنید ریشه MAX است. ریشه چه سکه ای را انتخاب کند، برایش بهتر است؟



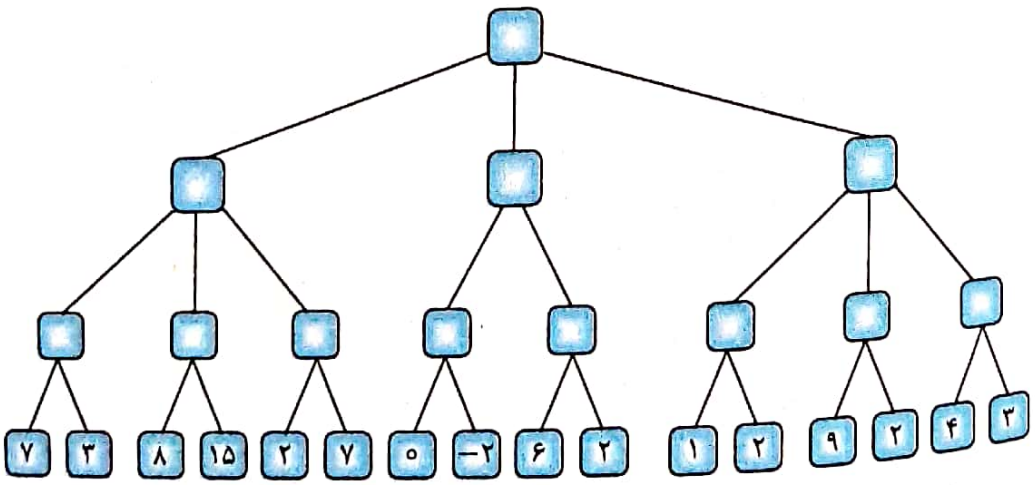
۱) A

۲) B

۳) در مسائلی که با شانس است، قابلیت انتخاب وجود ندارد.

۴) هر دو انتخاب به یک اندازه می‌تواند خوب باشد.

۲- درخت بازی زیر را در نظر بگیرید. در هرس آلفا-بتا مشخص کنید تعداد کل گره‌های هرس شده چه تعداد است. گره ریشه MAX است.



۱) ۱

۲) ۱۰

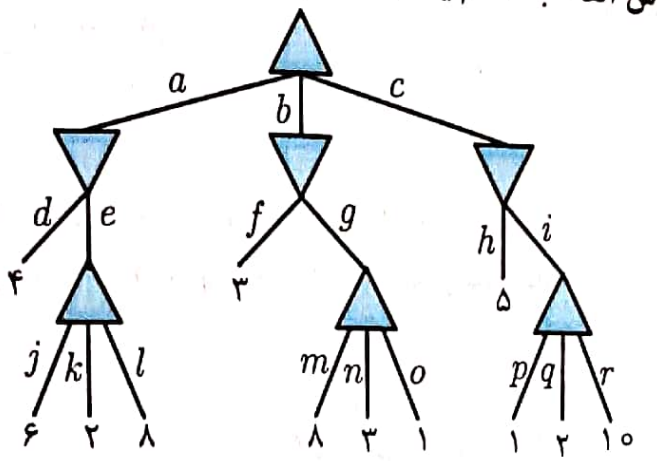
۳) ۱۱

۴) هیچکدام

۳- در سوال game tree مساله قبلی، اگر امکان آن باشد که ترتیب برگهایی که گره پدر آنها یکسان است را جابجا کنید، حداکثر تعداد گره های هرس شده در همه‌ی جابجایی های ممکن چه تعداد است:

۱) ۱۰ ۲) ۱۱ ۳) ۱۲ ۴) ۲۰

۴- پس از اجرای الگوریتم هرس آلفا – بتا کدام یک از یال‌های درخت Min-Max زیر هرس می‌شوند؟



۱) $q,r,g,,m,n,o,l$ ۲) k,l,g,m,n,o,r ۳) k,l,n,o,r ۴) k,l,g,m,n,o

پاسخ تشریحی تست‌های تألیفی

۱) برخلاف مسائل مربوط به بازیهای با شانس، سوال طوری نیست که درخت دارای chance node باشد. در واقع در این مسئله بازیکن فقط حق انتخاب سکه را دارد و اینکه H باشد یا T به شانس بستگی دارد.

در صورت انتخاب سکه A توسط X (X یک گره از درخت بازی است)، بهره وری بصورت U(X,A) نمایش داده می‌شود که برابر است با:

$U(X,A) = ۸*۰.۷۵ + ۹*۰.۲۵ = ۸.۲۵$

$U(X,B) = ۷*۰.۱ + ۳*۰.۹ = ۳.۴$

چون X یک گره min است.

