فصل دوم

حل مسائل از طریق جستجو

در این فصل نوعی از عاملهای هدف گرا که عامل حلال مسئله نام دارد تشریح می شوند. عاملهای حل مسئله بر پایهٔ یافتن دنباله ای از عملیات که به حالات مطلوب منجر می شود، و کارایی را ماکزیمم می کند تصمیم گیری می کنند.

فرموله سازی هدف بر مبنای حالت جاری و اندازهٔ کارایی عامل، اولین قدم حل مسئله است. این دسته از عاملها در ابتدا برایشان یک هدف تعریف می شود و بعد از یک حالت start شروع می کنند. عامل با چندین انتخاب بینابینی و مقادیر ناشناخته می تواند تصمیم گیری کرده و آنگاه بهترین دنباله را انتخاب کند.

این فرآیند برای یافتن چنین دنباله ای **جستجو** نامیده می شود. الگوریتم جستجو مسئله ای به عنوان ورودی گرفته و **پاسخی** به شکل دنبالهٔ عمل باز می گرداند. پس از یافتن پاسخ، اعمال توصیه شده اجرا خواهند شد. این فاز اجرا نامیده می شود. بنابراین، طراحی ساده "فرموله سازی، جستجو، اجرا" را برای عامل داریم که در شکل زیر نشان داده شده است. بعد از فرموله سازی هدف و مسئله ای برای حل، عامل روال جستجویی را برای حل کردن، فراخوانی می کند. سپس از پاسخ حاصل، جهت راهنمایی اعمال استفاده می کند تا دنباله را یکی یکی اجرا و حذف کند. بعد از راه حل، عامل هدف جدیدی را فرموله می کند.

قبل از پرداختن به جزئیات، نگاه مختصری به عامل حل مسئلهٔ منطبق بر بحث عامل ها و محیط ها ، در فصل قبل داشته باشیم. طراحی عامل در شکل بالا فرض می کند محیط ایستا است، زیرا فرموله سازی و حل مسئله بدون توجه به هر تغییر ممکن در محیط انجام می شود. طراحی عامل همچنین فرض می کند که حالت آغازین شناخته شده است. شناختن آن به ویژه زمانی ساده است که محیط قابل مشاهده باشد. ایدهٔ شمارش "گروه اعمال جایگزین" فرض می کند که محیط گسسته است. در نهایت، و مهمتر از همه آنکه طراحی عامل فرض می کند محیط قطعی 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)
  is empty seq
       goal ← FORMULATE-GOAL (state)
       problem ← FORMULAE-PROBLEM (state, goal)
       seq ←SEARCH (problem)
  action ← FIRST (seq)
  seq ← REST (seq)
  return action
```

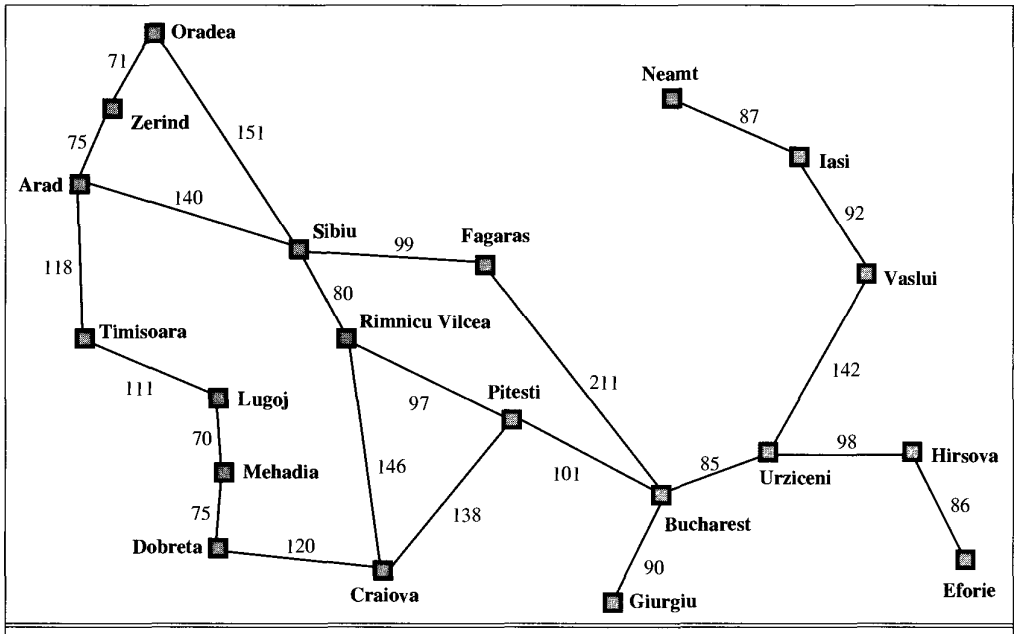
شکل ۲-۱: عامل سادهٔ حل مسئله. ابتدا هدف و مسئله را فرموله سازی می کند، دنباله ای از اعمال که مسئله را حل می کنند جستجو می کند و آنگاه اعمال را اجرا می کند. پس از تکمیل آن، هدف دیگری را فرموله سازی کرده و روی آن کار می کند. توجه کنید در حین اجرا ادراک را فراموش خواهد کرد، فرض می کند که پاسخ یافت شده همواره کار می کند.

پاسخ ها به حل اینگونه مسائل دنباله ای واحد از اعمال است، بنابراین نمی توانند رویدادهای غیر منتظره را پیش بینی کنند. به علاوه پاسخ ها بدون توجه به ادراکات اجرا می شوند. عامل برنامهٔ خود را چشم بسته

دنبال می کند. (نظریه پردازان کنترل آن را سیستم **حلقه- باز** می نامند، زیرا حذف ادراکات، حلقه بین عامل و محیط را می شکنند). تمامی این مفروضات به معنی آن است که ما با ساده ترین نوع محیط سروکار داریم.

فرموله کردن مسئله

برای تشریح مسئله از یک مثال استفاده می کنیم. فرض کنید ما در شهر آراد در کشور رومانی هستیم و از آنجا قصد رفتن به شهر بخارست را داریم. نقشه مسیرهای موجود در کشور رومانی در شکل زیر نشان داده شده است.



شکل ۲-۲: نقشه راه های کشور رومانی

فضای حالت : فضای حالت، گرافی تشکیل می دهد که در آن گره ها حالات و لبه های بین گره ها اعمال هستند. (نقشهٔ رومانی نشان داده شده در شکل می تواند به عنوان فضای حالت تفسیر شود اگر هر جاده را همان دو عمل راندن بر شماریم که هر کدام در یک جهت است) مسیر در فضای حالت، دنباله ای از حالات است که توسط دنباله ای از اعمال به هم متصل شده باشند.

