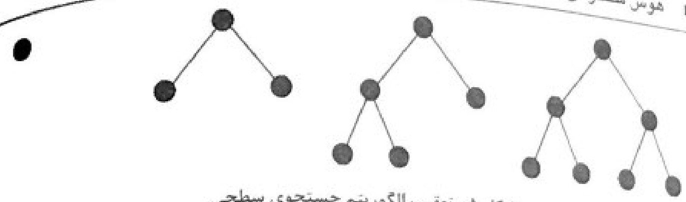


شکل ۵- تعقیب الگوریتم جستجوی سطحی

تاکنون، اخبار در مورد جستجوی سطحی، خوب بوده است.برای اینکه بفهمیم که چرا همیشه این استراتژی مطلوب نیست، بایستی میزان زمان و حافظه را در نظر بگیریم. برای این امر، ما باید یک فضای حالت فرضی را در نظر بگیریم که در آن هر حالت می‌توانـد گسترش داده شـود تـا بـه b حالت جدید برسد می‌گوییم که فاکتور انشعاب (branching factor) ازاین حالات (واز درخت جستجو) b است.

ریشه درخت جستجو b گره در اولین سطح تولید می‌کند. هر کدام گره‌های b بیشتری تولید می‌کنند و $b^2$ گره در سطح دوم خواهیم داشت. هر کدام از این‌ها، گره‌های b بیشتری تولید می‌کنند تا در سطح سوم به گره‌های $b^3$ برسد و این کار تا انتها ادامه پیدا می‌کند. حال تصور کنید که راه‌حل بـرای ایـن مسئله طول مسیری از d دارد. سـپس حـداکثر تعـداد گـره‌هـای بـسط داده شـده قبـل از پیداشـدن راه‌حـل $1+b+b^2+b^3+...+b^d$ است.

فرمول بالا حداکثر تعداد را نشان می‌دهد، اما راه حل در هـر نقطـه از سـطح d ام مـی‌توانـد پیدا شود در بهترین حالت تعداد، عدد کوچکتری را نشان می‌دهد. آنهایی که آنالیز پیچیدگی انجام می‌دهند، زمانی که با یک مرتبه زمانی نمایی مانند $O(b^d)$ برخورد می‌کنند عصبانی می‌شوند.

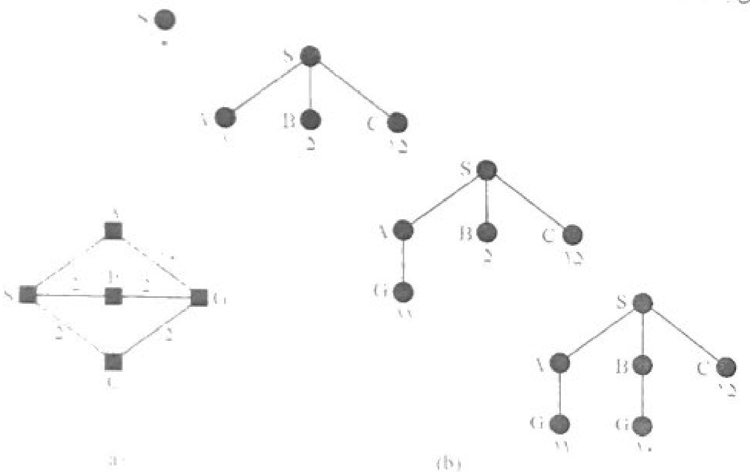
حافظه درخواست شده مسئله جدی‌تری برای جستجوی سطحی نسبت به زمان اجرای آن است. برای مثال بیشتر مردم حوصله انتظار ۱۸ دقیقه‌ای برای کامل شدن جستجو در عمـق ۶ را دارنـد، امـا خیلی از آنها حافظه‌ای در حدود ۱۱۱ مگابایت را برای این امر در اختیار ندارند. و اگر چه ۳۱ ساعت زمان زیاد طولانی برای انتظار حل مسئله در عمق ۸ نیست، اما تعداد انگشت‌شماری بـه ۱۱ گیگـا بایـت حافظـه‌ای مـورد نیاز، دسترسی دارند. خوشبختانه استراتژیهای جستجوی دیگری وجود دارند که نیاز به حافظه زیاد ندارند.

نکته دیگر آنست که زمان مورد نیاز هنوز عامل مهمی است. اگر مـسئله شـما دارای عمـق ۱۲ بـرای پاسخ باشد آنگاه می‌توان ثابت کرد ۳۵ سال زمان برای جستجوی غیر آگاهانه جهت حـصول پاسخ لازم است. البته اگر شرایط کنونی تا ۱۰ سال دیگر برقرار باشد،شما قادر به خرید کامپیوتری با هزینه کنونی خواهید بود که ۱۰۰ برابر سریعتر است. حتی با آن کامپیوتر، به ۱۲۸ روز زمان برای حل مسئله با عمق ۱۲ پاسخ نیاز دارید و همینطور ۳۵ سال برای پاسخ با عمق ۱۴. علاوه بر آن ، هـیچ روشـی جـستجوی غیـر آگاهانه دیگری برای نتایج بهتر وجود نداردو پس در کل، مسائل جستجوی با پیچیدگی نمایی حتی برای نمونه‌هـای کوچک نیز غیر قابل حل هستند.



## جستجوی هزینه یکسان

جستجوی سطحی کم‌عمق‌ترین حالت هدف را پیدا می‌کند، اما همیشه کم هزینه ترین حالت برای یک تابع هزینه مسیر عمومی نیست. جستجوی با هزینه یکسان (Uniform cost search) استراتژی جستجوی سطحی را توسط بسط کم هزینه‌ترین گره (که توسط تابع هزینه مسیر g(n) اندازه‌گیری شده است) به کم عمق‌ترین گره، بهبود می‌بخشد. آسان است ببینیم که جستجوی سطحی همان جستجوی با هزینه یکسان با $g(n) = DEPTH(n)$ است.



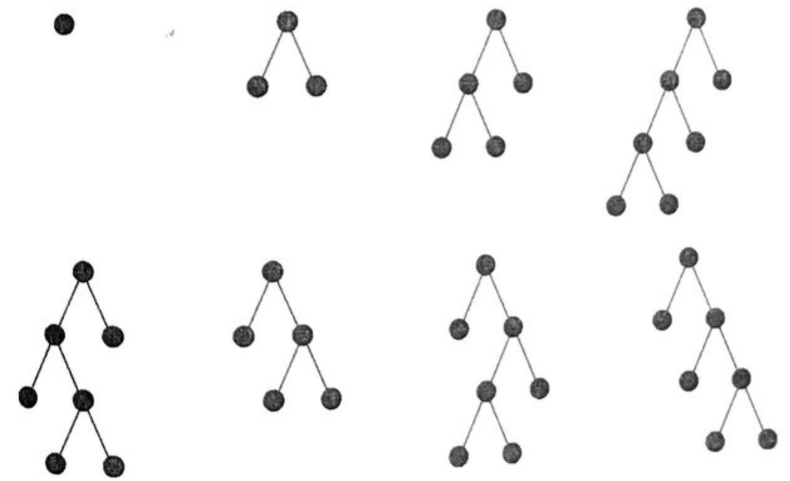

شکل ۶ مثالی از جستجوی با هزینه یکسان

زمانی که شرایط عمومی برقرار باشد، اولین راه حل پاسخ، ضمانت می‌کند که ارزانترین راه نیز باشد. زیرا اگر مسیر ارزانتری وجود داشته باشد که راه‌حل نیز باشد، زودتر بسط داده شده است و از این رو در ابتدا پیدا شده است. اگر در عمل نگاهی به استراتژی بیندازیم به تعریف ما کمک خواهد کرد. مسئله مسیریابی در شکل زیر را در نظر بگیرید. مشکل رسیدن از نقطه S به G است، و هزینه هر عملگر مشخص شده است. ابتدا ، استراتژی حالت اولیه را بسط می‌دهد. که منجر به پیدایش مسیرهای عملگر A، B، C می‌شود. چون هزینه‌های مسیر به A کمتر از بقیه است، در مرحله بعد، A قبل از بقیه بسط داده می‌شود و مسیرهای SAG بوجود میاید که در حقیقت همان راه‌حل است ولی بهینه نیست بهرحال الگوریتم این راه حل را به عنوان حل نهایی قبول نمی‌کند زیرا هزینه‌ای معادل ۱۱ دارد و از این رو در صف زیر مسیر SB که هزینه‌ای معادل ۵ دارد قرار می‌گیرد. شاید به نظر خجالت‌آور باشد که چرا الگوریتم راه‌حلی را پیدا کرد که می‌بایست در صف زیر بقیه مدفون شود اما این کار برای یافتن راه‌حل بهینه لازم است. مرحله بعد بسط SB است، که SBG تولید می‌شود که اکنون کم هزینه ترین مسیر باقی مانده در صف است. سارا این هدف چک شده است و به عنوان راه‌حل برگردانده می‌شود.

جستجو با هزینه یکسان، کم‌هزینه‌ترین راه‌حل را پیدا می‌کند: هزینه مسیر نباید با ادامه مسیر کاهش پیدا کند. محدودیت غیر کاهشی بودن هزینه مسیر، این حس را به وجود می‌آورد که هزینه مسیر یک گره، در اصل مجموع هزینه عملگرهایی است که مسیر رامی سازند. اگر هر عملگر هزینه غیر منفی داشته باشد سپس هزینه یک مسیر با ادامه مسیر، کاهش پیدا نخواهد کرد و جستجو با هزینه یکسان می‌تواند کم هزینه‌ترین مسیر را بدون کندوکاو در تمام درخت جستجو، پیدا کند. اما اگر بعضی از عملگرها، هزینه منفی داشته باشند، یک جستجوی خسته کننده از تمام گره‌ها باید صورت گیرد تا راه‌حل بهینه پیدا شود، زیرا ما هرگز نمی‌دانیم که چه زمانی یک مسیر، بدون توجه به طول و هزینه، به مرحله‌ای با هزینه منفی بالا وارد می‌شود، از این رو به بهترین مسیر تبدیل می‌شود.

## جستجوی عمقی (Depth – first –Search)

جستجوی عمقی همیشه یکی از گره‌ها را در پایین‌ترین سطح درخت، بسط می‌دهد. فقط زمانی که جستجو به یک بن بست می‌رسد (یک گره غیر هدف بدون امکان بسط)، برگشت داده می‌شود و به سراغ گره‌هایی در سطوح کم عمق تر می‌رود. این استراتژی با صفی می‌تواند پیاده‌سازی شود که همیشه حالات تولید شده جدید را در جلوی صف قرار می‌دهد. (صف اولویت) زیرا گره بسط داده شده، عمیق‌ترین گره بوده، بچه‌های آن حتی عمیق‌تر خواهند بود و بنابراین اکنون عمیق ترین هستند. پیشرفت این جستجو در شکل ۷ نمایش داده شده است.



شکل۷- جستجوی عمقی

جستجوی عمقی نیاز به حافظه نسبتاً کمی دارد. همان‌طور که شکل نشان می‌دهد، این جستجو فقط نیاز به ذخیره مسیر واحدی از ریشه به یک گره برگی دارد، باضافه گره‌های باقی ماندهٔ بسط داده نشده که برای



هر گره روی مسیر وجود دارد. برای یک فضای حالت با فاکتور انشعاب $b$ و حداکثر عمق $m$، جستجوی عمقی فضایی فقط به اندازهٔ $bm$ گره را درخواست می‌کند، برخلاف مقدار $b^d$ که در جستجوی سطحی مورد نیاز بود جایی که عمق‌ترین هدف درعمق $d$ است.

پیچیدگی زمانی برای جستجوی عمقی $O(b^m)$ است. برای مسائلی که راه‌حلهای زیادی دارند، جستجوی عمقی سریعتر از جستجوی سطحی عمل می‌کند، زیرا شانس خوبی برای یافتن راه‌حل بعد از بررسی فقط یک قسمت کوچک از کل فضا را دارد. جستجوی عمقی در بدترین حالت دارای پیچیدگی زمانی $O(b^m)$ است.

یکی از مضرات جستجوی عمقی آنست که در یک مسیر اشتباه هنگام پایین رفتن، گیر می‌کند. مسائل زیادی دارای درختهای عمیق و نامحدودی هستند، بنابراین جستجوی عمقی هرگز قادر نخواهد بود که از یک انتخاب ناموفق در یکی از گره‌های نزدیک بالای درخت، جان سالم بدر برد. جستجو همیشه به سمت پایین ادامه خواهد یافت بدون اینکه به طرف بالا برگردد، حتی زمانی که راه‌حل کوتاه‌تری نیز وجود داشته باشد. از این رو در این مسائل نیز جستجوی عمقی در یک حلقه بی‌پایانی خواهد افتاد و هرگز راه‌حلی را پیدا نخواهد کرد، یا بالاخره ممکن است راه‌حلی پیدا کند که طولانی‌تر از راه‌حل بهینه باشد. این بدان معناست که جستجوی عمقی نه کامل و نه بهینه است. به همین علت، جستجوی عمقی باید از درختهای جستجوی با عمق نامحدود یا بزرگ اجتناب ورزد. معمولاً جستجوی عمقی بصورت فراخوانی بازگشتی و بکمک پشته سیستم پیاده‌سازی می‌شود.

## جستجوی عمقی محدود شده (Depth – limited search)

این جستجو، از به دام‌افتادن جستجوی عمقی توسط به کارگیری یک برش روی عمیق‌ترین جای مسیر، جلوگیری می‌کند. این برش می‌تواند توسط الگوریتم جستجوی عمقی محدود شده، یا توسط استفاده از الگوریتم جستجوی عمومی با عملگرهایی که ردپای عمق را نگه می‌دارند، پیاده‌سازی شود. با این مجموعه عملگر جدید، ما در یافتن راه‌حل اگر وجود داشته باشد، ضمانت پیدا کرده‌ایم، اما هنوز موفق به یافتن کوتاهترین راه نشده‌ایم. جستجوی عمقی محدود شده کامل است، اما بهینه نیست. اگر ما محدوده عمقی را انتخاب کنیم که خیلی کوچک باشد، جستجوی عمقی محدوده عمقی شده، مشابه جستجوی عمقی است. این جستجو پیچیدگی زمانی $O(b^L)$ و فضای $O(bL)$ را خواهد داشت، که $L$ محدودهٔ عمق است.

## جستجوی عمیق‌کننده تکراری (Iterative deepening Search)

قسمت دشوار جستجوی عمقی محدود شده، انتخاب یک محدودهٔ خوب است. در روش جستجوی محدود شده یافتن مقدار مناسب $L$ درکارائی الگوریتم تأثیر بسزایی دارد. اگر فاصله دورترین گره از گره آغازین را داشته باشیم، می‌توان تخمین بسیار مناسبی را یافت. این مقدار، به عنوان قطر (diameter) فضای حالت شناخته می‌شود، و محدودهٔ عمق بهتری را می‌دهد، که ما را به سوی جستجوی کاراتری سوق می‌دهد. بهرحال، برای بیشتر مسائل، محدودهٔ عمقی مناسب را تا زمانی که مسئله حل نشده است، نمی‌شناسیم.