$U(X) = \min(U(X,A),U(X,B)) = ۳.۴$

$U(Y,A) = ۴*۰.۷۵ + ۲*۰.۲۵ = ۳.۵$

$U(Y,B) = ۰*۰.۱ + ۳*۰.۹ = ۲.۷$

$U(Y) = \min(U(Y,A),U(Y,B)) = ۲.۷$

$U(Z,A) = ۷*۰.۷۵ + ۵*۰.۲۵ = ۳$

$U(Z,B) = ۹*۰.۱ + ۷*۰.۹ = ۷.۲$

$U(Z) = \min(U(Z,A),U(Z,B)) = ۳$

$U(U,A) = ۱*۰.۷۵ + ۶*۰.۲۵ = ۲.۲۵$

$U(U,B) = ۸*۰.۱ + ۰*۰.۹ = ۰.۸$

$U(U) = \min(U(U,A),U(U,B)) = ۰.۸$

در مرحله بعد، گره S بایستی بین A و B یکی را انتخاب کند. که در آنصورت

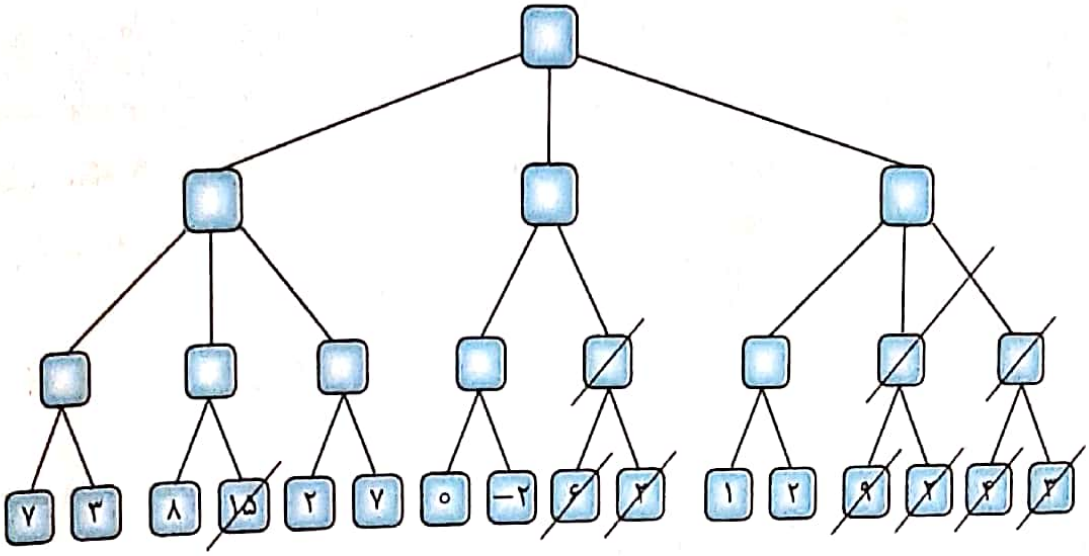
$U(S,A) = U(X)*۰.۷۵ + U(Y) * ۰.۲۵ = ۲.۹۲۵$

$U(S,B) = U(Z)*۰.۱ + U(U) * ۰.۹ = ۱.۰۲$

چون S یک گره max است. بنابراین S بایستی A را انتخاب کند.

۲) گزینه ۲ درست است.

با پیمایش درخت شاخه هایی که هرس می‌شوند در شکل زیر رسم شده است. که جمعا ۱۰ گره هرس می‌شود.



۳) گزینه ۲ درست است.

بهترین حالت هرس موقعی اتفاق می‌افتد که در گره‌ها max اولین فرزندش بزرگترین مقدار بین کل فرزندان آن گره را داشته باشد و بر عکس، در گره‌های min اولین فرزندش کوچکترین مقدار بین کل فرزندان آن گره را داشته باشد. در این مسأله برگها طوری چیده شده‌اند که به جز یک مورد ، امکان افزودن تعداد هرسها وجود ندارد. اگر برگهای شماره ۶ و ۵ (ازسمت چپ) با هزینه ۲و ۷ را جابجا کنیم ، بطوریکه پدرش (که گره max است)، ابتدا گره با ارزشتر را ببیند، آنگاه گره دوم، (گره با هزینه ۲) هرس می‌شود و تعداد هرسها ، به ۱۱ می-رسد.

۴) گزینه ۴ درست است.

با توجه به انکه گره ماکزیمم d را می‌بینید (با مقدار ۴) ، پس از رویت j (با مقدار ۶) عملا بقیه گره‌ها (یعنی l,k) مفید نیستند و $k,1$ هرس میشود. همچنین با اتمام شاخه a مقدار ۴ توسط ریشه دیده میشود و بنابراین با دیدن f (مقدار ۳) عملا هیچ مقداری از بقیه (یعنی g) مفید نخواهد بود و کل شاخه g هرس می-شود.