گراف فضای حالت (state space graph) : به وسیله یک ۴ تائی بصورت $G=(V,E,S,D)$ نشان داده می شود که در آن:

V : مجموعه گره ها یا وضعیت گراف است.

E : مجموعه لبه های متعلق به گراف است.

S : زیر مجموعه غیر تهی از V بوده و شامل گره (گره های) آغازین می باشد.

D : زیر مجموعه غیر تهی از V بوده و شامل گره (گره های) هدف می باشد.

V یا گره های گراف وضعیت های حاکم به محیط مسئله را معین می کند، یعنی هر وضعیت پایدار نسبی در مسئله بوسیله یک گره نمایش داده می شود. لبه های گراف یا E معرف گذر از یک وضعیت (گره) به وضعیت دیگر هستند. وضعیت یا وضعیت های آغازین، منظور نقطه یا نقاطی است که از آن مسئله شروع می شود و در شرح صورت مسئله معین شده است. متقابلاً وضعیت پایانی شامل وضعیتی است که هدف نامیده می شود و در آن نقطه، راه حل به اتمام خواهد رسید (پاسخ مسئله).

نکته قابل توجه در این تعریف آنست که کدام گروه هدف شناخته می شود؟ معمولاً تعیین گره هدف از طریق دو حالت صورت می گیرد : اول آنکه تابعی معین کند که وضعیت جاری (بدون توجه به مسیر طی شده) هدف است. و دوم آنکه مسیر طی شده معین کند که به هدف رسیده ایم.

یک مسئله از طریق ۴ مولفه می تواند تعریف شود، و برای تعریف گراف فضای حالت به آنها نیاز داریم.

۱- **حالت آغازین:** حالتی که عامل از آنجا شروع می کند. برای مثال، حالت اولیه برای عامل ما در کشور رومانی می تواند In (Arad) توصیف شود.

۲- **تشریح اعمال ممکن موجود برای عامل** : رایج ترین فرموله سازی استفاده از **تابع مابعد** است. در حالت داده شدهٔ x, successor-fn(x) مجموعه ای به صورت زوج مرتب <action, successor> برمی گرداند که هر عمل یکی از اعمال قانونی در x است و هر مابعد حالتی است که می تواند از x به کمک عملی تولید شود. برای مثال، از حالت In(Arad) تابع مابعد برای مسئله رومانی می تواند مقدار زیر را برگرداند:

{<Go(Sibiu), In(Sibiu)> , <Go(Timisoara), In(Timisoara)> , <Go(zerind), In(zerind)>}

حالت اولیه و تابع مابعد به یاری هم **فضای حالت** مسئله را تعریف می کنند.

۳- **آزمون هدف:** آزمون هدف به ما می گوید که حالات داده شده می تواند حالت هدف باشد یا خیر. گاهی مجموعهٔ مشخصی از حالات هدف وجود دارد و آزمون به سادگی کنترل می کند که حالت داده شده یکی از آنهاست. هدف عامل رومانی مجموعهٔ واحد {In(Bucharest)} است. گاهی هدف از طریق خواص انتزاعی تعریف می شود تا یک مجموعهٔ شمارش پذیر واضح از حالات. برای مثال، در شطرنج هدف رسیدن به حالت "کیش مات" است که شاه رقیب تحت حمله قرار گرفته و راهی برای فرار نیز ندارد.

۴- **تابع هزینه مسیر:** این تابع هزینهٔ عددی به هر مسیر نسبت می دهد. عامل حل کنندهٔ مسئله تابع هزینه ای را انتخاب می کند که بازتابی از اندازهٔ کارایی آن باشد. برای عاملی که تلاش می کند به بخارست برسد، زمان کلیدی است و به همین علت هزینهٔ مسیر می تواند طول به کیلومتر باشد. در این فصل، فرض می کنیم که هزینه مسیر می تواند به عنوان مجموع اعمال منفرد در طول مسیر محسوب گردد. هزینهٔ مرحلهٔ انجام عمل a برای رفتن از حالت x به حالت y توسط c(x,a,y) بیان می شود. هزینهٔ مراحل برای رومانی به عنوان فواصل نشان داده شده است. فرض می کنیم که هزینهٔ مراحل غیر منفی هستند.

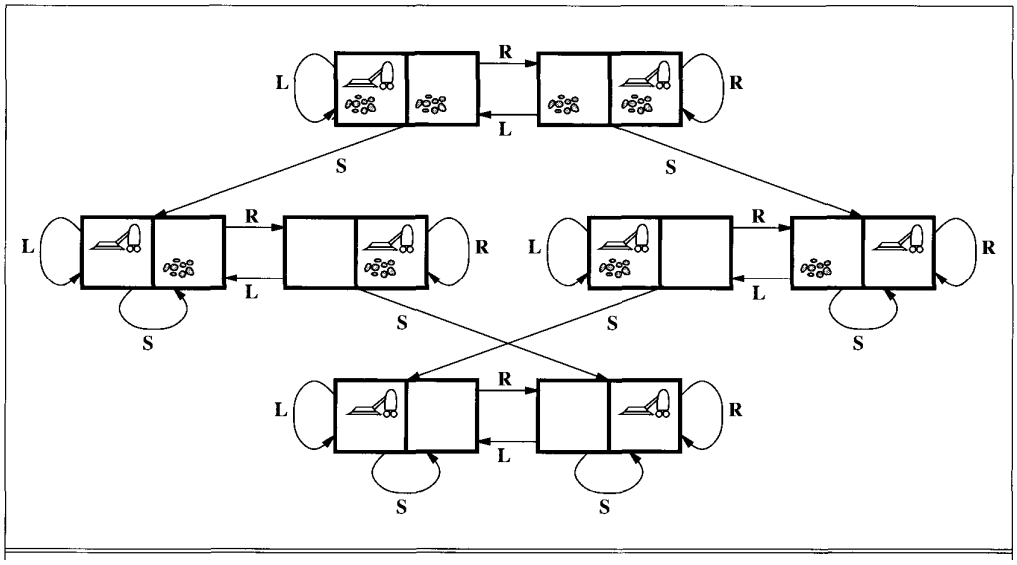
عناصر پیشین مسئله را تعریف می کنند و می توانند در کنار هم، در یک ساختار دادهٔ واحد، به عنوان ورودی، به الگوریتم حل کننده مسئله داده شوند. پاسخ به مسئله مسیری از حالت آغازین به حالت هدف

است. کیفیت پاسخ توسط تابع هزینهٔ مسیر اندازه گیری می شود و پاسخ بهینـه کمتـرین هزینـهٔ مسـیر در بین تمامی پاسخ هاست.