جستجوی عمیق‌کنندهٔ تکراری استراتژی است که نظریه انتخاب بهترین محدودهٔ عمقی توسط امتحان تمام محدودهٔ مسیرهای ممکن را یادآوری می‌کند: اول عمق ۰، سپس عمق ۱،سپس عمق ۲ و الی آخر. الگوریتم در شکل ۸ نشان داده شده است.

جستجوی عمیق‌کننده تکراری، مزایای جستجوی سطحی و عمقی را با هم ترکیب می‌کند. این جستجو مانند جستجوی سطحی کامل و بهینه است، اما فقط مزیت درخواست حافظه اندک را از جستجوی عمقی دارد. مرتب بسط حالات مشابه جستجوی سطحی است، به جز اینکه بعضی حالات چند بار بسط داده می‌شوند. شکل ۸ اولین چهار تکرار ITRATIVE-DEEPENING-SEARCH را روی درخت جستجوی دو دویی نشان می‌دهد.

جستجوی عمیق کننده تلف کننده وقت به نظر می‌آید، زیرا حالات زیادی را چند بار بسط می‌دهد. برای بیشتر مسائل، بهرحال، سرریزی این بسطهای تکراری واقعاً کوچک است.

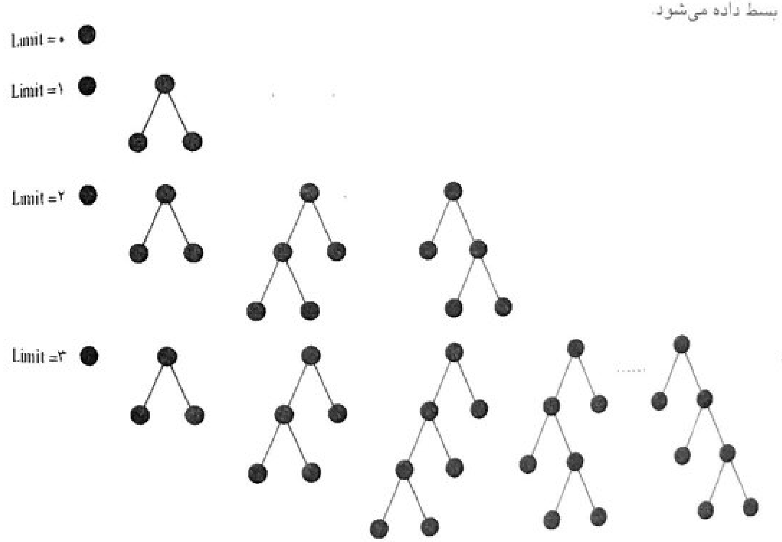
زیرا در یک درخت جستجوی نمایی، تقریباً تمام گره‌ها در سطح پایین هستند. بنابراین، موردی ندارد که سطوح بالایی چندین مرتبه بسط داده شوند. تعداد بسطها در یک جستجوی عمقی محدود شده با عمق d و فاکتور انشعاب b به قرار زیراست:

$$۱+b+b^{۲}+\ldots+b^{d-۲}+b^{d-۱}+b^{d}$$

برای $b=۱۰$ و $d=۵$، نتیجه برابر است با:

$$۱+۱۰+۱۰۰+۱۰۰۰+۱۰۰۰۰+۱۰۰۰۰۰=۱۱۱۱۱۱$$

در جستجوی عمیق‌کننده تکراری، گره‌های سطوح پایینی یکبار بسط داده می‌شوند، آنهایی که یک سطح بالاتر قرار دارند دوبار بسط داده می‌شوند و الی آخر، تا به ریشه درخت جستجو برسد، که d+1 بار بسط داده می‌شود.




شکل ۸- جستجوی عمیق شونده تکراری در چهار مرحله اول آن



بنابراین مجموع دفعات بسط در این جستجو عبارتست از:

$$(d+۱)۱+(d)b+(d-۱)b^{۲}+\ldots+۳b^{d-۲}+۲b^{d-۱}+۱b^{d}$$

دوباره برای $b=۱۰$ و $d=۵$ تعداد برابر است با:

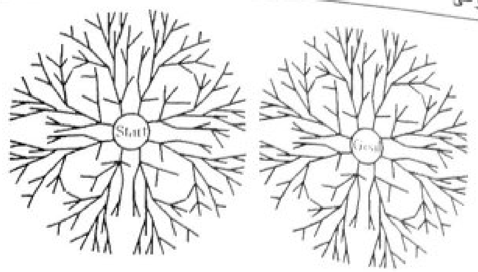
$$۶+۵۰+۴۴۰۰+۳۰۰۰+۲۰۰۰۰+۱۰۰۰۰۰=۱۲۳۴۵۶$$

در جستجوی عمیق کننده تکراری تمام مسیر از عمق ۱ تا عمق d فقط در حدود ۱۱٪ گره بیشتر از جستجوی سطحی یا جستجوی عمقی محدود شده تا عمق d، بسط می‌دهند (زمانی که $b=۱۰$ است.) فاکتور انشعاب بالاتر سرریزی کمتری در حالات بسط داده شده تکراری را موجب می‌شود. اما حتی زمانی که فاکتور انشعاب ۲ است، این جستجو فقط ۲ برابر زمان می‌گیرد. این بدان معناست که پیچیدگی زمانی این جستجو هنوز $O(b^{d})$است، و پیچیدگی فضا $O(bd)$است. در حالت کلی، عمیق کنندهٔ تکراری روش جستجوی برتری است، زمانی که فضای جستجوی بزرگی وجود دارد و عمق راه‌حل نیز مجهول است.

## جستجوی دو طرفه (Bidirectional search)

ایده جستجوی دو طرفه در واقع شبیه‌سازی جستجویی به سمت جلو (forward) از حالت اولیه و به سمت عقب (backward) از هدف است و زمانی که این دو جستجو به هم برسند، متوقف می‌شود (شکل ۹). برای مسائلی که فاکتور انشعاب در دوجهت b است، جستجوی دو طرفه تفاوت بزرگی را ایجاد می‌کند. اگر ما فرض بگیریم که راه‌حلی از عمق d وجود دارد، این راه‌حل در مراحل $O(۲b^{d/۲})=O(b^{d/۲})$ پیدا خواهد شد، زیرا جستجوی به سمت عقب و جلو، می‌بایست فقط نیمی از راه را طی کند. برای اثبات این امر: برای $b=۱۰$ و $d=۶$، جستجوی سطحی ۱،۱۱۱،۱۱۱ گره تولید می‌کند، در حالی که جستحوی دو طرفه زمانی که هر جهت در عمق ۳ است موفق می‌شود و در این حالت ۲۲۲۲ گره تولید می‌کند موارد زیادی قبل از اینکه الگوریتم بتواند پیاده‌سازی شود، نیاز به پاسخگویی دارند.

- سؤال اصلی این است که، جستجو از سمت هدف به چه معنی ست؟ ما قبل‌های (predecessors) یک گره n را گره‌هایی در نظر می‌گیریم که n ما بعد (successor) آنها باشد. جستجو به سمت عقب بدین معناست که تولید ما قبل‌ها از گرهٔ هدف آغاز شود.
- زمانی که تمام عملگرها، قابل وارونه‌شدن باشند، مجموعه ما قبل‌ها و ما بعدها یکسان هستند: برای بعضی از مسائل، بهرحال، محاسبه والدها ممکن است بسیار مشکل باشد.
- چه کار می‌توان کرد زمانی که هدفهای متفاوتی وجود داشته باشد؟ اگر لیست صریحی از حالتهای هدف وجود داشته باشد، مانند دو حالت هدف در شکل ۲-۳، می‌توانیم یک تابع ما قبل برای مجموعه حالت تقاضا کنیم در حالی که تابع ما بعد یا (جانشین) در جستجوی مسائل چند وضعیت به کار می‌رود. اگر ما فقط تعریفی از مجموعه داشته باشیم، ممکن خواهد بود که شرحی از «مجموعه حالاتی که مجموعه هدف را تولید می‌کنند» حاصل شود. اما این یک عمل زیرکانه خواهد بود برای مثال، چه حالتی، والدهای هدف کیش و مات در شطرنج هستند؟




شکل ۹- جستجوی دو طرفه

- باید یک راه موثر برای کنترل هر گره جدید وجود داشته باشد تا متوجه شویم که **آیا این گره قبلاً در** درخت جستجو توسط جستجوی طرف دیگر، ظاهر شده است یا خیر
- نیاز داریم که تصمیم بگیریم که چه نوع جستجویی در هر نیمه قصد انجام دارد. **برای مثال شکل ۹ دو** جستجوی سطحی را نشان می‌دهد. آیا این جستجو بهترین انتخاب است؟

شکل پیچیدگی $O(b^{d/2})$ فرض می‌کند که فرآیند آزمون برای اشتراک دو مجموعه فرزند **در زمان ثابتی** می‌تواند انجام گیرد. (بدین معناست که مستقل از تعداد حالتهاست). این امر اغلب **با استفاده از جدول** پراکندگی (hash) صورت می‌گیرد. برای اینکه دو جستجو بالاخره همدیگر را ملاقات کنند، **گره‌های حداقل** یکی از جستجوها باید در حافظه ذخیره شوند (مانند جستجوی سطحی) این بدین **معناست که پیچیدگی** فضا در جستجوی دو طرفه غیر آگاهانه $O(b^{d/2})$ است.

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bridirectional (if applicable) |
|---|---|---|---|---|---|---|
| Time | $b^d$ | $b^d$ | $b^m$ | $b^l$ | $b^d$ | $b^{d/2}$ |
| Space | $b^d$ | $b^d$ | bm | $b^l$ | bd | $b^{d/2}$ |
| Optimal? | Yes | Yes | No | No | Yes | Yes |
| Complete? | Yes | Yes | No | Yes, if $l \geq d$ | Yes | Yes |

ارزیابی استراتژیهای جستجو.b فاکتور انشعاب، d ، عمل پاسخ m ماکزیمم عمق درخت جستجو: **l محدودیت عمق است.**

## اجتناب از حالات تکراری

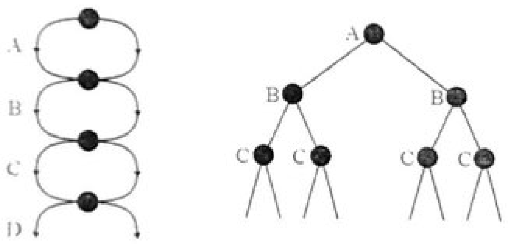
تا اینجا، یکی از پیچیدگیهای مهم پردازش جستجو را حذف کردیم: امکان اتلاف زمان **توسط بسط حالاتی** که قبلاً در مسیرهای دیگر بسط داده شده‌اند. برای بعضی مسائل، این امکان وجود ندارد؛ **هر حالت فقط** در یک مسیر بوجود می‌آید. مدل سازی موثر مسئله ۸ وزیر، در **قسمتهای بزرگ نیز کارآمد است به همین** علت، هر حالت فقط می‌تواند از یک مسیر مشتق شود.

برای مسائل زیادی، حالات تکراری غیر قابل اجتناب هستند این **شامل تمام مسائلی می‌شود که عملگرها** قابل وارونه شدن باشند، مانند مسائل مسیریابی و کشیش‌ها و **آدمخوارها. درخت‌های جستجو برای این** مسائل نامحدود هستند. اما اگر تعدادی از حالات را حذف کنیم، **می‌توانیم درخت جستجو را از پایین برش**



داده و آن را به اندازه محدود تبدیل کنیم، و فقط آن قسمتی از درخت که گراف فضای حالت را تشکیل می‌دهد، باقی بماند.

حتی زمانی که درخت محدود است، اجتناب از وقوع حالات تکراری می‌تواند موجب کاهش نمایی در هزینه جستجو شود. مثال کلاسیک آن در شکل ۱۰ نشان داده شده است. فضا فقط شامل $m+1$ حالت می‌شود. جایی که m حداکثر عمق است. به علت اینکه درخت هر مسیر ممکنی را در فضا شامل می‌شود، $2^m$ شاخه دارد.




شکل ۱۰ - حالات تکراری

سه راه برای حل مشکل حالات تکراری برای مقابله با افزایش مرتبه و سرریزی فشار کار کامپیوتر وجود دارد.

- به حالتی که هم‌اکنون از آن آمده‌اید، برنگردید. داشتن تابع بسط (یا مجموعه عملگرها) از تولید مابعدهایی که مشابه حالتی هستند که در آنجا نیز والدین این گره‌ها وجود دارند، جلوگیری می‌کند
- از اینجا مسیرهای دوار بپرهیزید. داشتن تابع بسط (یا مجموعه عملگرها) ازتولید مابعدهای یک گره که مشابه اجداد آن گره است، جلوگیری می‌کند.
- حالتی را که قبلاً تولید شده است، مجدداً تولید نکنید. این مسئله باعث می‌شود که هرحالت در حافظه نگه‌داری شود، پیچیدگی فضای $O(b^d)$ داشته باشد. بهتر است که به $O(s)$ توجه کنید که s تعداد کل حالات در فضای حالت ورودی است.

برای پیاده سازی آخرین امکان، الگوریتم‌های جستجو اغلب از جدول پراکندگی که تمامی گره‌های تولیدی را ذخیره می‌کند، استفاده می‌کنند. این امر کنترل وضعیت‌های تکراری را بخوبی امکان‌پذیر می‌سازد. بده - بستان بین هزینه مرتب‌سازی و کنترل و هزینه اضافه جستجو بسته به مسئله دارد. فضاهای حالت حلقه‌ای تر بیشتر به کنترل تمایل دارند.

## ۲-۲- روش‌های جستجوی آگاهانه

دریافتیم که مدل‌سازی مسئله بصورت گراف فضای حالت اولین قدم در راه‌حل مسئله است. در فصل قبل دیدیم که روش‌های غیر هوشمند چگونه قادر به جستجو بر روی این گراف جهت دار هستند. اکنون زمان آن فرا رسیده که به سراغ روش‌های هوشمند رویم. روش‌های غیر آگاهانه در بدترین شرایط ناگزیر هستند که تمامی گره‌های فضای حالت برای یافتن پاسخ را جستجو کنند. اگر تعداد گره ها به اندازه کافی بزرگ باشد، این روش‌ها در زمان قابل قبول قادر به یافتن پاسخ نخواهند بود.

روش‌های هوشمند برای این طراحی شده‌اند تا به جای جستجوی تمامی گره‌ها، با هدف و جهت معین بخشی از گره‌ها را جستجو کرده و پاسخ را در بین آنها بیابند. برای حصول این هدف دو کار باید صورت گیرد:

الف - تابعی معین کند که گره جاری، گره مناسبی برای رسیدن به جواب هست یا خیر (تابع کشف کننده)

ب - استراتژی جستجوای که قادر به استفاده از این تابع کشف کننده باشد.

## توابع کشف کننده (heuristic functions)

معمای ۸ یکی از مسائل اولیه کشف کنندگی بود. همان طور که قبلاً ذکر شد، هدف از معما لغزاندن چهار خانه‌ها به طور افقی یا عمودی به طرف فضای خالی است تا زمانی که ساختار کلی مطابق با هدف باشد.