مسائل نمونه

- **فرموله کردن عامل مکش**

شکل زیر گراف فضای حالت را نشان می دهد. فرض کنید حسگرهای عامل اطلاعات کافی در مـورد اینکـه عامل دقیقاً در چه حالتی است را می دهد. یعنی محیط قابل دسترس می باشد. از طرفی عامل نتیجه دقیق هر یک از عمل هایش را می داند یعنی عامل دقیقاً می داند که بعد از هر رشته از عمل ها در کـدام حالـت قرار خواهد گرفت. یعنی محیط قطعی است.



شکل ۲-۳: گراف فضای حالت برای دنیای مکش با حسگر

این می تواند به عنوان یک مسئله به طریق زیر فرموله شود:

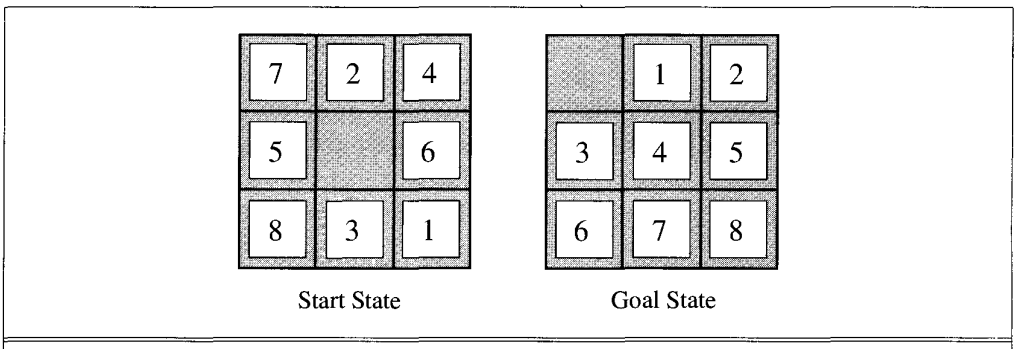
- **حالات:** عاملی در یکی از دو موقعیت است که هر کدام از آنها ممکن است کثیف باشـند. بنـابراین $۲ \times ۲^۲ = ۸$ حالت ممکن در دنیا وجود دارد.

- **حالات آغازین:** هر حالتی که به عنوان حالت اولیه طراحی شده باشد.
- **تابع مابعد:** حالات مجاز را بر مبنای قوانین مجاز (left, Right, Suck) تولید می کند. فضای حالت کامل در شکل نمایش داده شده است.
- **آزمون هدف:** کنترل می کند آیا تمامی خانه ها تمیز هستند.
- **هزینهٔ مسیر:** برای هر مرحله برابر ۱ است، بنابراین هزینهٔ مسیر تعداد مراحل مسیر است.

- **مسئله معمای هشت**

معمای هشت، که مثالی از آن در شکل زیر آمده، شامل صفحهٔ ۳×۳ با هشت مهر و یک خانهٔ خالی است. مهرهٔ مجاور فضای خالی می تواند به داخل آن بلغزد. هدف رسیدن به حالت هدف مشخصی است که برای مثال در شکل سمت راست مشخص شده است. فرموله سازی استاندارد به قرار زیر است:

- **حالات:** تشریح حالات مکان هر هشت مهره و خانهٔ خالی.
- **حالات آغازین:** هر حالتی که به عنوان حالت اولیه طراحی شده باشد.
- **تابع مابعد:** این حالت مجاز منتج از چهار عمل (حرکت خانهٔ خالی به چپ، راست، بالا یا پایین) را تولید می کند.
- **آزمون هدف:** کنترل می کند که حالت به چیدمان هدف نشان داده شده در شکل ۲-۴ تطبیق دارد (دیگر چیدمان های هدف نیز ممکن هستند).
- **هزینهٔ مسیر:** هزینهٔ هر مرحله ۱، پس هزینهٔ کل تعداد مراحل مسیر است.

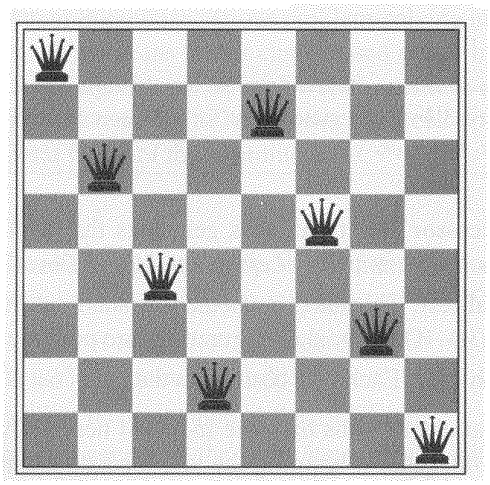


شکل ۲-۴: نمونه تیپک از معمای هشت

معمای-۸ دارای ۱۸۱.۴۴۰ = ۲ / !۹ حالت قابل دسترسی است و به سادگی حل می شود. معمای-۱۵، ۱.۳تریلیون حالت دارد و نمونه های تصادفی می توانند با بهترین الگوریتم های جستجو در چند میلی ثانیه حل شوند، معمای-۲۴ در حدود $۱۰^{۲۳}$ حالت دارد و حل بهینهٔ نمونهٔ تصادفی آن با الگوریتم ها و ماشین های کنونی بسیار دشوار است.

- **مسئله هشت وزیر**

هدف مسئلهٔ هشت وزیر چیدمان هشت وزیر روی صفحهٔ شطرنج به گونه ای است که هیچ وزیری دیگری را تهدید نکند. (یک وزیر به هر خانه در سطر، ستوی یا قطر، خود حمله می کند) شکل زیر نمایش یک پاسخ شکست خورده است، وزیر در آخرین ستون سمت راست به وزیر در بالا- چپ حمله می کند.



شکل ۲-۵: راه حل تقریبی برای مسئله هشت وزیر

اگر چه الگوریتم های خاص کارا برای حل این مسئله و کل خانوادهٔ n وزیر وجود دارد، هنوز به عنوان مسئله آزمون جالبی برای الگوریتم های جستجو، باقی مانده است. دو نوع فرموله سازی اصلی وجود دارد: اول، فرموله سازی تدریجی شامل عملگرهایی است که شرح حالت را ضمیمه می سازد و از یک حالت تهی آغاز می کنند. برای حل مسئله هشت وزیر، این یعنی هر عمل افزون بر حالت است. دوم، فرموله سازی حالت کامل از وضعیتی شروع می کند که تمامی هشت وزیر روی صفحه هستند و آنها را به اطراف حرکت خواهیم داد. در هر نوع، هزینهٔ مسیر اهمیتی ندارد چرا که تنها حالت نهایی محسوب خواهد شد. فرموله سازی تدریجی را می توان به صورت زیر توصیف کرد:

- **حالات:** هر چیدمان از صفر تا ۸ وزیر روی صفحهٔ یک حالت است.
- **حالت آغازین:** وزیری روی صفحه نیست.
- **تابع مابعد:** یک وزیر به یک خانهٔ خالی اضافه کن.
- **آزمون هدف:** ۸ وزیر روی صفحه هستند و همدیگر را تهدید نمی کنند.

در این فرموله سازی $۳ \times ۱۰^{۱٤} \approx ۵۷ \times ... \times ٦٣ \times ٦٤$ دنبالهٔ متفاوت برای تفحص خواهیم داشت. فرموله سازی بهتر می تواند از قرار دادن وزیر در هر خانه ای که در حال تهدید است ممانعت کند:

- **حالات:** چیدمان n وزیر $(0 \le n \le 8)$ هر کدام در یک ستون در سمت چپ ترین n ستون با فرض آنکه هیچ وزیری در حال تهدید دیگری در حالت مفروض نباشد.
- **تابع مابعد:** وزیری به هر خانهٔ سمت چپ ترین ستون خالی اضافه کن، چنانچه توسط هیچ وزیر دیگری تهدید نشود.

این فرموله سازی فضای حالت ۸ وزیر را از 3×10^{14} به تنها ۲۰۵۷ حالت تقلیل می دهد و یافتن پاسخ ساده است. به زبان دیگر، ۱۰۰ وزیر فرموله سازی اولیه تقریباً 10^{400} حالت دارد، حال آنکه فرموله سازی تقویت شده در حدود 10^{52} حالت خواهد داشت. این یک کاهش عظیم است، اما فضای حالت بهبود یافته هنوز برای الگوریتم های مطرح شده در این فصل بسیار بزرگ محسوب می گردد.

- **مسئله ریاضیات رمزی Crypt Arithmetic**

در مسائل کریپت اریتمتیک، حروف به جای ارقام می نشینند و هدف یافتن جایگزینی از اعداد برای حروف است که مجموع نتیجه از نظر ریاضی درست باشد. معمولاً هر حرف باید به جای یک رقم مختلف بنشیند.

مثال زیر یک نمونه شناخته شده از این مورد است:

FORTY	*Solution*:	29786	$F = 2,\ O = 9,\ R = 7, etc.$
$+\ TEN$		$+850$	
$+\ TEN$		$+850$	
$----$		$---$	
SIXTY		31486	

فرمول زیر شاید ساده ترین فرم باشد:

- **حالات:** همه حالتهای ممکن جایگزینی حروف به جای ارقام
- **حالت آغازین:** یک معمار crypt arithmetic با چند حرف جایگزین شده توسط ارقام.
- **تابع مابعد:** یک حرف را با یک رقم جایگزین کنید که قبلاً در معما ظاهر نشده باشد.
- **آزمون هدف:** معما فقط شامل ارقام است و یک مجموع صحیح را بر می گرداند.

- **هزینهٔ مسیر:** صفر

مسائل دنیای واقعی

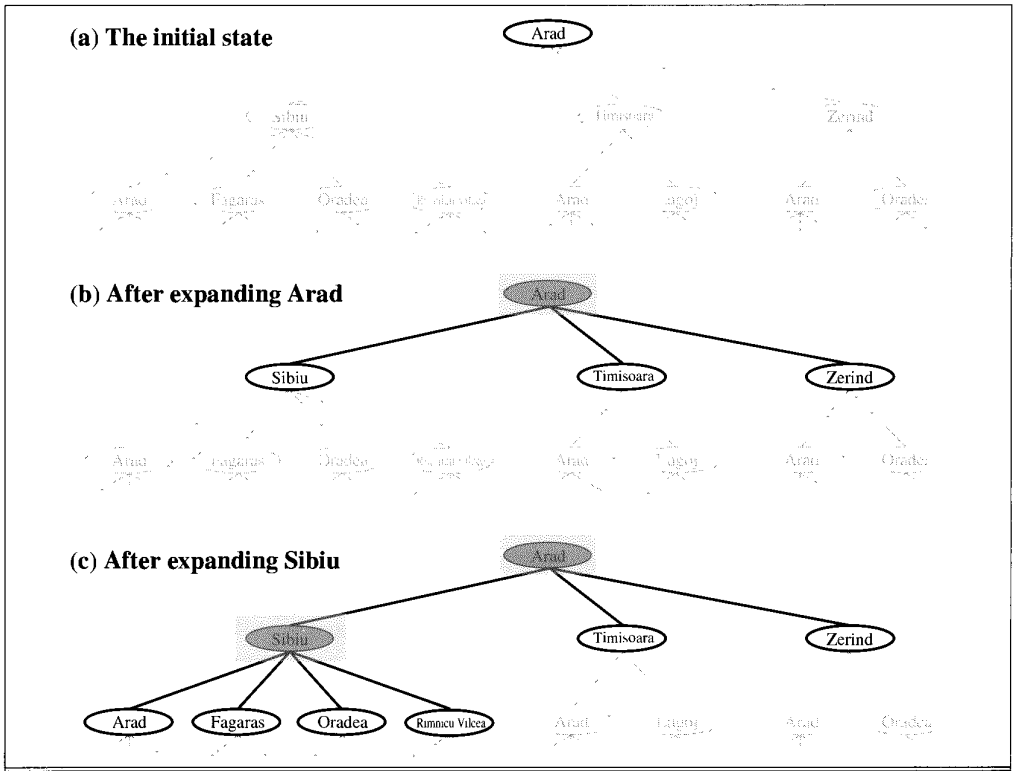
مسائلی مانند:

- Robot Navigation : می خواهیم برای روبات مسیری را پیدا کنیم که کمترین هزینه را داشته باشد.
- Route Finding : مسیریابی در شبکه های کامپیوتری: مثلاً برای ارسال یک بسته اطلاعاتی مسیری را پیدا کنیم که در کمترین زمان ممکن ارسال شود.
- Assembly Sequencing : می خواهیم چند قطعه را روی هم قرار دهیم یا مونتاژ کنیم بطوریکه اگر یک قطعه خراب شد با کمترین تعداد باز کردن قطعات بتوان آن را تعمیر کرد.
- VLSI Design : مسئله طراحی VLSI نیازمند جایابی میلیون ها مؤلفه و اتصالات آنها در یک تراشه به گونه ای است که مساحت و تأخیرات مدار و ظرفیت های موقت حداقل شود.