معمای ۸ یکی از مسائل مشکل است که در نوع خودش جالب نیز است. یکی راه‌حل نمونه در حدود ۲۰ مرحله دارد. اگر چه به حالت اولیه نیز بستگی دارد. فاکتور انشعاب در حدود ۳ (زمانی که خانه خالی در وسط باشد، ۴ حرکت ممکن وجود دارد؛ زمانی که خانه خالی در گوشه‌ها باشد، ۲ حرکت و زمانی که در لبه‌ها باشد ۳ حرکت وجود دارد) است. بدین معنا که یک جستجوی خسته کننده با عمق ۲۰ در حدود $۳^{۲۰} = ۳/۵\times ۱۰^{۹}$ حالت وجود دارد.

توسط نگهداری اثر حالات تکراری، می‌توانیم آن را کاهش دهیم چون فقط $۹! = ۳۶۲۸۸۰$ ترتیب متفاوت از ۹ مربع وجود دارد. ولی هنوز تعداد حالات زیاد است، بنابراین کار بعدی، یافتن یک تابع کشف کننده خوب است. اگر خواستار یافتن راه‌حل‌های کوتاه باشیم، به یک تابع کشف‌کننده نیاز داریم که هرگز در تعداد مراحل به هدف اغراق نکند. اینجا ما دو کاندید داریم:

- $h_1$ = تعداد چهار خانه‌هایی که در مکانهای نادرست هستند. در شکل ۱۱، ۷ عدد از ۸ چهارخانه خارج از مکان واقعی هستند. بنابراین حالت شروع دارای $h_1 = ۷$ خواهد بود. $h_1$ یک کشف‌کننده مجاز است. زیرا واضح است که هر چهار خانه‌ای که خارج از مکان درست باشد حداقل یکبار باید جابه‌جا شود.
- $h_2$ = مجموع فواصل چهارخانه‌ها از مکانهای هدف صحیح‌شان است. زیرا چهار خانه ها نمی‌توانند به صورت قطری حرکت کنند. فاصله‌ای که ما حساب می‌کنیم، مجموع فواصل عمودی و افقی است که بعضی وقتها city block distance و یا Manhattan distance نامیده می‌شوند. $h_2$ هم‌چنین مجاز است، چون هر حرکتی فقط می‌تواند یک چهارخانه را یک مرحله به هدف نزدیکتر کند، چهار خانه‌های ۱ تا ۸ در حالت شروع فاصلهٔ مانهاتانی در حدود $h_2 = ۲+۳+۳+۲+۴+۲+۰+۲ = ۱۸$ می‌دهد

### اثر صحت کشف کنندگی بر کارآیی

یک راه برای تشخیص کیفیت کشف کنندگی فاکتورانشعاب موثر $b^*$ (effetive branching factor) است. اگر مجموع تعداد گره‌های بسط داده شده توسط $A^*$ برای یک مسئله ویژه N باشد و عمق راه‌حل d سپس $b^*$ فاکتور انشعابی است که یک درخت یکنواخت با عمق d خواهد داشت تا گره‌های N را نگه دارد. از این رو، $N = ۱ + b^* + (b^*)^۲ ... + (b^*)^d$ برای مثال، اگر $A^*$ راه‌حلی را در عمق ۵ با استفاده از ۲۵ گره پیدا کند. فاکتور انشعاب موثر ۱/۹۱ است.




شکل ۱۱- مسئله معمای هشت

معمولاً فاکتور انشعاب موثر که توسط کشف کنندگی نمایش داده می‌شود، علاوه بر وجود دامنه وسیعی از نمونه‌های مسئله، مقدار ثابتی دارد، و بنابراین اندازه‌گیریهای تجربی $b^*$ روی مجموعهٔ کوچکی از مسائل می‌تواند راهنمای خوبی برای مفید بودن کشف کنندگی ایجاد کند. یک کشف‌کنندگی خوب طراحی شده، $b^*$ در حدود ۱ دارد، که اجازه می‌دهد مسائل زیادی حل شوند. برای امتحان توابع کشف کننده $h_1$، $h_2$، ۱۰۰ مسئله تصادفی را که هر کدام طولهای حل ۲،۴،...،۲۰ داشتند، با استفاده از جستجوی $A^*$ با $h_1$، $h_2$ به خوبی جستجوی عمیق کننده تکراری حل شدند. نتایج نشان داد که $h_2$ بهتر از $h_1$ است، و جستجوی ناآگاهانه بدترین است.

کسی ممکن است بپرسد آیا $h_2$ همیشه از $h_1$ بهتر است؟ جواب مثبت است. آسان است که از تعاریف دو کشف کننده بفهمیم که برای هر گروه $h_2(n) \geq h_1(n)$ است. ما می‌گوییم که $h_2$ بر $h_1$ تسط دارد. تسلط داشتن مستقیماً به کارآبودن بر می‌گردد:

### ابداع توابع کشف‌کننده

دیدیم که $h_1$ و $h_2$ هر دو کشف‌کننده‌های خوبی برای مسئله معمای ۸ هستند، و هم‌چنین $h_2$ مناسب‌تر است. ما نمی‌دانیم که چطور یک تابع کشف‌کننده را به وجود بیاوریم. چگونه کسی می‌تواند به $h_2$ برسد؟ آیا برای یک کامپیوتر ممکن است که به طور مکانیکی چنین تابعی را تولید کند؟

$h_1$ و $h_2$ برای طول‌های مسیر مسئله معمای ۸ تخمین زده می‌شوند، اما آنها می‌توانند طولهای مسیر کاملاً صحیحی داشته باشند اگر معما صورت ساده‌تری به خود بگیرد. اگر قوانین معما به صورتی تغییر کند که هر چهارخانه می‌تواند به هر جایی حرکت کند، به جای اینکه فقط به خانه خالی مجاور برود، سپس $h_1$ به درستی قادر خواهد بود تا تعداد مراحل کوتاهترین مسیر را تعیین کند. مشابهاً، اگر یک چهارخانه بتواند در هر جهتی یک خانه حرکت کند، حتی به درون یک خانه اشغال شده، $h_2$ قادر خواهد بود که تعداد دقیق مراحل کوتاه‌ترین راه‌حل را ارائه دهد. مسئله‌ای با محدودیت‌های کمتر بروی عملگرها یک مسئله راحت (relaxed problem) نامیده می‌شود. هزینه راه‌حل دقیق یک مسئله راحت، کشف‌کنندهٔ خوبی برای مسئله اصلی است.

اگر تعریف مسئله به زبان رسمی نوشته شده باشد، امکان دارد که مسائل راحت را به طور اتوماتیک بوجود آوریم. برای مثال، اگر عملگرهای معمای ۸ به صورت زیر تعریف شوند: یک چهارخانه می‌تواند از خانه A به خانه B حرکت کند اگر A همسایه B باشد و B یک خانه خالی باشد، ما می‌توانیم سه مسئله راحت را توسط حذف یک یا بیشتر شرایط تولید کنیم:

الف - یک چهارخانه می‌تواند از خانه A به خانه B برود اگر مجاور همدیگر باشند

ب - یک چهارخانه می‌تواند از خانه A به خانه B برود اگر B خالی باشد.

B برود.

ج- یک چهارخانه می‌تواند از خانه A به خانه B برود.

اخیراً، برنامه‌ای که ABSOLVER نام دارد نوشته شده که می‌تواند کشف‌کننده‌ها را به **طور اتوماتیک با** استفاده از تعاریف مسئله تولید کند، که از متد «مسئله راحت» و دیگر تکنیک‌ها **استفاده می‌کند absolver** کشف‌کننده جدیدی را برای معمای مشهور مکعب Rubik پیدا کرده است.

یکی از مشکلات تولید توابع کشف‌کننده جدید، شکست دریافتن بهترین تابع است. **اگر مجموعه‌ای از توابع** کشف‌کننده $h_1 ... h_m$ موجود باشد، و هیچکدام از آنها بر دیگری برتری نداشته باشند، **در انتخاب بهترین** تابع دچار مشکل خواهیم شد و نیاز داریم بالاخره یکی را انتخاب کنیم. **ما می‌توانیم بهترین آنها را با** تعریف داشته باشیم:

$$h(n) = \max(h_1(n), ..., h_m(n))$$

این کشف کننده مرکب، آن تابعی را استفاده می‌کند که در گره مورد پرسش، **صحیح‌تر باشد. زیرا** قسمت‌های کشف‌کننده‌ها، قابل قبول هستند، بنابراین h نیز قابل قبول است. **علاوه بر آن، h بر تمام** کشف‌کننده‌های اختصاصی، برتری دارد.

راه‌دیگری برای ابداع یک کشف‌کننده خوب، استفاده از اطلاعات آماری است. **این اطلاعات می‌توانند توسط** اجرای جستجو روی تعدادی مسائل جمع‌آوری شوند، مانند ۱۰۰ مسئله‌ای که **ساختار معمای ۸ را داشته** باشند و به طور تصادفی انتخاب شده باشند، و در انتها آمار آنها گرفته می‌شود. **برای مثال، در می‌یابیم** زمانی که $h_2(n) = ۱۴$، ۹۰٪ مواقع فاصله حقیقی به هدف ۱۸ است. **سپس زمانی که با مسئله «واقعی»** روبرو می‌شویم، می‌توانیم از مقدار ۱۸ زمانی که $h_2(n) = ۱۴$ استفاده کنیم. **البته، اگر از اطلاعات احتمالی** مانند این استفاده کنیم، در برابر ضمانت قابل قبول بودن تسلیم می‌شویم، **ولی تمایل پیدا می‌کنیم که** گره‌های کمتری را بسط دهیم.

اغلب ممکن است که طرح‌هایی از یک حالت را انتخاب کنیم که تابع ارزیاب **کشف‌کننده را شرکت می‌دهد** حتی اگر مشکل باشد که دقیقاً بگوییم چه شرکتی باید باشد. برای مثال، **هدف در بازی شطرنج، کیش و** مات کردن رقیب است، و طرحهای مربوطه شامل تعداد و نوع مهره‌ها و مکان آنها، **ترکیبی خطی از مقادیر** طرح می‌باشد. حتی اگر هیچ ایده‌ای در مورد درجه اهمیت طرحها نداشته باشیم، **و ندانیم که کدام خوب یا** بد است، هنوز امکان دارد که یک الگوریتم یادگیری را به منظور درخواست **امتیازات منطقی برای هر طرح،** استفاده کنیم. در شطرنج، برای مثال، یک برنامه می‌تواند یاد بگیرد که **کسی دارای وزیر باشد امتیاز مثبت** زیادی را می‌تواند داشته باشد، در صورتی که **پیادهٔ حریف دارای امتیاز منفی است.**

عامل دیگری که تابه‌حال در نظر نگرفته‌ایم، جستجوی هزینه واقعی **اجرای تابع کشف‌کننده روی یک گره** است. فرض کرده بودیم که هزینه محاسبه تابع کشف‌کننده **در حدود هزینه بسط یک گره است، بنابراین به** حداقل رساندن گره‌های بسط داده شده، کار مناسبی است. اما اگر **تابع کشف‌کننده بسیار پیچیده باشد و** محاسبه مقدار آن برای یک گره زمان زیادی برابر با بسط صدها گره داشته باشد، **بنابراین ناچاریم که** تجدید نظر کنیم. بالاخره، ساده است که کشف کننده‌ای داشته باشیم **که کاملاً صحیح عمل کند. اگر ما به** کشف کننده اجازه عمل بدهیم، در اصل یک جستجوی واقعی به حداقل می‌رساند، **اما هزینه جستجو را** کاهش نخواهد داد. یک تابع کشف‌کننده خوب باید همان‌طور که صحیح عمل می‌کند، **کارآیی نیز داشته باشد.**



### استراتژی جستجوی Best-First

مشهورترین استراتژی جستجوی آگاهانه Best-First نام دارد که الگوریتم آن به قرار زیر است:

```
best - first( )
{
open = [start];
closed = [];
x = delete Queue(open);
if(GoalTest(x) == true)
return path;
generate childs of x;
for (each childs of x)
case : the child is not onopen or closed
{
assign the child a heunistic value;
add it to open;
}
case : the child is already on open
if (the child was reached by a shorter path)
correct the path;
case :the child is already on closed
if(the child was reached by a shorter path)
{
remove it from closed;
add it to open;
}
put x on closed;
reoder open based on heuristic values;
}//end of while
retuen failure;
}
```

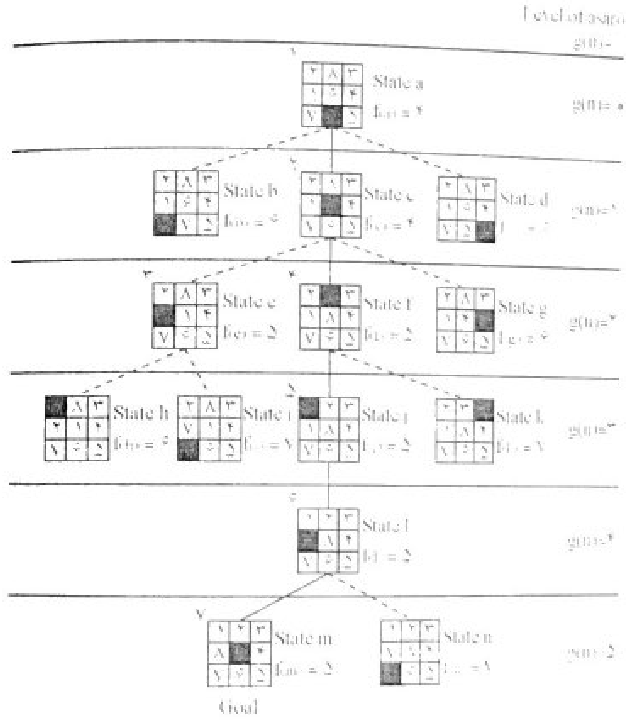
در این الگوریتم دو مجموعه open و closed تعریف شده که در مجموعه open گره‌هائی که در نوبت بسط قرار دارند ذخیره می‌شوند و closed به گره‌هائی تعلق دارد که بسط یافته‌اند. به همین دلیل ابتدا closed به تهی و open به گره آغازین مقداردهی اولیه شده است. سپس گره سمت چپ یا جلوی صف اولویت open را حذف کرده و اسم آن را x می‌گذاریم. تابع delete Queue عنصر جلوی صف open را حذف کرده و آنرا x می‌نامد. اگر این گره هدف بد الگوریتم به پایان می‌رسد و در غیر این صورت تمامی فرزندان آن

تولید خواهد شد. برای فرزند ۸ سه حالت وجود دارد. یا این فرزند قبلاً ملاقات نشده پس نه در open و نه در closed قرار دارد. دراین شرایط مقدار کشف‌کننده آن طبق تابع کشف‌کننده حساب شده و به مجموعه open یا گره‌های در انتظار برای توسعه اضافه خواهد شد.

دقت کنید مجموعه open همواره بر حسب مقدار کشف کنندگی هر گره مرتب است. پس در سمت چپ آن گره‌ای قرار دارد که بیشترین شانس رسیدن به جواب را خواهد داشت. معمولاً برای پیاده‌سازی open از صف اولویت استفاده می‌شود.

اما اگر گره قبلاً در open آمده باشد (یا بعبارتی از مسیر دیگری به آن رسیده باشیم)، باید کنترل شود آیا مسیر کنونی از مسیر قبلی کوتاهتر است یا خیر و در صورت کوتاهتر بودن مسیر فعلی، مسیر برای این گره تصحیح شود. در حالت سوم گره بسط داده شده و در closed قرار گرفته و اگر برای چنین گره‌ای از مبدا مسیر کوتاهتری پیدا شود، لازم است خود این گره و تمامی گره‌هایی که تحت آن بسط یافته‌اند تصحیح مسیر گردند.