جستجو برای راه حل

بعد از فرموله سازی برخی از مسائل، نیاز به حل آنها داریم. که این عمل از طریق جستجو در گراف فضای حالت انجام خواهد شد. شکل زیر چند توسعه در درخت جستجو برای یافتن مسیر از آراد تا بخارست را نشان می دهد. ریشهٔ درخت جستجو یک گره جستجو است که به حالت آغازین In(Arad) وابسته است. اولین مرحلهٔ آزمون آن است که آیا هدف است؟ از آنجا که این حالت هدف نیست، نیاز به در نظر گرفتن حالات دیگر داریم. این کار از طریق توسعهٔ حالت جاری انجام می شود، یعنی تابع مابعد را به حالت جاری اعمال نموده و مجموعهٔ جدیدی از حالات تولید خواهد شد. در این مورد، سه حالت جدید خواهیم داشت:

In(Sibiu)، In(Timisoara) و In(Zerind). حال می بایست یکی از این سه را برای ادامه انتخاب کنیم.



شکل ۲-۶: درخت های جستجوی نیمه کاره برای یافتن مسیری از آراد به بخارست. گره های توسعه یافته خاکستری شده اند، گره هایی که تولید شده اند اما توسعه نیافته اند حاشیه توپر دارند و گره هایی که هنوز تولید نشده اند با خط های تیره مشخص شده اند.

فرض کنید ابتدا سیبو را در نظر می گیریم. اول کنترل می کنیم آیا هدف است (یا نه) و سپس آن را برای رسیدن به In(Fagaras)، In(Arad)، In(Oradea) و In(Rimnicu Vilcea) توسعه می دهیم. سپس می توانیم هر یک از آن چهار تا را انتخاب کرده، یا به عقب برگشته و دو شهر دیگر را انتخاب کنیم. انتخاب، آزمون و توسعه را تا جایی تکرار می کنیم که یا پاسخی یافت شود و یا اینکه حالتی برای توسعه باقی نمانده باشد. انتخاب حالتی که برای توسعه تعیین می شود استراتژی جستجو نام دارد.

باید به تفاوت بین فضای حالت و درخت جستجو توجه کرد. برای مسائل یافتن مسیر، تنها ۲۰ حالت در فضای حالت، یکی برای هر شهر، وجود دارد. اما تعداد نامتناهی مسیر در این فضای حالت دیده می شود. بنابراین درخت جستجو می تواند تعداد نامتناهی گره داشته باشد.

برای مثال، سه مسیر آراد- سیبیو، آراد- سیبیو- آراد، آراد– سیبیو- آراد- سیبیو اولین سه دنباله مسیر نامتناهی هستند. (آشکارا، الگوریتم جستجوی متناسب از چنین حالات تکراری جلوگیری می کند).

راه های متفاوتی برای بازنمایی گره ها وجود دارد. اما فرض می کنیم که یک گره دارای ساختار داده ای با پنج مولفه است:

- حالت: حالت در فضای حالت که گره به آن وابسته است.
- گره والد: گره ای در درخت جستجو که این گره را تولید کرده است.
- عمل: عملی که به والد اعمال شده تا گره را تولید کند.
- هزینه مسیر: هزینه معمولاً با $g(n)$ بیان می شود که شامل مسیر از گره آغازین تا گره جاری است.
- عمق: تعداد مراحل در طول مسیر از حالت آغازین

همچنین نیاز داریم تا مجموعه گره های تولید شده ولی توسعه نیافته را بازنمایی کنیم که این مجموعه گره ها حاشیه (گره زنده fringe) نامیده می شوند. هر عنصر حاشیه یک گره برگی در درخت است. در شکل ۲-۶ حاشیه هر درخت شامل گره هایی با حاشیه توپر است. ساده ترین راه نمایش گره های حاشیه ای استفاده از یک مجموعه است. استراتژی جستجو می تواند تابعی باشد که گره بعدی را از این مجموعه برای توسعه انتخاب کند. در این کار اگر چه از نظر نظری، همه چیز روشن است اما، از نظر محاسباتی هزینه بر خواهد بود چرا که تابع استراتژی ممکن است به هر عنصر مجموعه دقت کند تا بهترین آنها را انتخاب کند. بنابراین، ما مجموعه گره ها را به صورت یک صف پیاده سازی می کنیم. عملیات روی صف به قرار زیرند:

- Make-Queue(element,…) صفی با عناصر داده شده می سازد.

- Empty(queue) درست بر می گرداند اگر عنصر دیگری در صف نباشد.
- First(queue) اولین عنصر صف را بر می گرداند.
- Remove-First(queue) ابتدا First(queue) را بر می گرداند و سپس آن را از صف حذف می کند.
- Insert(element, queue) عنصر در صف درج می کند و صف حاصل را بر می گرداند
- Insert-All(elements, queue) مجموعه عناصری را داخل صف درج کرده و صف را بر می گرداند.

الگوریتم عمومی جستجوی درخت در شکل ۲-۷ آمده است.

```
function TREE-SEARCH (problem, fringe) returns a solution, or failure
    fringe ← INSERT (MAKE-NODE (INITIAL-STATE [problem]), fringe)
    loop do
        if Empty (fringe) the 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]+1
      add s to successors
  return successors
```

شکل ۲-۷: الگوریتم جستجوی درخت عمومی. (توجه کنید که آرگومان *fringe* می بایست یک صفت تهی باشد و نوع صف بر توالی جستجو اثر می گذارد) تابع *SOLUTION* دنباله ای از اعمال کسب شده توسط بازگشت از طریق اشاره‌گر به والدین تا ریشه را بر می گرداند.

اندازه گیری کارایی حل مسئله

خروجی الگوریتم حل کردن مسئله یا شکست و یا راه حل است. (برخی الگوریتم ها در یک حلقه بینهایت گیر می کنند و هیچ گاه خروجی تولید نخواهند کرد). می توانیم کارایی الگوریتم را از چهار طریق ارزیابی کنیم:

- **کاملیت:** آیا الگوریتم ضمانتی در یافتن راه حل در صورت وجود آن دارد؟
- **بهینگی**: آیا استراتژی قادر به یافتن پاسخ بهینه است؟
- **پیچیدگی زمانی**: چقدر طول می کشد تا راه حل را پیدا کند؟
- **پیچیدگی فضای مصرفی**: چقدر حافظه برای انجام جستجو لازم است؟

پیچیدگی زمان و فضای مصرفی همواره به عنوان اندازه های دشواری مسئله لحاظ می شود. در AI که گراف به طور ضمنی توسط حالت آغازین و تابع مابعد نمایش داده می شود و اغلب نامتناهی است، پیچیدگی به صورت سه کمیت بیان می شود:

- **b** فاکتور انشعاب یا حداکثر تعداد مابعدهای یک گره،
- **d** عمق کم سطح ترین گره هدف،
- **m** : طول حداکثر هر مسیری در فضای حالت.

زمان اغلب توسط تعداد گره های تولید شده در طی جستجو اندازه گیری می شود و فضا توسط حداکثر تعداد گره های ذخیره شده در حافظه.

برای حصول موثر بودن الگوریتم جستجو، تنها هزینه جستجو در نظر گرفته شده که به طور نمونه وابسته به پیچیدگی زمانی است اما می تواند اغلب شامل مفهومی برای مصرف حافظه نیز باشد . یا اینکه از هزینه

کل استفاده کنیم که ترکیبی از هزینه جستجو و هزینه مسیر راه حل یافت شده است. برای مسئله یافتن مسیری از آراد به بخارست، هزینه جستجو اندازه زمان در نظر گرفته شده. جستجو و هزینه راه حل، هزینه کل مسیر خواهد بود. بنابراین برای محاسبه هزینه کل، می بایست کیلومترها و میلی ثانیه ها را اضافه کنیم. هیچ گونه نرخ تبدیل رسمی بین این دو وجود ندارد اما منطقی به نظر می رسد که در این حالت کیلومترها را به کمک تخمین سرعت متوسط ماشین به میلی ثانیه ها تبدیل کنیم. (زیرا زمان چیزی است که عامل به آن توجه دارد) این عامل را قادر می سازد تا نقطه توازن بهینه ای را بیابد که در آن محاسبات بیشتر برای یافتن مسیر کوتاه تر ساده تر گردد.

تقسیم بندی روشهای جستجو

- **جستجوی ناآگاهانه (Uninformed Search) Blind Search**

این روش جستجو که غالباً جستجوی کورکورانه نیز نامیده می شود بدان معنی است که به جز آنچه که در تعریف مسئله آمده فاقد هر نوع اطلاعات اضافی درباره وضعیت هاست. این روشها همه کاری که انجام می دهند تولید گره های مابعد و تمایز وضیعت هدف از وضعیت غیر هدف است. آنها هیچ ایده ای راجع به فاصله گره های فعلی تا هدف ندارند.

- **جستجوی آگاهانه (Informed Search) Heuristic Search**

در این نوع جستجو که غالباً جستجوی کشف کننده نیز نامیده می شود، نیاز به اطلاعاتی داریم که بتوانیم به شکلی فاصله تا هدف را تخمین بزنیم. این نوع جستجوها می دانند که آیا یک وضعیت غیر هدف امید بخش تر از دیگران هست یا نه.

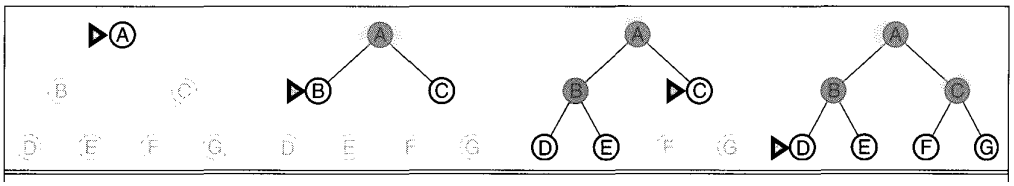
همه روشهای جستجو براساس توالی توسعه (بسط) گره‌ها، از هم متمایز می گردند. در ادامه به معرفی چند روش جستجوی ناآگاهانه می پردازیم.

جستجوی اول- سطح Breadth-First Search

جستجوی سطحی

یکی از استراتژیهای ساده جستجوی سطحی Breadth-First Search است. در این استراتژی ابتدا، گره ریشه، گسترش می یابد، سپس تمام گره هایی که توسط ریشه تولید شده اند، خودشان گسترش می یابند و سپس مولدهای آنها، و به همین ترتیب. در حالت کلی در درخت جستجو تمام گره های عمق d، قبل از گره های عمق $d+1$ گسترش داده می شوند. جستجوی سطحی توسط یک صف پیاده سازی می شود.

جستجوی سطحی یک استراتژی بسیار سیستماتیک است زیرا ابتدا تمام مسیرهای با طول ۱ را در نظر می گیرد و سپس مسیرهای با طول ۲ و الی آخر. شکل (۲-۸) پیشرفت جستجو را روی یک درخت دو دویی ساده نشان می دهد. اگر راه حلی وجود داشته باشد، جستجوی سطحی ابتدا کم عمق ترین وضعیت هدف را پیدا می کند. مطابق با چهار معیار گفته شده در قبل، جستجوی سطحی کامل و هم چنین بهینه است چرا که هزینه مسیر، یک تابع کاهش نیابنده از عمق گره است. (این شرط معمولاً فقط زمانی برقرار می شود که تمام عملگرها هزینه مشابهی داشته باشند. این الگوریتم زمانی تشخیص می دهد که یک نود هدف است که بخواهد آن را بسط دهد.



شکل ۲-۸ جستجوی اول- سطح بر روی یک درخت سادهٔ دو دویی، در هر مرحله گره توسعه یابنده در مرحلهٔ بعد توسط علامتی مشخص شده است.

تا کنون، اخبار در مورد جستجوی سطحی، خوب بوده است. برای این که 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 حالت می تواند گسترش داده شود تا به b حالت جدید برسد. می گوییم که فاکتور انشعاب (branching factor) از این حالات (و از درخت جستجو) b است.

ریشه درخت جستجو، b گره در اولین سطح تولید می کند. هر کدام گره های b بیشتری تولید می کنند و b^2 گره در سطح دوم خواهیم داشت. هر کدام از این ها، گره های b بیشتری تولید می کنند تا در سطح سوم به گره های b^3 برسد و این کار تا انتها ادامه پیدا می کند. حال تصور کنید که راه حل برای این مسئله طول مسیری از d دارد. سپس حداکثر تعداد گره های بسط داده شده قبل از پیدا شدن راه حل $1+b+b^2+b^3+...+b^d+\left(b^{d+1}-b\right)$ است.

فرمول بالا حداکثر تعداد را نشان می دهد، اما راه حل در هر نقطه از سطح d ام می تواند پیدا شود. در بهترین حالت تعداد، عدد کوچکتری را نشان می دهد. آنهایی که آنالیز پیچیدگی انجام می دهند، زمانی که با یک مرتبه زمان نمایی $O\left(b^{d+1}\right)$ برخورد می کنند عصبانی می شوند.

شکل ۲-۹ علت را نمایش می دهد. این شکل لیستی از زمان و فضای مورد نیاز برای جستجوی اول- سطح با فاکتور انشعاب $b=10$ را ارائه می دهد که برای مقادیر متفاوت عمق پاسخ d معین شده اند. جدول فرض می کند ۱۰/۰۰۰ گره در هر ثانیه تولید می شوند و هر گره ۱۰۰۰ بایت فضا نیاز دارد.

حافظه درخواست شده مسئله جدی تری برای جستجوی سطحی نسبت به زمان اجرای آن است. برای مثال بیشتر مردم حوصله انتظار ۱۹ دقیقه ای برای کامل شدن جستجو در عمق ۶ را دارند، اما خیلی از آنها حافظه ای در حدود ۱۰ گیگا بایت 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 انگشت شماری به ۱ ترا بایت حافظه مورد نیاز، دسترسی دارند.