برای درک بهتر این الگوریتم به مثال زیر توجه کنید: (در این مثال $g(n)$ معرف فاصله از مبدا و $h(n)$ معرف تعداد عناصری است که در هر گره در جای نهائی خود قرار ندارند و $f(n) = g(n) + h(n)$)




شکل ۱۲- مثالی از حل مسئله معمای هشت بوسیله الگوریتم اول - بهترین



## الگوریتم $A^*$

تاکنون با استراتژی اول - بهترین (Best – First) آشنا شدیم. دیدیم که [illegible] محاسبه اولویت در صف اولویت هر گره استفاده می‌شود. [illegible] اساسی پاسخ دهیم اول اینکه چرا تابع $f(n)$ باید مورد استفاده قرار گیرد و دوم اینکه تحت چه شرایطی استراتژی اول - بهترین منجر به یافتن پاسخ بهینه خواهد شد؟

قبلاً تعریف کردیم که اگر n گره‌ای متعلق به گراف فضای حالت [illegible] $f(n) = g(n) + h(n)$ را تعریف نمود که در آن $g(n)$ فاصله واقعاً طی شده از مبدا تا گره [illegible] خروجی تابع کشف کننده مورد استفاده خواهد بود. متقابلاً $f^*(n) = g^*(n) + h^*(n)$ را تعریف می‌کنیم که در آن $g^*(n)$ حداقل فاصله ممکن است از گره آغازین تا گره n و $h^*(n)$ حداقل فاصله ممکن [illegible] تا نزدیکترین پاسخ خواهد بود. می‌توان حدس زد که $f^*(n)$ کوتاه‌ترین فاصله از گره آغازین تا نزدیکترین پاسخ است مشروط بر آنکه حتماً از گره n عبور کرده باشیم. مسلم است که اگر گره n بر روی مسیر بهینه قرار داشته باشد $f^*(n)$ پاسخ بهینه حل مسئله خواهد بود. از سوی دیگر $h(n)$ تخمینی از این فاصله است با همان شرط که حتماً از گره n نیز عبور کنیم. بسادگی می‌توان ثابت کرد $\forall n\ g(n) \geq g^*(n)$ چرا که طبق تعریف $g^*(n)$ همواره کوتاه‌ترین فاصله از مبدأ تا گره n است و بنابراین $g(n)$ نمی‌تواند کوتاهتر از آن باشد چون فاصله واقعاً طی شده است.

استراتژی Best –First را $A^*$ می‌گویند اگر $\forall n\ h(n) \leq h^*(n)$ باشد.

الگوریتم قابل پذیرش (admissible) نامیده می‌شود اگر برای هر گرافی منجر به یافتن پاسخ کامل و بهینه گردد.

**قضیه:** الگوریتم $A^*$ قابل پذیرش است. یعنی اگر مسئله پاسخی داشته باشد همواره [illegible] کوتاه‌ترین مسیر خواهد یافت.

**اثبات:** ابتدا تلاش می‌کنیم ثابت کنیم الگوریتم $A^*$ حتماً متوقف خواهد شد. این امر برای گراف‌های فضای حالت متناهی بدیهی است. اما برای اثبات این امر در گراف‌های نامتناهی [illegible] ثابت می‌کنیم که همواره گره‌ای همانند n در مجموعه open وجود دارد که $f(n)$ آن گره از طول مسیر بهینه کوچکتر یا مساوی است.

فرض کنید مسیر بهینه بصورت دنباله $n_0, n_1, \ldots, n_k$ تعریف شده که $n_0$ گره آغازین و $n_k$ هدف است. هر زمان قبل از اتمام $A^*$، گره‌ای همانند $n'$ در ابتدای این دنباله قرار دارد که در مجموعه open نیز وجود دارد. بنابراین همواره یکی از گره‌های متعلق به مسیر بهینه در مجموعه open یافت می‌شود که مکان آن در این مجموعه نامعین است ولی قطعاً در آن وجود دارد. از سوی دیگر داریم

طبق تعریف

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

از آنجا که $n'$ خود گره‌ای بر روی مسیر بهینه است و تمامی اعقاب آن گره نیز در مجموعه closed قرار دارند، سپس $g(n') = g^*(n')$ یعنی:

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

$$f(n') \le g^*(n') + h^*(n') = f^*(n')$$

$f^*(n')$ همان کوتاهترین مسیر از مبدأ به مقصد است که ما آنرا با $f^*(s)$ (s گره آغازین است) نشان می‌دهیم. بنابراین داریم:

**نتیجه ۱:** هر زمان قبل از اتمام الگوریتم $A^*$، گره‌ای همانند $n'$ در مجموعه open وجود دارد که این گره بر روی مسیر بهینه قرار داشته و $f(n') \le f^*(s)$ می‌باشد.

**نتیجه ۲:** اگر مسیر از گره آغازین به هدف وجود داشته باشد، $A^*$ **متوقف خواهد شد.**

علت این امر آنست که هر گره‌ای که نوبت توسعه به آن می‌رسد دارای مقدار f ای است که از یک کران بالا (که همان طول مسیر بهینه است) مقدار کوچکتری خواهد داشت. یافتن این کران بالا برای اعضای مجموعه open مبین رسیدن به پاسخ در صورت وجود است. دو نکته را فراموش نکنید: اول آنکه ثابت کردیم همواره $n'$ بر روی مسیر بهینه در open وجود دارد و دوم آنکه open یک صف اولویت است که بر اساس مقدار f مرتب شده است.

از نتیجه ۲ می‌توان ثابت کرد که هر گره‌ی همانند n که در مجموعه open قرار داشته و $f(n) \le f^*(s)$ باشد بوسیله $A^*$ بسط داده خواهد شد.

اکنون با کمک نتایج فوق اثبات قضیه اصلی ساده‌تر خواهد شد. فرض کنید $A^*$ به اتمام برسد و موفق به یافتن مسیر بهینه نگردد، یعنی مسیر غیر بهینه‌ای را به عنوان پاسخ تولید کند. این چنین رویه ای غیر ممکن است، چرا که از سوئی ثابت کردیم همواره گره‌ای بر روی مسیر بهینه همانند $n'$ که $f(n') \le f^*(s)$ است و اگر فرض کنیم $A^*$ بعنوان آخرین گره t را بسط داده و به پاسخ بهینه نرسیده، باید $f(n') \le f^*(s) < f(t)$ باشد. این امر ممکن نیست چون قبل از بسط گره t می‌بایست گره $n'$ توسعه یابد. بنابراین حکم ثابت است.

**نتیجه ۳:** برای هر گره‌ای همانند n که نوبت توسعه به آن توسط $A^*$ رسیده: $f(n) \le f^*(s)$

## مقایسه بین الگوریتم های $A^*$

دقت کشف کنندگی تابع h وابسته به دانشی است که این تابع از دامنه مسئله خود دارد. اگر $\forall n\ h(n) = 0$ باشد، الگوریتم پیشنهادی فاقد هر نوع هوشمندی خواهد بود، چرا که در این شرایط $\forall n\ f(n) = g(n)$ بوده یعنی تصمیم‌گیری بر مبنای مسیر طی شده از مبدأ صورت خواهد گرفت که این همان الگوریتم جستجوی سطحی (BFS) است.

فرض کنید دو الگوریتم $A^*$ بصورت زیر برای حل مسئله معین مطرح شده:

$$f_1(n) = g_1(n) + h_1(n)$$



$$f_2(n) = g_2(n) + h_2(n)$$

که در آن $\forall h_2(n) \le h^*(n)$ و $\forall n\ h_1(n) \le h^*(n)$ "الگوریتم $A_2$ آگاهتر از الگوریتم $A_1$ نامیده می‌شود اگر برای تمامی گره‌های غیر هدفی همانند n $h_2(n) > h_1(n)$ باشد.

**قضیه:** اگر $A_1$ و $A_2$ دو نسخه از $A^*$ برای حل یک مسئله باشند چنانچه $A_2$ آگاهتر از $A_1$ باشد، در زمان اتمام جستجو بر روی گراف فضای حالت، هر گره‌ای که توسط $A_2$ توسعه داده شده توسط $A_1$ نیز توسعه داده شده است یعنی $A_1$ حداقل تمامی گره‌های توسعه داده شده توسط $A_2$ را نیز توسعه می‌دهد.

## محدودیت یکنوائی (monotone)

استراتژی Best-First هر زمان گره‌ای همانند n را برای توسعه انتخاب می‌کند ممکن است گره‌های منشعب (successor) آن در مجموعه‌های open یا closed قرار داشته باشند. به همین علت در شرط دوم و سوم استراتژی مطرح شده، باید کنترل شود که برای این گره‌ها مسیر بهتری یافت شده یا خیر. این فرایند بویژه در زمانی که گره ما بعد در Closed بوده و از مسیر [illegible] گره‌های منشعب [illegible] یافته‌ای [illegible] نظر محاسباتی پر هزینه خواهد بود. پس بدیهی است اگر بتوان از وقوع چنین موردی جلوگیری کرد، [illegible] اجرای الگوریتم بشدت بهبود خواهد یافت.

تابع کشف‌کننده h دارای محدودیت یکنوائی است اگر برای تمامی گره‌های $n_i$ و $n_j$ چنانچه $n_j$ گره مابعد $n_i$ باشد داشته باشیم:

$$h(n_i) - h(n_j) \le c(n_i, n_j)$$

که در این رابطه $c(n_i, n_j)$ هزینه انتقال از گره $n_i$ به $n_j$ می‌باشد.

**قضیه:** اگر محدودیت یکنوائی برقرار باشد، الگوریتم $A^*$ همیشه هر گره‌ای را که بسط می‌دهد مسیر بهینه برای آنرا نیز یافته است. یعنی اگر $A^*$ گره n را برای توسعه انتخاب کند و اگر محدودیت یکنوائی نیز رعایت کند، آنگاه $g(n) = g^*(n)$ خواهد بود.

محدودیت یکنوائی نتیجه مهم دیگری را نیز ایجاد می‌کند. مقادیر f دنباله‌گره‌هائی که توسط $A^*$ برای بسط انتخاب می‌شوند، غیر نزولی است. زمانیکه محدودیت یکنوائی رعایت نشده باشد ممکن است برخی گره‌ها دارای مقادیر f کوچکتری از گره‌های توسعه یافته ما قبل خود داشته باشند.

## جستجوی Iterative deepening A*(IDA*)

نشان دادیم که عمیق‌کننده تکراری تکنیکی مفید برای کاهش درخواست حافظه است. می‌توانیم این ایده را مجددا تکرار کنیم. جستجوی $A^*$ را به عمیق کنندهٔ تکراری $A^*$ تبدیل کنیم تا IDA* [illegible] در این الگوریتم هر تکرار یک جستجوی عمقی است. جستجوی عمقی تغییر یافته تا محدودهٔ f cost را به جای یک محدوده عمقی استفاده کند. از این‌رو در هر تکرار، تمام گره‌های داخل ناحیه برای f cost جاری بسط می‌دهد [illegible] ناحیه را نیز جستجو می‌کند تا ناحیه بعدی را بیابد.

```
function IDA*(problem) returns a solution sequence
inputs: problem, a problem
loca variables: f - limit, the current f - CosT limit
            root, a node
root ← Make - NODE(INTIAL - STATE[problem])
f - limit ← f - Cost(root)
loop do
   solution, f - limit ← DFS - CONTOUR(root, f - limit)
   if solution is non - null then return solution
   if f - limit = ∞ then return failure; end
function DFS - CONTOUR(node, f - limit) returns a solution sequence and a new
 f - CosT limit
inputs: node, anode
       F limit, the current f - COST limit
Local variables: next - f, the f - COST limit for the next contour, initially ∞
if f - COST[node] > f - lim it then return null, f - COST[node]
if GOAL - TEST[problem](STATE[node]) then return node, f - limit
foreach node s in SUCCESSORS(node) do
   solution, new - f ← DFS - CONTOUR(s, limit)
   if solution is non - null then return solution, f - limit
   next - f ← MIN(next - f, new - f); end
return null, next, f
```

زمانی که جستجو داخل یک ناحیه داده شده، کامل شد، یک تکرار جدید با استفاده از یک f-cost **جدید برای** ناحیه بعدی، آغاز می‌شود.

IDA* با تقاضای مشابه A* کامل و بهینه است، اما چون عمقی است، فقط فضا را به **تناسب طولانی‌ترین** مسیری که پیدا شده، تقاضا می‌کند. اگر $\delta$ کوچکترین هزینه عملگر و $f^*$ هزینه راه‌حل **بهینه باشد، در** بدترین حالت، $bf^*/\delta$ گره از منبع درخواست می‌کند. در بیشتر موارد، bd تخمین **خوبی از تقاضاهای** منبع است.

پیچیدگی زمانی IDA* شدیداً به تعداد مقادیر مختلفی که تابع کشف کننده می‌تواند بگیرد، بستگی **دارد، در** کشف کننده فاصله Manhattan که در مسئله معمای ۸ کاربرد دارد، عموماً f فقط ۲ یا ۳ **بار در طول مسیر** افزایش پیدا می‌کند. از این رو، IDA* فقط دو یا سه تکرار را انجام می‌دهد، و کارآیی آن مشابه $A^*$ **است** در حقیقت، آخرین تکرار IDA* معمولاً تعداد مشابهی صف را بسط می‌دهد، و سرریزی هر **گرهٔ آن کمتر**



از A* می‌تواند باشد. راه‌حلهای بهینه برای مسائل عملی زیادی ابتدا توسط IDA* پیدا شدند، که برای سالهای زیادی تنها الگوریتم کشف کنندهٔ کامل با حافظه محدود بود.

متأسفانه IDA* در دامنه‌های پیچیده‌تر با مشکل روبرو می‌شود. در مسئله فروشندهٔ دوره گرد، هر ناحیه شامل یک حالت بیشتر از ناحیه قبلی است. اگر $A^*$ N گره را بسط دهد، IDA* باید N تکرار را انجام دهد، بنابراین $N^2$ مطمئناً زمان زیادی برای انتظار است!

یک راه برای جلوگیری از این مسئله، افزایش محدودهٔ f-cost توسط یک مقدار ثابت $\varepsilon$ روی هر تکرار است. بنابراین مجموع تعداد تکرارها متناسب با $1/\varepsilon$ است. این قضیه می‌تواند هزینه جستجو را کاهش دهد و برگرداندن راه‌حلهایی که بدتر از راه حل بهینه با وجود $\varepsilon$ است، جلوگیری کند. چنین الگوریتمی $\varepsilon$ − sdmissible نامیده می‌شود.

## جستجوی SMA*

مشکلات IDA* در فضاهای قطعی مسئله می‌تواند منجر به استفاده بسیار جزئی از حافظه موجود شود بین تکرارها، این جستجو فقط یک عدد منفرد از محدوده جاری را نگه می‌دارد. چون نمی‌تواند تاریخچه‌اش را به خاطر آورد، IDA* مجبور به تکرار آن می‌شود این مسئله زمانی که در فضاهای حالتی باشد که به جای درخت، گراف وجود دارد، دو چندان مصداق پیدا می‌کند. IDA* می‌تواند به منظور کنترل مسیر جاری برای حالتهای تکراری تولید شده توسط مسیرهای محلی اجتناب ورزد.