حداقل دو درس را باید از شکل ۲-۹ فراگرفت. اول، نیاز حافظه مسئلهٔ بزرگتری برای جستجوی اول- سطح از زمان اجرا است. ۳۱ ساعت زمان بسیار طولانی انتظار جهت حصول پاسخ یک مسئلهٔ مهم تا عمق ۸ نیست، اما کمتر کامپیوتری حافظهٔ اصلی ترابایتی دارد. خوشبختانه روش های دیگری جهت استراتژی جستجو وجود دارند که به فضای کمتری نیاز دارند. درس دوم آن است که زمان مورد نیاز هنوز عامل

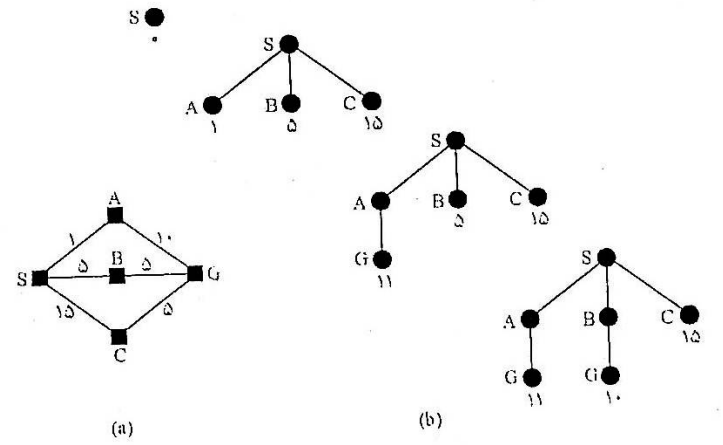
مهمی است. اگر مسئلهٔ شما عمق پاسخ ۱۲ داشته باشد، جستجوی اول- سطح به ۳۵ سال وقت (یا در واقع هر روش غیر آگاهانه ای) برای یافتن پاسخ نیاز دارد. به طور کلی، مسائل جستجو با پیچیدگی نمایی، نمی توانند توسط روش های غیرآگاهانه به جز کوچکترین نمونه های آن حل شوند.

Depth	Nodes	Time	Memory
2	1100	11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35years	10 petabytes
14	10^{15}	5,523 years	1exabyte

شکل ۲-۹ نیازهای زمانی و فضایی جستجوی اول- سطح اعداد نمایش داده شده با فرض فاکتور انشعاب $b=10$ ۱۰/۰۰۰ گره/ ثانیه و ۱۰۰۰ بایت/ گره هستند.

جستجوی هزینه یکسان (یکنواخت) (Uniform Cost Search)

جستجوی سطحی کم عمق ترین حالت هدف را پیدا می کند، اما همیشه کم هزینه ترین حالت برای یک تابع هزینه مسیر عمومی نیست. جستجوی با هزینه یکسان استراتژی جستجوی سطحی را توسط بسط کم هزینه ترین گره (که توسط تابع هزینه مسیر g(n) اندازه گیری شده است)، نه کم عمق ترین گره، بهبود می بخشد. آسان است ببینیم که جستجوی سطحی همان جستجوی با هزینه یکسان با g(n)= DEPTH(n) است.



شکل ۲-۱۰ مثالی از جستجوی با هزینه یکسان

زمانی که شرایط عمومی برقرار باشد، اولین راه حل پاسخ، ضمانت می کند که ارزانترین راه نیز باشد، زیرا اگر مسیر ارزانتری وجود داشته باشد که راه حل 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 می دهد، که منجر به پیدایش مسیرهای عملگر C,B,A می شود. چون هزینه های مسیر به A کمتر از بقیه است، در مرحله بعد، A قبل از بقیه بسط داده می شود و مسیرهای SAG بوجود میاید که در حقیقت همان راه حل است.

قبلاً بیان کردیم که الگوریتم زمانی تشخیص می دهد یک نود هدف است که بخواهد آن را بسط دهد. بنابراین با اینکه به نود G رسیده ایم اما هنوز به عنوان هدف تشخیص داده نمی شود. در مرحله بعد از بین هزینه نودهای بسط داده نشده $G(11)$، $B(5)$ و $C(15)$ نود B انتخاب می شود و بسط داده می شود. مسیر SBG بوجود می آید. اکنون نودهای بسط داده نشده عبارتند از $G(11)$، $a(10)$ و $C(15)$ که $a(10)$ انتخاب می شود و در اینجا هدف تشخیص داده می شود. بنابراین مسیر هدف SBG با هزینه ۱۰ خواهد بود.

جستجوی عمقی Depth-First-Search

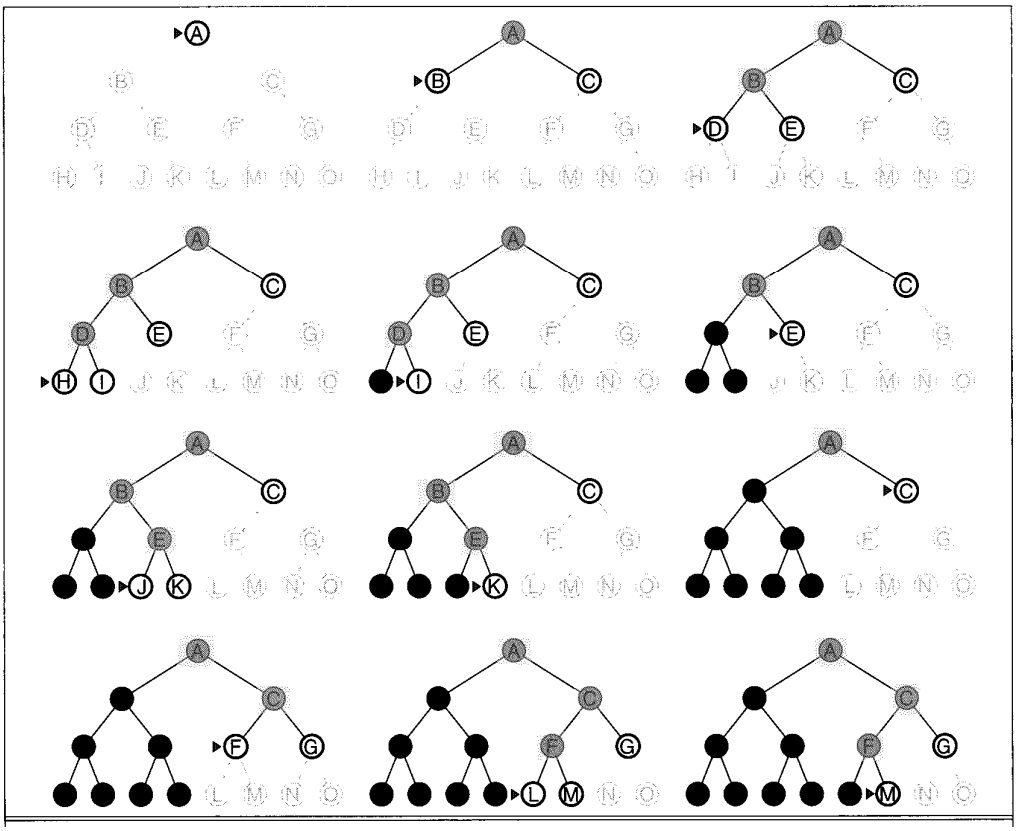
جستجوی اول- عمق همواره عمیق ترین گره را در حاشیهٔ درخت جستجوی کنونی توسعه می دهد. پیشرفت جستجو در شکل ۲-۱۱ نشان داده شده است. جستجو بلافاصله تا عمیق ترین سطح درخت جستجو گسترش می یابد، جایی که گره ها ما بعدی ندارند. از آنجا که این گره ها توسعه یافته اند، از حاشیه حذف شده و بنابراین جستجو به عقب برگشته و به دنبال عمیق ترین گره بعدی که هنوز توسعه نیافته خواهد گشت.