در این قسمت، ما الگوریتم SMA*، حافظه محدود A* ساده شده (Simplified - Memory - Bounded A*) را تشریح می‌کنیم که قادر است تا از تمام حافظه موجود برای اجرای جستجو استفاده کند. استفاده از حافظه بیشتر کارآیی جستجو را وسعت می‌بخشد. می‌توان همیشه از فضای اضافی صرفنظر کرد. اما معمولاً بهتر است که گره‌ای را داشته باشیم که زمانی مورد نیاز است، آن را تولید کند. SMA* دارای خواص زیر است:

- می‌تواند ازتمام حافظه دسترس استفاده ببرد.
- از حالات تکراری تا جایی که حافظه اجازه می‌دهد، جلوگیری می‌کند
- این الگوریتم کامل است بشرط آنکه حافظه برای ذخیره کم عمق‌ترین مسیر راه‌حل کافی باشد
- این الگوریتم بهینه است، اگر حافظه کافی برای ذخیره کم عمق‌ترین مسیر راه‌حل کافی باشد. بعلاوه بهترین راه‌حلی را بر می‌گرداند که بتواند با حافظه موجود مطابقت داشته باشد.
- زمانی که حافظه موجود برای درخت جستجوی کامل کافی باشد، جستجو Optimally efficient است

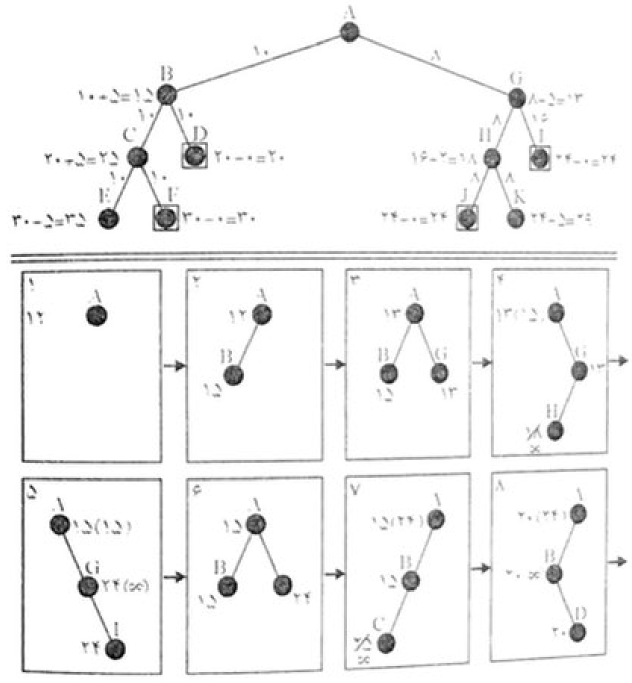
یکی از سوالات حل نشده این است که آیا SMA* همیشه در بین تمام الگوریتمهایی که اطلاعات کشف کنندگی مشابه می‌دهند و حافظه مشابهی را نیز به خود اختصاص می‌دهند، Optimally efficient است؟ طراحی SMA* ساده است. زمانی که نیاز به تولید فرزند داشته باشد ولی حافظه‌ای نداشته باشد، نیاز به ساختن فضا بروی صف دارد. برای انجام این امر، یک گره را حذف می‌کند. گره‌هایی که به این طریق از صف حذف می‌شوند، گره‌های فراموش شده یا (forgotten nodes) نامیده می‌شوند گره‌هایی محدود زیر

درختهایی که از حافظه حذف شده‌اند، در گره‌های اجدادی، اطلاعاتی در مورد کیفیت بهترین مسیر در زیر درخت فراموش شده، نگهداری می‌شود. از این طریق، فقط زمانی زیر درختها دوباره تولید می‌شوند که ثابت شود تمام مسیرهای دیگر بدتر از مسیر فراموش شده باشند. راه دیگر این است که اگر تمام نسلهای یک گره n، فراموش شده باشند و ما نمی‌دانیم که کدام راه از n می‌رود، اما هنوز ایده‌ای داریم که نشان می‌دهد که رفتن به جای دیگر از n چقدر ارزش دارد.

SMA* را می‌توان با یک مثال بهتر تعریف کرد که در شکل۱۳ نشان داده می‌شود. بالای شکل فضای جستجو را نشان می‌دهد. هر گره با مقادیر f=g+h بر چسب خورده است، و گره‌های هدف (J,I,F,D) هستند که در چهار گوشه‌ها نشان داده شده‌اند. هدف یافتن گره هدف با کمترین هزینه و حافظه کافی برای فقط سه گره است. مراحل جستجو به ترتیب از چپ به راست نشان داده شده است و هر مرحله با عددی که توضیح آن در زیر آمده، مشخص شده است. هر گره با f-cost هر یک نسلهایش را منعکس کند. مقادیری که در پرانتز هستند، ارزش بهترین نسل فراموش شده را بر می‌گردانند.

الگوریتم طبق مراحل زیر دنبال می‌شود:

۱- در هر مرحله، یک فرزند به عمیق‌ترین گره که دارای حداقل f-cost باشد اضافه می‌شود و فرزندانی دارد که در حالت جاری در درخت نیستند. فرزند چپ که B است به ریشه A اضافه می‌شود.



شکل ۱۳- مثالی از الگوریتم SMA*



۲- اکنون هنوز f(A) = ۱۲، بنابراین ما فرزند سمت راست را اضافه می‌کنیم. G(f = ۱۳) حالا که ما تمام فرزندان A را دیدیم، می‌توانیم f-cost آن را به حداقل فرزندانش، تغییر دهیم که مقدار ۱۳ است. حافظه اکنون پر است.

۳-G در حال حاضر آمادهٔ بسط دادن است، اما ابتدا باید گره‌ای را حذف کنیم تا فضای کافی موجود باشد. برگی که کم‌عمق‌ترین است و بیشترین f-cost را دارد که همان B است را حذف می‌کنیم. زمانی که این عمل را انجام دادیم، متوجه می‌شویم که بهترین نسل فراموش شده [illegible] دارد، که در پرانتز نشان داده شده است. سپس H را با f(H) = ۱۸ اضافه می‌کنیم. متأسفانه، H گره هدف نیست. اما مسیری که به H می‌رود تمام حافظه موجود را استفاده می‌کند. از این رو، راهی برای یافتن یک راه‌حل از طریق H نیست، بنابراین f(H) = ∞ قرار می‌دهیم.

۴- G دوباره بسط داده می‌شود. H را حذف کردیم و I را اضافه نمودیم. f(I) = ۲۴ حالا هر دو فرزند G را دیدیم، با مقادیری از ∞ و ۲۴، بنابراین f(G) برابر با ۲۴ می‌شود. f(A) برابر با ۱۵ می‌شود که همان می‌نیمم ۱۵ (مقدار فرزند فراموش شده) و ۲۴ است. توجه کنید که I یک گره هدف است، اما بهترین راه‌حل نیست چون f-cost مربوط به A فقط ۱۵ است.

۵- A دوباره گره‌ای است که بیشترین تعهد را دارد، بنابراین B برای بار دوم تولید می‌شود. ما درمی‌یافتیم که مسیر از طریق G چندان جالب نبود.

۶- C، اولین فرزند B ، یک گره غیر هدفی در عمق حداکثر است، بنابراین f(C) = ∞ است.

۷- در نگاه کردن به فرزند دوم ، D، ابتدا C را حذف می‌کنیم. سپس f(D) = ۲۰ می‌شود، و این مقدار توسط A،B به ارث می‌رسد.

۸- اکنون، عمیق‌ترین و کمترین f-cost را گره D دارد، بنابراین D انتخاب می‌شود، و چون گرهٔ هدف است، جستجو خاتمه پیدا می‌کند.

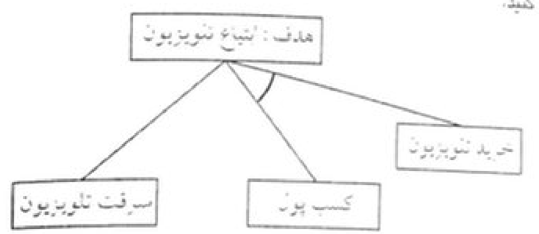
در این مورد، حافظه کافی برای کم عمق‌ترین مسیر حل بهینه وجود دارد. اگر J هزینه ۱۹ به جای ۲۴ داشت، بهرحال، SMA* هنوز قادر نبود که آن را پیدا کند زیرا مسیر راه‌حل شامل چهار گره است. از این رو SMA* D را برگرداند، که بهترین راه‌حل قابل دسترسی است. ساده است که الگوریتمی موجود باشد تا اطلاع دهد راه حل پیدا شده بهینه نیست.

با دادن میزان حافظه منطقی، SMA* می‌تواند مسائل مشکل‌تری را نسبت به A* حل کند، بدون آنکه متحمل سرریزی به صورت گره‌های اضافه تولید شده، شود. این جستجو ارائه خوبی را با فضاهای حالت مرتبط به هم و کشف کننده‌هایی با ارزش واقعی روی مسائل دارد، بهرحال، اغلب مواقع این مسئله بوجود می‌آید که SMA* مجبور می‌شود که به طور پیوسته بین یک سری از مسیرهای حل کاندید شده، رفت و برگشت داشته باشد. بدین معنا که مسائل که به طور عملی توسط A* قابل حل شدن هستند، با دادن حافظه نامحدود، برای SMA* حل نشدنی می‌شوند. مانند این است که بگوییم محدودیت حافظه می‌تواند مسئله را از نقطه نظر زمان محاسبه حل نشدنی کند. اگر چه تئوری‌ای وجود ندارد که ارتباط بین زمان و حافظه را تعریف کند، و به نظر می‌رسد که این یک مسئله غیر قابل اجتناب باشد. تنها راه‌نجات، حذف درخواستهای بهینگی است.

### تقلیل مسئله

تاکنون الگوریتم‌هائی طرح شدند که بر روی گراف فضای حالت از نوع گراف OR اعمال می‌شوند. منظور از گراف OR آنست که برای یافتن مسیر حل مسئله از گره آغازین تا هدف بدنبال یک مسیر واحد هستیم بعبارت دیگر لبه‌های متصل به هر گره، هر کدام از دیگری مستقل می‌باشند. در چنین ساختاری برای انتخاب گره‌بعدی از گره جاری، می‌توان از هر لبه‌ای بطور مستقل استفاده نمود.

نوع دیگری از ساختار گراف، گراف AND-OR می‌باشد. این ساختار برای حل مسائلی بکار برده می‌شود که برای حل می‌بایست مسئله به زیر مسائل کوچکتری شکسته شود و تمامی زیر مسائل می‌بایست حل شوند این تجزیه یا تقلیل، لبه‌هائی در گراف تولید می‌کند که لبه‌های AND نامیده می‌شوند. یک لبه AND می‌تواند به هر تعداد گره ما بعد اشاره کند که تمامی آنها می‌بایست حل شوند تا راه‌حل حاصل شود. همانند گراف OR ممکن است تعدادی لبه از یک گراف واحد نشأت گرفته باشند که مبین راه‌حل‌های متفاوت ممکن برای حل مسئله هستند به همین علت گراف می‌تواند لبه‌های AND و یا OR داشته باشد. برای نمونه به شکل زیر توجه کنید.




شکل ۱۴- مثالی از گراف AND-OR

برای اعمال الگوریتم *A بر روی گراف AND-OR، باید تغییراتی در این الگوریتم برای برخورد با لبه‌های AND صورت گیرد. پاسخ الگوریتم، یک زیر گراف خواهد بود.

برای درک الگوریتم جستجو بر روی گراف AND-OR ، سه نکته در هر مرحله نباید فراموش شود:

۱- گراف را از گره آغازین پیمایش کن و بهترین مسیر کنونی را تعقیب کن و در کنار آن مجموعه گره‌هائی که بر روی مسیر قرار دارند و تاکنون بسط نیافته‌اند را جمع کن

۲- یکی از گره‌های توسعه نیافته را انتخاب کن و توسعه بده. فرزندان آنرا تولید کن و مقادیر F مربوط به هر گره را محاسبه کن (در این شرایط $f(n) = h(n)$ خواهد بود و از مقدار g صرفنظر می‌گردد)

۳- با یافتن گره‌های ما بعد، مقادیر تخمین زده شده f را تغییر بده . این تغییرات را در جهت وارونه در گراف توسعه بده برای هر گره‌ای که حین بالا رفتن از گراف ملاقات می‌شود، تصمیم‌بگیر که کدام لبه‌های ما بعد محتمل تر هستند و آنرا بعنوان بهترین مسیر کنونی علامت بزن. این فرایند در الگوریتم *A صورت نمی‌گیرد.

جستجو بر روی گراف AND-OR از جنبه دیگری نیز از جستجو بر روی گراف OR متمایز می‌شود نباید فراموش کرد که مسیرهای جداگانه از گره دیگر، نمی‌توانند مستقل از مسیرهائی در نظر گرفته شوند که



توسط لبه‌های AND به گره‌های اصلی متصل شده‌اند. در الگوریتم *A مسیر مورد نظر از یک گره به دیگری، همواره با کمترین هزینه خواهد بود ولی در گراف AND-OR همیشه اینگونه نیست

نکته دیگر آنست که گراف AND-OR باید فاقد حلقه (cycle) باشد. اگر چه این محدودیت عمل جستجو در گراف را به ظاهر ساده‌تری سازد، اما کار در جای دیگری دشوار خواهد شد. هر زمان که گره ما بعدی توسعه یافته، باید کنترل کنیم که این گره، ما قبل گره توسعه داده شده‌ای نباشد.

## چند توسعه بر الگوریتم *A

یکی از ایده‌هائی که در سال اخیر برای بهبود الگوریتم *A مطرح شده، جستجوی اول - بهترین بازگشتی نام دارد. این الگوریتم نوع توسعه یافته استراتژی اول - بهترین است. این روش بر مبنای پشته‌های بازگشتی عمل می‌کند که مسیر کامل از مبدأ تا گره جاری از برخی گره‌های بخش توسعه یافته درخت تجاوز کرد، الگوریتم به عمیق‌ترین نیای (ancestor) مشترک باز می‌گردد و جستجو را از مسیر جدیدی به پایین آغاز می‌کند. الگوریتم اول بهترین بازگشتی می‌تواند با تعداد کمتری گره نسبت جاری مسیر را نگهداری می‌کند. می‌توان ثابت نمود الگوریتم اول - بهترین بازگشتی می‌تواند با تعداد کمتری گره نسبت به *IDA به جواب برسد. تنها تولید گره‌ها در این روش هزینه بیشتری دارد.

در برخی از کاربردها (همانند مسیریابی روبات متحرک) امکان حل یکباره مسئله وجود ندارد. بعبارت ساده‌تر در مثال روبات متحرک مسیریاب، دید روبات ناگزیر به حرکت است تا بتواند در مورد حرکت آینده خود تصمیم‌گیری کند. از سوی دیگر گاهی زمان کافی برای تصمیم‌گیری نیز وجود ندارد و می‌بایست و در یک بازی محدود زمانی حرکت بعدی را انتخاب کرد.

تحت چنین شرایطی که معمولاً به شرایط بلادرنگ (real - time) شهرت دارد، الگوریتم‌های جستجو می‌بایست دستخوش تغییراتی شوند. از جمله این گروه از تغییرات در الگوریتم *A می‌توان به الگوریتم *RTA (Real - Time A*) اشاره کرد. *RTA بر مبنای تابع F(n) و بسته به زمان، گراف فضای حالت را تا جای ممکن توسعه می‌دهد. پس از اتمام زمان از بین گره‌های موجود در صف اولوت، گره جلوی صف را بعنوان حرکت بعدی انتخاب خواهد کرد. بر مبنای این تصمیم حرکت بر روی گراف آغاز خواهد شد در بازه زمانی بعدی ک فرصت به ادامه تصمیم‌گیری داده خواهد شد. در صورتیکه تابع مکاشفه‌ای f(n) به درستی عمل نکرده باشد و متوجه اشتباه در تصمیم‌گیری شویم، تلاش می‌کند بهترین راه برای القاء نمودن حرکت جاری و بازگشت به مسیر بهینه را پیدا کند. منطق حکم می‌کند که بازگشت به عقب هزینه بیشتری از ادامه مسیر جاری داشته باشد. به همین علت این الگوریتم معمولاً تمایل به ادامه مسیر جاری دارد، مگر آنکه اشتباه در محاسبه f(n) آنقدر باشد که هزینه بازگشت به عقب و ادامه مسیر در راه بهینه مقرون به صرفه باشد.

### ۲-۳- مسائل ارضای محدودیت (constraint satisfaction)

مسئله ارضاء محدودیت نوع خاصی از مسائل هستند که در آن هر مسئله حاوی مجموعه‌ای از متغیرهاست که هر متغیر دربازه معین از مقادیر تعریف شده است. هدف یافتن مقادیر معینی برای تمام متغیرها بگونه‌ای است که محدودیت‌های ذکرشده در صورت مسئله رعایت شوند.

برای نمونه می‌توان به مسئله مشهور ۸ وزیر اشاره کرد که می‌توان آنرا یک نمونه از مسائل ارضاء محدودیت در نظر گرفت. مکان قرار گرفتن هر وزیر در یک سطر را می‌توان متغیر در نظر گرفت که در این صورت مسئله دارای هشت متغیر است که بازه تغییرات تمامی آنها بین ۱ تا ۸ است. انتخاب ترکیبی از پاسخ برای این هشت متغیر باید محدودیت تهدید سطری، ستونی و قطری را رعایت کنند.

اگر چه امکان حل این دسته از مسائل با روش‌های جستجو اشاره شده نیز وجود دارد ولی به علت ساختار خاص این مسائل بهتر است از الگوریتم های ویژه برای حل استفاده گردد تا کارائی راه‌حل بهبود یابد.

محدودیت‌ها می‌توانند به اشکال متنوعی ظاهر شوند. محدودیت ممکن است یکتا باشد، به این معنی که مقدار آن تنها به یک متغیر وابسته باشد. ممکن است دو تائی باشند یعنی به دو متغیر وابسته باشند والی آخر. محدودیت می‌تواند مطلق باشد یعنی عدم پیروی از آن راه‌حل را رد می‌کند و یا اینکه اولویت دار باشد که تعیین می‌کند کدام راه‌حل ارجع‌تر است.

اگر متغیر $V_i$ در مسئله ارضا محدودیت متعلق به بازه مقادیر $D_i$ باشد، محدودیت یکتا زیر مجموعه‌ای از مقادیر مجاز متعلق به این بازه است و محدودیت دو تائی شامل زیر مجموعه مجازی از تولیدات مشترک دو دامنه را تشکیل می‌دهد اگر مسئله ارضا محدودیت گسسته باشد، محدودیت ها می‌توانند بسادگی شمارش شوند. اما در صورت پیوسته بودن مقادیر متغیر، حل نیازمند به استفاده از روش‌های جبری است.

برای حل مسائل ارضاء محدودیت با استفاده از ایده الگوریتم‌های جستجو، کافیست حالت اولیه یا همان وضعیت آغازین را حالتی قرار دهیم که در آن تمامی متغیرهای مسئله فاقد مقدار هستند. عملگرها مقداری به یک متغیر دربازه آن نسبت می‌دهند و آزمون هدف کار کنترل محدودیت را به انجام می‌رساند. بدین ترتیب حداکثر عمق گراف فضای حالت n (تعداد متغیرها) خواهد بود، پس می‌توان نتیجه گرفت ایده جستجوی DFS با فکر قرار گرفتن در مسیر بی‌پایان، روبرو نخواهد شد. از سوی دیگر فاکتور انشعاب به اندازه $\sum_i |D_i|$ خواهد بود.

برای رشد DFS بهتر است از روشی همانند پی‌جوئی به عقب (back tracking) استفاده کنیم یعنی به مجرد برخورد به نقض محدودیت، از ادامه زیر شاخه منصرف شده و دیگر مسیرهای ممکن در گراف فضای حالت را تست کنیم. البته پی جوئی به عقب خود با مشکلاتی روبرو است. فرض کنید در مسئله هشت وزیر، شش وزیر اول انتخاب شده‌اند ولی همگی ستون‌های سطر هشتم تهدید می‌شوند. پی‌جوئی به عقب مجبور است تمام ترکیبات دو سطر باقیمانده را مورد آزمون قرار دهد که این کار زمانگیر خواهد بود. برای رفع این نقصیه از کنترل رو به جلو (Forward checking) استفاده می‌شود، بدین ترتیب که هر زمان متغیری تعریف می‌شود کنترل رو به جلو تلاش می‌کند مغایرت با متغیرهای بعدی را کنترل کند. در صورت یافتن



مغایرت زیر گراف مربوط به آنرا حذف خواهد کرد و در صورت عدم تهی شدن زیر مسیرهای [illegible] به جلو حرکت خواهد کرد و اگر زیر شاخه‌ای برای آزمون باقی نمانده است به سراغ پی جوئی به عقب خواهد رفت.

تاکنون به حل مسائل ارضاء محدودیت توسط روش‌های غیر هوشمند (عمدتاً DFS) پرداختیم [illegible] در مورد مسائل بزرگ با گراف فضای حالت گسترده روش‌های غیر هوشمند نمی‌توانند [illegible] باشند. اما برای ابداع روش‌های هوشمند در این گستره، ابتدا لازم است تابع کشف کنندگی [illegible] از مسائل طرح کنیم. برای اینکار از دو ایده اساسی متغیر با محدودیت حداکثر [illegible] variable) و با محدودیت حداقل استفاده می‌شوند.

کشف‌کننده متغیر با محدودیت حداکثر سعی می‌کند با استفاده از ایده کنترل رو به جلو مقادیری که [illegible] متغیرها نسبت داده شود را در خود ذخیره کند. در هر مرحله متغیر با مقادیر ممکن کمتر انتخاب می‌شود تا مقداری به خود بگیرد. این فرایند منجر به کاهش فاکتور انشعاب خواهد شد. بعبارت دیگر این کشف کننده تلاش می‌کند فاکتور انشعاب را در انتخاب های بعدی توسط انتساب یک مقدار به متغیری که [illegible] متغیرهای انتساب نشده و در بیشترین تعداد محدودیت درگیر است، کاهش دهد.

از سوی دیگر متغیر با محدودیت حداقل، تلاش می‌کند که مقداری را انتخاب کند تا کوچکترین تعداد مقادیر در متغیرهایی که به متغیر جاری توسط محدودیت‌ها مرتبط هستند، را رد کند.

مسائل ارضاء محدودیت می‌توانند توسط روش‌های اصلاح تکراری با استفاده از مقدار دادن به تمام متغیرها و سپس به کار بردن عملگرهای تغییر به منظور حرکت دادن ساختار به طرف یک راه‌حل حل شوند. عملگرهای تغییر به سادگی یک مقدار متفاوت را به یک متغیر نسبت می‌دهند. برای مثال در مسئله هشت وزیر، یک حالت اولیه تمام ۸ وزیر را روی صفحه دارد و یک عملگر وزیری را از یک جانب به جانب دیگر حرکت می‌دهد.

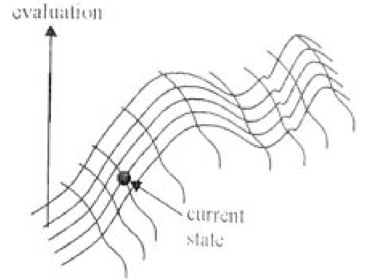
الگوریتم‌هائی که CSP راحل می‌کنند ، روش‌های تصحیح کشف کنندگی نامیده می‌شوند. زیرا آنها [illegible] قضات را در ساختار جاری مسئله، اصلاح می‌کنند. در انتخاب مقادیر جدید برای یک متغیر، واضح‌ترین کشف کنندگی انتخاب مقداری است که کمترین مقدار تناقضات را با دیگر متغیرها نتیجه دهد.

### ۴- ۲- الگوریتم‌های اصلاح تکراری (Iterative improvement)

در فصل قبل دیدیم که مسائل شناخته شده بسیاری (مانند ۸ وزیر) دارای این مشخصه هستند که شرح حالت، خودش شامل تمام اطلاعات مورد نیاز برای مسئله است.

مسیری که توسط آن به راه‌حل می‌رسیم، نامربوط است. در این موارد، الگوریتم‌های اصلاح تکراری [illegible] راهکارهای عملی‌تری را بیان می‌کنند. برای مثال، ما با ۸ وزیر آغاز می‌کنیم وزیرها را روی صفحه حرکت می‌دهیم و سعی داریم که تعداد برخوردها را به حداقل برسانیم. عقیده کلی بر این است که با یک ساختار کامل آغاز کرده و تغییراتی را به منظور اصلاح کیفیت این ساختار انجام دهیم.

بهترین راه‌برای فهم الگوریتمهای اصلاح تکراری در نظر داشتن تمام حالاتی است که روی سطح یک دورنمایی در معرض دید قرار داده شده است. ارتفاع هر نقطه در دورنما مطابق با تابع ارزیاب حالت آن نقطه است (شکل ۱۵) ایده اصلاح تکراری، حرکت کردن در اطراف دورنما و سعی بر یافتن قله‌های مرتفع است، که همانا راه‌حلهای بهینه هستند. الگوریتمهای اصلاح تکراری روشی برای مسائل عملی سخت است.




الگوریتمهای اصلاح تکراری به دو گروه اصلی تقسیم می‌شوند. الگوریتم‌های تپه نوردی (Hill-climbing) (یا gradient descent در صورتی که ما به تابع ارزیاب به عنوان یک هزینه بجای کیفیت نگاه کنیم) همیشه سعی بر ساخت تغییراتی دارند که حالت جاری را صلاح کند. الگوریتم‌های Simulated Annealing بعضی اوقات می‌توانند تغییراتی را ایجاد کنند که حداقل به طور موقت، مشکلات را بدتر سازند.

### جستجوی تپه نوردی (Hill-climbing)

جستجوی تپه نوردی درشکل زیر نشان داده شده است و آن حلقه ساده‌ای است که به طور پیوسته در جهت افزایش مقدار حرکت می‌کند. الگوریتم شامل یک درخت جستجو نیست، بنابراین ساختار دادهٔ گره فقط رکورد حالت و ارزیابی آن را نیاز دارد، که ما آن را VALUE می‌نامیم. یک اصلاح خوب این است زمانی که بیش از یک فرزند خوب برای انتخاب وجود دارد، الگوریتم بتواند به طور تصادفی از میان آنها یکی را انتخاب کند. این سیاست ساده، سه زیان عمده دارد.

```
Function HILL-CLIMBING(problem)returns a solution state
Inputs: problem, a problem
Local variables: current, anode
          next, a node
current ← MAKE – NODE(INTIAL – STATE[problem])
Loop do
   next ← a highest – valued successor of current
   if VLUE[next] < VALUE[current] then return current
   current ← next
end
```



- Local Maxima: یک ماکزیمم محلی، برخلاف ماکزیمم عمومی، قله‌ای است که پایین‌تر از بلندترین قله در فضای حالت است. زمانی که روی ماکزیمم محلی هستیم، الگوریتم توقف خواهد نمود، اگر چه راه‌حل نیز ممکن است دور از انتظار باشد.
- Plateaux: یک فلات محوطه‌ای از فضای حالت است که تابع ارزیاب یکنواخت باشد. جستجو یک قدم تصادفی را برخواهد داشت.
- Ridges: نوک کوه، دارای لبه‌های سراشیب است. بنابراین جستجو به بالای نوک کوه به آسانی می‌رسد، اما بعد با ملایمت به سمت قله می‌رود. مگر اینکه عملگرهایی موجود باشند که مستقیماً به سمت بالای نوک کوه حرکت کنند، جستجو ممکن است از لبه‌ای به لبه دیگر نوسان داشته باشد و پیشرفت کمی را حاصل شود.

در هر مورد، الگوریتم به نقطه‌ای می‌رسد که هیچ پیشرفتی نیست. اگر این اتفاق بیفتد، تنها کار ممکن برای انجام دادن آغاز مجدد از نقطه شروع دیگری دوباره آغاز می‌شود. تپه‌نوردی با شروع تصادفی (Random-restart hill-climbing) این عمل را انجام می‌دهد: آن یک سری از جستجوهای تپه نوردی را از حالات اولیه تولید شده تصادفی رهبری کرده، و هر کدام از آنها را تا زمانی که متوقف شود و یا هیچ پیشرفت قابل ملاحظه‌ای را ایجاد نکنند، اجرا می‌کند و بهترین نتایج یافته شده توسط هر جستجویی را ذخیره شده برای یک تعداد مشخص تکرار، اصلاح نشده باشد.

روشن است که، اگر تکرارهای کافی وجود داشته باشد، تپه نوردی با شروع تصادفی بالاخره بهترین راه‌حل را پیدا خواهد کرد. موفقیت hill-climbing خیلی به ظاهر فضای حالت «سطح» بستگی دارد. اگر فقط ماکزیمم‌های محلی کمی وجود داشته باشد، تپه نوردی با شروع تصادفی خیلی سریع راه‌حل خوبی را پیدا خواهد کرد. یک مسئله واقع‌گرایانه دارای سطحی است که بسیار شبیه یک جوجه تیغی است. اگر مسئله NP-complete باشد، سپس در تمام احتمالات ما نمی‌توانیم بهتر از زمان نمایی عمل نمائیم. باید یک تعداد نمایی عمل نمائیم. باید یک تعداد نمایی ماکزیمم محلی برای گیر افتادن وجود داشته باشد. معمولاً، بهرحال، یک راه‌حل خوب و منطقی پس از چند تکرار می‌تواند پیدا شود.

### Simulated annealing

به جای شروع دوباره به طور تصادفی زمانی که در یک ماکزیمم محلی گیر افتادیم، می‌توانیم اجازه دهیم که جستجو چند قدم به طرف پایین بردارد تا از ماکزیمم محلی فرار کند. این ایده‌ای از annealing شبیه‌سازی شده است. حلقه میانی annealing شبیه‌سازی شده کاملاً شبیه به تپه نوردی است. به جای انتخاب بهترین حرکت، یک حرکت تصادفی را انجام می‌دهد.

```
Function SIMULATED-ANNEALING(problem,schedule)returns a solution state
Inputs: 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(INITAL – STATE[problem])
```

```
for t ← 1 to ∞ do
T ← schedule[t]
if T = 0 then return current
next ← a randomly selected successor of current
ΔE ← VALUE[next] − VALUE[current]
if ΔE > 0 then current ← next
else current ← next only with probability e^ΔE/T
```

اگر حرکت واقعاً شرایط را بهبود بخشد، آن حرکت همیشه اجرا می‌شود. علاوه بر آن، الگوریتم حرکت را با احتمالی کمتر از ۱ انجام می‌دهد. احتمال به صورت نمایی با «بدی» حرکت کاهش می‌یابد (میزان ΔE) که نشان می‌دهد ارزیابی بدتر شده است. پارامتر دوم T برای تعیین احتمال استفاده می‌شود. در مقادیر بالای T، حرکت بد بیشتر اجازه عمل می‌یابند. در حالی که اگر T به سمت صفر برود، این حرکات بد محدود می‌شوند. تا زمانی که الگوریتم کم و بیش مشابه تپه نوردی عمل کند. جدول ورودی به عنوان تابعی که تعداد چرخه‌هایی که قبلاً ... شده‌اند را در خود ذخیره می‌کند، مقدار T را تعیین می‌کند.

خواننده اکنون باید ... باشد که نام "Stimulaed annealing" و پارامترهای $T, \Delta E$ به دلیل خوبی انتخاب شده‌اند. الگو... ... صریحی با annealing (پردازشی که به طور آهسته مایعی را تا زمانی که یخ ببندد سرد می‌کند) گسترش یافته است. مقدار تابع مطابق با انرژی ورودی اتمهای ماده است، و T با دما مطابقت دارد. جدول میزان دما را در جایی که پایین آمده است، تعیین می‌کند.

حرکات اختصاصی بواسطهٔ انتشار حرارت در فضای حالت به صورت ترقی و تنزل‌های تصادفی هستند. می‌توان ثابت نمود که اگر دما به اندازه کافی و به آرامی پایین آورده شود، ماده به پایین‌ترین ساختار سطح انرژی می‌رسد. این امر مانند این است که بگوییم اگر جدول T را به اندازه کافی پایین آورد، الگوریتم نقطه بهینه عمومی پیدا خواهد نمود.

### ۵-۲- استراتژی تولید و آزمون (generate and test)

این استراتژی در برخی از منابع ذکر شده و چون در برخی پرسش‌های آزمون‌های سراسر و فناوری اطلاعات عنوان شده، شرح داده می‌شود.

استراتژی تولید و آزمون در واقع مشابه با DFS است، چرا که در این روش حرکت تا عمق ممکن برای رسیدن به اولین جواب ادامه پیدا می‌کند. این استراتژی ابتدائی در سه مرحله انجام می‌شود:

۱- تولید پاسخ ممکن: برای برخی مسائل این به معنی بسط یک گره جدید در گراف فضای حالت است و برای برخی دیگر، به معنی تولید مسیری از گره آغازین خواهد بود.

۲- آزمون اینکه آیا پاسخ یافت شده (یا بعبارت ساده‌تر مسیر یافت شده) تنها یک بن بست است و یا کلیه شرایط پاسخ را نیز دارد.

۳- اگر راه‌حل مسئله پیدا شده که به اتمام حل رسیده‌ایم، در غیر اینصورت به مرحله یک برگرد.



اگر تولید پاسخ‌های ممکن بر مبنای روش سیستماتیک ایجاد شود، در این صورت این روش به شرط متناهی بودن گراف فضای حالت اگر پاسخی وجود داشته باشد آنرا پیدا خواهد کرد. اگر چه معمولاً مسائل کاربردی واقعی گراف فضای حالت بسیار بزرگی دارند و تحت این شرایط در واقع گراف نامتناهی خواهد بود و رسیدن به پاسخ ممکن است به زمان غیر قابل قبولی نیاز داشته باشد. روش تولید آزمون بجای جستجوی سیستماتیک می‌تواند بصورت تصادفی به جستجو بپردازد که تحت این شرایط الگوریتم موزه بریتانیایی نامیده می‌شود. در بین این دو روش سیستماتیک و موزه بریتانیایی حالت مابینی وجود دارد که به جای جستجوی کل فضا، از بین فرزندان و به کمک تابع مکاشفه‌ای تنها بخشی از فرزندان مورد جستجو قرار گیرد. این کار در واقع به معنی استفاده از استراتژی DFS و تابع مکاشفه‌ای بصورت توأمان است. حاصل کار ممکن است منجر به تولید جواب شود ولی مسلماً فضای جستجو بشدت تقلیل خواهد یافت. نباید فراموش کرد روش تپه‌نوردی که قبلاً به آن اشاره شد، در واقع نوعی از روش و آزمون محسوب می‌شود. با این تفاوت که در روش تپه‌نوردی پس از مرحله آزمون بازخوردی برای انتخاب بعدی تولید خواهد شد.

## تست‌های فصل دوم

۱- روش جستجوی *A، تحت چه شرایطی یافتن پاسخ بهینه را تضمین می‌کند؟ (سراسری)

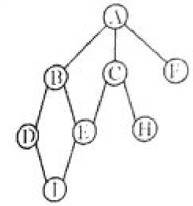
۱) اصلاً روش‌های ابتکاری (heuristic) از جمله *A قادر به یافتن پاسخ بهینه نیستند.

۲) شرط لازم برای اینکه روش *A پاسخ بهینه را تضمین کند، به دامنه مسئله بستگی دارد.

۳) در صورتی که تابع ابتکاری (heuristic function) مورد استفاده، فاصله وضعیت‌های مختلف تا وضعیت هدف را، هرگز بیش‌تر از، هرگز بیش تر از مقدار واقعی تخمین نزند.

۴) در صورتی که تابع ابتکاری مورد استفاده، فاصله وضعیت‌های مختلف تا وضعیت هدف حداکثر به اندازه مقدار کوچک دلتا و بیشتر از مقدار واقعی تخمین بزند.

۲- اگر در گراف زیر جستجو در عمق (Deth First Search) را از رأس C شروع کنیم، کدام گره‌ها بترتیب از چپ به راست رویت (visit) می‌شوند؟ (فرض کنید فرزندان یک گره بر اساس ترتیب حروف الفبا انتخاب شوند) (سراسری)




۱) ABCDEFHI

۲) CABDIEFH

۳) CAEHBFID

۴) CABDEHIF

۳- شرط پذیرش (admissible) بودن یک الگوریتم برای یافتن جواب مسأله کدام است؟ (سراسری)

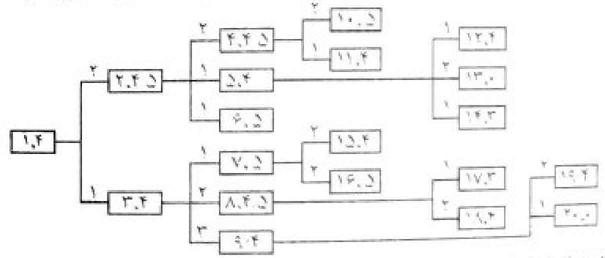
۱) $\exists n\; h(n) \le h^*(n), g(n) \ge g^*(n)$

۲) $\exists n\; h(n) \ge h^*(n), g(n) \ge g^*(n)$

۳) $\forall n\; h(n) \le h^*(n), g(n) \ge g^*(n)$

۴) $\forall n\; h(n) \le h^*(n), g(n) \le g^*(n)$

۴- در درخت تصمیم‌گیری زیر با استفاده از جستجوی *A کدام گزینه شماره‌گره‌های مورد بررسی را مشخص می‌کند؟ توجه کنید که هزینه هر گره در کنار شماره آن و هزینه هر شاخه روی آن نوشته شده است؟ (در هر گره اولین عدد شماره‌گره و دومین عدد هزینه می‌باشد) (سراسری)




۱) ۱و۲و۳و۴و۵و۶و۱۲و۱۳و۱۴

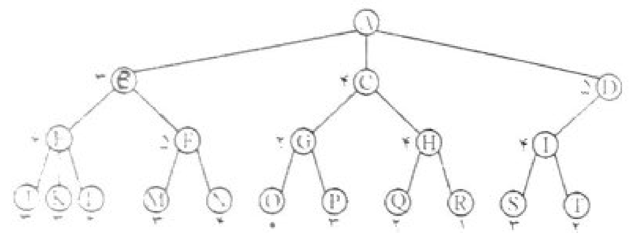
۲) ۱و۲و۳و۷و۸و۹و۱۹و۲۰

۳) ۱و۲و۳و۷و۸و۹و۱۷و۱۸و۱۹و۲۰

۴) ۱و۲و۳و۷و۸و۹و۴و۵و۶و۱۲و۱۳و۱۴



۵- در درخت جستجوی زیر به شرطی که گره O، گره هدف باشد. بر اساس الگوریتم جستجوی Best-First، ترتیب دیدن گره‌ها کدام است؟ (سراسری – 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

۶- آیا روش جستجوی تولید و آزمون (generate and test) یک روش اکتشافی (heuristic) محسوب می‌شود و یا اینکه روشی است سیستماتیک (systematic)؟ (سراسری –۸۴)

۱) اکتشافی

۲) سیستماتیک

۳) بستگی به مرحله آزمون دارد

۴) بستگی به مرحله تولید دارد

۷- نقطه ضعف روش *IDA (Iterative depending A*) در چیست؟ (سراسری ۸۴)

۱) کامل نبودن

۲) دوباره کاری

۳) کارائی پائین

۴) مصرف حافظه زیاد

۸- می‌خواهیم با استفاده از روش جستجوی *A پاسخ بهینه (optimal) مسأله‌ای را بیابیم. با فرض اینکه هر یک از سه تابع ابتکاری $h_1, h_2, h_3$ برای این منظور قابل استفاده باشد. کدام یک از توابع ترکیبی زیر نیز برای یافتن حل بهینه‌ی مسأله قابل استفاده خواهد بود؟ (سراسری ۸۳)

۱) $h_1 + h_2 + h_3$

۲) $\frac{h_1 + h_2 + h_3}{3}$

۳) $h_1 \times h_2 \times h_3$

۴) $\sqrt{h_1 \times h_2 \times h_3}$

۹- کدامیک از موارد زیر در خصوص روش جسجوی *RTA (Real-Time A*) در مقایسه با روش *A صحیح تر است؟ (سراسری ۸۳)

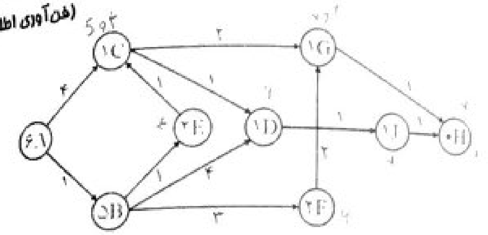
۱) *RTA اغلب تمایل بیشتری به ادامه مسیر جاری دارد.

۲) *RTA همواره مسیرهای کوتاهتری را می‌یابد.

۳) *RTA اغلب تمایل کمتری به ادامه مسیر جاری دارد

۴) *RTA همواره مسیرهای طولانی‌تری را دارد

۱۰- الگوریتم *A در گراف زیر کدامیک از مسیرهای زیر را به عنوان پاسخ می‌یابد؟ (اعداد روی یال‌ها هزینه واقعی هر گره هزینه تخمینی آن گره تا گره‌ی هدف است) (A گره شروع و H گره هدف است) (فن‌آوری اطلاعات ۸۴)



۱) ACGH
۲) ACDJH
۳) ABFGH
۴) ABCDJH

۱۱- در صورت استفاده از روش تپه نوری از طریق تندترین شیب (steepest ascend hill clinbing) در حل مسئله قبل (سئوال ۱۰) کدام یک از گزینه‌های زیر صحیح تر است؟ (فن‌آوری اطلاعات ۸۴)

۱) جستجو به فلات برخورد می‌کند.
۲) زمان حل مسئله کاهش خواهد یافت.
۳) جستجو به ماکزیمم محلی برخورد می‌کند.
۴) تغییری در مسیر حل مسئله صورت نمی‌گیرد.

۱۲- روش جستجوی تولید آزمون (generate & test) برای حل کدامیک از مسائل زیر مناسب‌تر است؟ (فن‌آوری اطلاعات ۸۴)

۱) شطرنج
۲) معمای هشت (eight puzzle)
۳) هشت وزیر (eight Queen)
۴) فروشنده دوره گرد (travelling salesman)

۱۳- کدامیک از گفته‌های زیر در مورد روش جستجوی عمق اول با تعمیق مکرر (Iterative deepeniling DFS) صحیح است؟ (فن‌آوری اطلاعات ۸۴)

(منظور از کامل complete و از قابل قبول admissible است)

۱) کامل است - قابل قبول نیست.
۲) کامل نیست - قابل قبول است.
۳) کامل نیست - قابل قبول نیست.
۴) کامل است - قابل قبول است.

۱۴- اگر بخواهیم کلید پاسخ‌های یک مسئله ارضاء محدودیت (Constraint satisfaction) را بیابیم کدامیک از روش‌های جستجوی زیر مناسب ترین است؟ (فن‌آوری اطلاعات ۸۴)

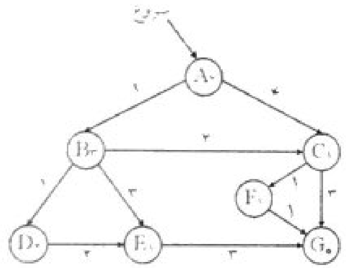
۱) عمق اول ۲) سطح اول ۳) تپه نوردی ۴) *A



۱۵- فرض کنید می‌خواهیم الگوریتم *A را برای جستجو بکار ببریم و سه هیوریستیک $H_1, H_2, H_3$ موجودند که همگی قابل پذیرش هستند و برای تمام وضعیت‌ها داریم $H_3 > H_1 > H_2$ کدامیک از عبارات زیر درست است؟ (سراسری ۸۴)

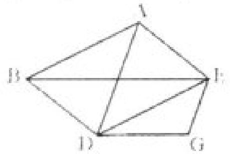
۱) مسیر بهینه فقط از بکارگیری $H_2$ حاصل می‌شود اما $H_1$ از $H_2$ بهتر است
۲) با هر کدام از هیوریستیک‌های فوق مسیر بهینه حاصل می‌شود اما $H_2$ مسیر ارزانتری را پیدا می‌کند
۳) با هر کدام از هیوریستیک‌های فوق مسیر بهینه حاصل می‌شود اما $H_2$ آنرا با کمترین بسط‌ها پیدا می‌کند.
۴) راه‌حل های حاصل از هیوریستیک‌ها از نظر بهینگی به همان ترتیب $H_3 > H_1 > H_2$ هستند

۱۶- مسیر یافت شده توسط الگوریتم *A برای گراف مقابل چیست؟ (سراسری ۸۴)



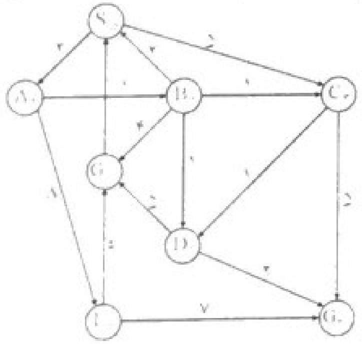
۱) ABCFG
۲) ACG
۳) ABCG
۴) ABGE

۱۷- در شکل مقابل (A نقطه شروع و G هدف است) حاصل جستجو با کدام روش به مسیر ABDG است؟ (فن‌آوری اطلاعات ۸۴)



۱) جستجوی عرض اول
۲) جستجوی عمق اول
۳) جستجوی *A
۴) جستجوی تپه‌نوردی

۱۸- در گراف زیر حاصل جستجو با روش *A کدام است؟ (نقطه شروع : S. اعداد روی یال ها و اعداد داخل گره‌ها هزینه تخمینی گره تا هدف است) (فن‌آوری اطلاعات ۸۴)



۱) S-B-G۱
۲) S-A-E-G۲
۳) S-A-E-G۱
۴) S-A-B-D-G۲

۱۹- هرگاه در یک مسئله به دنبال کلیه جواب ها باشیم، کدامیک از روش‌های زیر مناسبتر است؟ (فن آوری اطلاعات ۸۴)

۱) $A^*$ ۲) عمق اول ۳) تپه‌نوردی ۴) تولید و آزمون

۲۰- کدام یک از روش‌های جستجوی زیر برای مسئله قرارگیری (placement) تعدادی ساختار (module) در یک محدوده یا فضای خاص که کمترین فضا را اشغال می‌کنند مناسب‌تر است (همانند قراردهی تعدادی IC بر روی یک تخته مدار چاپی که مایل هستیم کمترین سطح تخته بکار برده شود)؟ (مکاترونیک ۸۴)

۱) جستجوی $A^*$

۲) جستجوی عمق سخت

۳) جستجوی تپه‌نوردی (hill climbing)

۴) جستجوی سرد شدن شبیه‌سازی شده (simulated annealing)

۲۱- کدام یک از موارد زیر در مورد مقایسه دو روش جستجوی تپه‌نوردی ساده و تپه‌نوردی از تندترین شیب صحیح است؟ (فن آوری اطلاعات - ۸۵)

۱) تپه‌نوردی ساده کمتر در ماکزیمم محلی قرار می‌گیرد

۲) تپه نوردی ساده با سرعت بیشتری حرکت می‌کند. اما مسیر طولانی‌تری را می‌یابد.

۳) تپه‌نوردی ازتندترین شیب با سرعت بیشتری حرکت می‌کند اما حافظه بیشتری نیز مصرف می‌کند

۴) تپه‌نوردی از تندترین شیب پاسخ بهینه را می‌یابد، در حالی که تپه‌نوردی ساده اینطور نیست



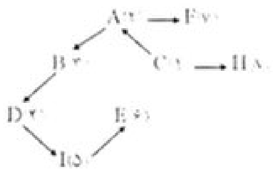
## پاسخ تست‌های فصل دوم

۱- گزینه «۳» صحیح است.

بدیهی است، طبق فرمول $\forall n\ h(n) \le h^*(n)$ که قبلاً مطرح شد، الگوریتم $A^*$ خواهد بود و تفسیر این رابطه آنست که فاصله هر وضعیت اختیاری تا هدف که تخمین زده شده ($h(n)$) هرگز از مقدار بهینه آن $h^*(n)$ بزرگتر نباشد.

۲- گزینه «۲» صحیح است.

شماره‌های کنار هر گره معرف توالی ملاقات از رأس C و به ترتیب ملاقات فرزندان بر حسب الفبااست.




۳- گزینه «۳» صحیح است.

بدیهی است، به متن درس مراجعه شود. توجه کنید طبق تعریف همواره $g(n) \ge g^*(n)$ چرا که $g^*(n)$ کوتاهترین مسیر از مبدأ تا گره n بوده و کوتاهتر از آن مسیری وجود ندارد.

۴- گزینه «۴» صحیح است.

| Open | closed |
|---|---|
| ۱ | |
| $۳_{۵}، ۲_{۶/۵}$ | ۱ |
| $۲_{۶/۵}، ۷_{۷}، ۸_{۷/۵}، ۹_{۸}$ | ۱،۳ |
| $۵_{۷}، ۷_{۷}، ۸_{۷/۵}، ۶_{۸}، ۹_{۸}، ۴_{۸/۵}$ | ۱،۳،۲ |
| $۱۳_{۵}، ۷_{۷}، ۱۴_{۷}، ۸_{۷/۵}، ۶_{۸}، ۹_{۸}، ۱۲_{۸}، ۴_{۸/۵}$ | ۱،۳،۲،۵ |

دقت کنید، در این راه‌حل هر گره به همراه کلیه فرزندان در پاسخ درنظر گرفته شده است.برای مثال اگر گره ۱ بسط داده شده، فرزندان آن یعنی ۲ و ۳ نیز بلافاصله ذکر شده‌اند. پس در واقع پاسخ ۱،۳،۲،۵ خواهد بود که بصورت گزینه ۴ (با افزودن فرزندان هر گره بلافاصله بعداز آن) در خواهد آمد.

۵- گزینه «۱» صحیح است.

| Open | closed |
|---|---|
| A | |
| $B_{۴}، C_{۵}، D_{۶}$ | A |
| $C_{۵}، D_{۶}، F_{۷}، E_{۸}$ | A.B |
| $G_{۴}، H_{۶}، D_{۶}، F_{۷}، E_{۸}$ | A.B.C |
| $O_{۴}، P_{۵}، H_{۶}، D_{۶}، F_{۷}، E_{۸}$ | A.B.C.G |

این پاسخ شامل ملاقات بترتیب A.B.C.G.O است که اگر فرزندان هرگره را بلافاصله از ملاقات هر گره به آن اضافه کنیم (همانند تست ۴) پاسخ ۱ درست خواهد بود.

۶- گزینه «۴» صحیح است.

در متن درس توضیح داده می‌شد که روش تولید و آزمون هم می‌تواند سیستماتیک و هم می‌تواند ابتکاری باشد. عامل مهم در زمان مرحله تولید اتفاق می‌افتد که طی آن تصمیم‌گیری می‌شود تا کدام گره‌ها توسعه یابند.

۷- گزینه «۲» صحیح است.

به متن درس مراجعه شود. روش $IDA^*$ کامل وبهینه است و مصرف حافظه را کاهش می‌دهد. اما مجبور به حل مکرر مسئله در عمق‌های متفاوت خواهد بود و به همین علت دوباره کاری در آن صورت می‌گیرد.

۸- گزینه «۲» صحیح است.

اگر تابع ابتکاری $h_1, h_2, h_3$ را قابل پذیرش بدانیم، ترکیب آنها نیز باید قابل پذیرش باشد. شرط پذیرش آن بود که $\forall n\ h(n) \le h^*(n)$ باشد. حال ترکیب این سه تابع نیز باید این ویژگی را داشته باشد. گزینه ۲ متوسط‌گیری بین این سه تابع را پیشنهاد نموده که صحیح است. جمع یا ضرب آنها در هم می‌تواند شرط مذکور را نقض کند. همچنین گزینه ٤ نیز می‌تواند این شرط را نقض کند.

۹- گزینه «۱» صحیح است.

به متن درس و این روش مراجعه شود. نباید فراموش کرد که روش $RTA^*$ در مورد کامل و بهینه بودن تضمینی ارائه نمی‌دهد.

۱۰- گزینه «۴» صحیح است.

| Open | closed |
|---|---|
| $A_{۶}$ | |
| $C_{۵}, B_{۶}$ | $A_{۶}$ |
| $B_{۶}, D_{۶}, G_{۷}$ | $A_{۶}, C_{۵}$ |
| $E_{۴}, D_{۶}, F_{۶}, G_{۷}$ | $A_{۶}, C_{۵}, B_{۶}$ |
| $C_{۴}, D_{۶}, F_{۶}, G_{۷}$ | $A_{۶}, B_{۶}, E_{۶}$ |
| $D_{۵}, F_{۶}, G_{۷}$ | A.B.C.E |
| $J_{۶}, F_{۶}, G_{۷}$ | A.B.C.D.E |
| $H_{۶}, F_{۶}, G_{۷}$ | A.B.C.D.E.J |



۱۱- گزینه «۱» صحیح است.

قبلاً در مورد روش تپه نوردی توضیح داده شد. تپه نوردی از طریق تند ترین شیب، از بین گره‌های مجاور وضعیت جاری بهترین گره در راستای شیب تعریف شده در مسئله را پیدا خواهد کرد. در صورتی که تپه‌نوردی عادی اولین گره بهتر را انتخاب می‌کند.

در مثال فوق، اگر انتخاب مناسب لبه‌ی (شیب)، انتخا[illegible] هزینه باشد، دراین صورت بر روی مسیری یالبه‌های وزن یک به سوی مقصد حرکت خواهیم کرد. این به معنی برخورد به سطح صاف (فلات) است.

۱۲- گزینه «۳» صحیح است.

امکان حل مسئله شطرنج و فروشنده دوره گرد بکمک روش‌های غیر هوشمند وجود ندارد. حتی اگر تابع مکاشفه‌ای هم به استراتژی DFS افزوده شود، بعلت عمق زیاد گراف فضای حالت، امکان حصول جواب وجود ندارد. در مسئله معمای هشت نیز بعلت وجود حلقه و عمق زیاد گراف، روش تولید و آزمون نمی‌تواند به جواب مناسب برسد ولی برای مسئله هشت وزیر بعلت محدودیت عمق گراف (هشت)، این روش می‌تواند پاسخگو باشد.

۱۳- گزینه «۴» صحیح است.

بدیهی است. به متن درس مراجعه شود.

۱۴- گزینه «۱» صحیح است.

بدیهی است. به متن درس مراجعه شود.

۱۵- گزینه «۳» صحیح است.

بدیهی است. به متن درس مراجعه شود.

۱۶- گزینه «۱» صحیح است.

| Open | closed |
|---|---|
| $A_{۲}$ | |
| $B_{۴}, C_{۵}$ | $A_{۲}$ |
| $C_{۴}, D_{۴}, E_{۵}$ | $A_{۲}, B_{۴}$ |
| $D_{۴}, F_{۵}, E_{۵}, G_{۶}$ | $A_{۲}, B_{۴}, C_{۴}$ |
| $F_{۵}, E_{۵}, G_{۶}$ | $A_{۲}, B_{۴}, C_{۴}, D_{۴}$ |
| $G_{۵}, E_{۵}$ | $A_{۲}, B_{۴}, C_{۴}, D_{۴}, F_{۵}$ |

پس مسیر بهینه گزینه ۱ خواهد بود.

۱۷- گزینه «۲» صحیح است.

واضح است. دنباله ABCDG تنها از جستجوی عمقی حاصل خواهد شد. دنباله جستجوی عرض اول ABEDG است و برای تپه‌نوردی و $A^*$ صورت مسئله ناقص است.

۱۸- گزینه «۴» صحیح است

| Open | Closed |
| --- | --- |
| $S_5$ | |
| $A_4 C_8$ | $S_5$ |
| $B_4 C_8 E_{16}$ | $S_5 A_4$ |
| $D_5 G1_7 C_7 E_{16}$ | $S_5 A_4 B_4$ |
| $G2_6 C_7 G1_7 E_{16}$ | $S_5 A_4 B_4 D_5$ |

مسیر بهینه $S-A-B-D-G2$

۱۹- گزینه «۴» صحیح است

بدیهی است به متن درس مراجعه شود. جستجوی عمقی تضمینی در مورد رسیدن **به جواب بهینه** می‌دهد روش $A^*$ معمولاً برای یافتن یکی از جواب های بهینه طراحی شده است. **روش تپه‌نوردی** که اصولاً ضمانتی برای رسیدن به جواب بهینه ندارد.

۲۰- گزینه «۴» صحیح است.

مسئله جایگذاری (قرارگیری) یکی از مشهورترین مثال‌هائی است که **توسط سردشدن شبیه‌سازی** شده حل شده است و نسبت به دیگر روش‌ها در حل این مسئله **موفق‌تر بوده است.**

۲۱- گزینه «۲» صحیح است

بدیهی است به حل تست ۱۱ توجه کنید. تپه‌نوردی ساده چون به اولین مجاور **بهتر برسد، آنرا** انتخاب خواهد کرد، سپس سریعتر حرکت می‌کند ولی احتمالاً به جواب **طولانی‌تر خواهد رسید.**

# فصل سوم

# جستجوی رقابتی

هدف از جستجوی رقابتی، یافتن پاسخ در یک گراف فضای حالت است، اما در این شرایط رقیب یا رقبائی نیز وجود دارند که می‌خواهند به پاسخ برسند در اینجا مهم نیست که آیا کوتاهترین راه‌حل برای رسیدن به پاسخ را یافته‌ایم یا خیر بلکه مهم آنست قبل از رقبا به هدف برسیم.

بدلیل این تغییر در صورت مسئله، راه‌حل اگر چه شباهت کلی با روش‌های جستجو در گراف فضای حالت دارد ولی تا حدودی متفاوت خواهد بود. قبل از آنکه بخواهیم به روش حل این دسته از مسائل بپردازیم البته لازم است تا نگاهی به انواع این دسته از مسائل و محدودیت‌های مطرح شده در راه‌حل پیشنهادی داشته باشیم.

بازی شطرنج نمونه بسیار خوبی برای بیان روش‌های جستجوی رقابتی سنتی است. در بازی شطرنج متوسط فاکتور انشعاب ۳۵ تخمین زده می‌شود و معمولاً یک بازی تا ۵۰ حرکت برای هر بازیکن ادامه خواهد داشت. پس اگر گره‌های موجود در گراف فضای حالت را بخواهیم تخمین بزنیم، در حدود ۳۵ گره یا وضعیت در بازی شطرنج می‌تواند وجود داشته باشد، اما بازی ساده‌ای همانند tic – tac – toe تنها !۹ حالت را شامل می‌شود. به همین دلیل شطرنج نیاز به تفکر عمیق دارد در حالی که بازی tic – tac – toe (یا o-x) بازی بسیار ساده‌ای است. یکی از کاربردهای عمده جستجوی رقابتی، استفاده از آن در بازیهای فکری است، اما آنچه که در این بخش بیان می‌شود شامل گروه محدودی از بازیها خواهد شد. ببینیم این محدودیت در نظر گرفته شده دربازی چه هستند؟

اولین نکته آنست که بازی ۲ نفره (در حالت خاص چند نفره) در نظر گرفته می‌شود، یعنی تیم‌خودی و رقیب در کار نخواهد بود. نوبت در بازی بر اساس قوانین بازی بین ۲ بازیکن خودی و رقیب جابجا خواهد شد بازیکن در هر لحظه قادر است تمام صفحه بازی را مشاهده کند و بعبارت دیگر هیچ اتفاقی در بازی برای دو بازیکن پوشیده نیست برد، باخت و مساوی در بازی طبق قوانین روشنی تعریف شده است. عامل تصادف در بازی ( همانند، سکه، ورق، تاس) وجود ندارد، البته می‌توان این تئوری را بازی‌های تصادفی نیز توسعه داد که در انتها به آن پرداخته شده است.

آنچه گفته شد بخش مهمی از محدودیت‌های لازم در بازی مورد نظر است. بازی شطرنج مثال خوبی از انواع بازی است که تمام این محدودیت‌ها را رعایت می‌کند ولی دقیقاً بازی فوتبال در نقطه قابل مقابل قرار دارد و الگوریتم‌های مطرح شده در این بخش برای روبات فوتبالیست کاربردی ندارد.