این استراتژی می تواند توسط tree-search توسط پشته (صف آخر- ورود- اول- خروج (LIFO)) پیاده سازی شود. الگوریتم آن به صورت زیر است:

```
function Depth-First-Search (problem) return solution
{
    Return TREE-SEARCH( problem , Enque-at-front)
}
```

جستجوی عمقی همیشه یکی از گره ها را در پایین ترین سطح درخت، بسط می دهد. فقط زمانی که جستجو به یک بن بست می رسد (یک گره غیر هدف بدون امکان بسط)، برگشت داده می شود و به سراغ گره هایی در سطوح کم عمق تر می رود.

جستجوی اول- عمق به حافظه بسیار کمی نیاز دارد. این روش تنها نیاز به ذخیرهٔ مسیر واحد از ریشه به گره برگی را دارد، به همراه گره های خواهر- برادر توسعه نیافته باقی مانده برای هر گره در مسیر. زمانی که گرهی توسعه می یابد، بلافاصله پس از اینکه تمام نوادگان کاملاً پویش شدند، از حافظه حذف می شود (شکل ۲-۱۱). برای فضای حالت با فاکتور انشعاب b و عمق حداکثر m، جستجوی اول- عمق به فضای تنها $bm+1$ گره نیاز دارد. با فرضیات شکل ۲-۹ و با فرض اینکه گره ها در عمق یکسان با گره هدف ما بعدی ندارند، جستجوی اول- عمق نیاز به ۱۱۸ کیلو بایت فضا بجای ۱۰ پتابایت برای عمق $d=12$ دارد، یعنی ۱۰ میلیون بار فضای کمتر.



شکل ۲-۱۱ جستجوی اول عمق در درخت دو دویی، گره هایی که توسعه یافته اند و نوادگانی در حاشیه ندارند و از حافظه حذف می شوند، با رنگ مشکلی نشان داده شده اند. گره های در عمق ۳ فرض شده فاقد فرزندی هستند و m تنها گره هدف است.

پیچیدگی زمانی برای جستجوی عمقی $O(b^m)$ است. برای مسائلی که راه حلهای زیادی دارنـد، جسـتجوی عمقی سریعتر از جستجوی سطحی عمل می کند، زیرا شانس خوبی برای یافتن راه حل بعد از بررسی فقط یک قسمت کوچک از کل فضا را دارد. جستجوی عمقی در بدترین حالـت دارای پیچیـدگی زمـانی $O(b^m)$ است.

یکی از مضرات جستجوی عمقی آنست که در یک مسیر اشتباه هنگام پایین رفتن، گیـر مـی کنـد. مسـائل زیادی دارای درختهای عمیق و نامحدودی هستند، بنابراین جستجوی عمقی هرگز قادر نخواهد بـود کـه از یک انتخاب ناموفق در یکی از گره های نزدیک بالای درخت، جان سالم بدر برد. جستجو همیشه به سـمت

پایین ادامه خواهد یافت بدون اینکه به طرف بالا برگردد، حتی زمانی کـه راه حـل کوتـاه تـری نیـز وجـود داشته باشد. از این رو در این مسائل نیز جستجوی عمقی در یک حلقه بی پایانی خواهـد افتـاد و هرگـز راه حلی را پیدا نخواهد کرد، یا بالاخره ممکن است راه حلی پیدا کند که طولانی تر از راه حل بهینه باشد. این بدان معناست که جستجوی عمقی نه کامل و نه بهینه اسـت. بـه همـین علـت، جسـتجوی عمقـی بایـد از درختهای جستجوی با عمق نامحدود یا بزرگ اجتناب ورزد. معمولاً، جسـتجوی عمقـی بصـورت فراخـوانی بازگشتی و به کمک پشته سیستم پیاده سازی می شود.

جستجوی عمق محدود شده Depth Limited Search

مسئلهٔ درختان با عمق فاقد کران می تواند توسط اعمال محدودیت عمق ℓ از پیش تعریف شده بـر طـرف گردد. یعنی، گره های عمق ℓ به عنوان گره های فرزنـد محسـوب شـوند. ایـن رهیافـت جسـتجوی عمـق محدود شده نام دارد. محدودیت عمق مسئلهٔ بی نهایت را حل می کند. متأسفانه، هنوز منبع دیگـری بـرای عدم کاملیت وجود دارد که همانا $\ell < d$ است، یعنی، کم عمق ترین هدف ماورای محـدودیت عمـق باشـد. (این زمانی محتمل است که d ناشناخته باشد). جستجوی عمق محدود شده همچنـین بهینـه نیسـت اگـر $\ell < d$ باشد. پیچیدگی زمانی آن $O(b^{\ell})$ و پیچیدگی فضای آن $O(b\ell)$ است. جستجوی اول- عمق حالـت خاصی از جستجوی عمق محدود شده حساب می شود که در آن $\ell = \infty$ است.

گاهی، محدودیت عمق بر پایهٔ دانش مسئله است. برای مثال، در نقشهٔ رومانی ۲۰ شهر وجود دارد. بنـابراین می دانیم اگر پاسخی وجود داشته باشد، حداکثر طول آن ۱۹ است. پس $\ell = 19$ یک انتخاب ممکن اسـت. اما اگر نقشه را دقیقاً مطالعه کنیم، کشف می کنیم که هر شهر حداکثر در ۹ مرحله قابل دسترسی از شـهر دیگری است. این عدد، که قطر فضای حالت نام دارد، به ما محدودیت عمق بهتری می دهد کـه منجـر بـه جستجوی عمق محدود شدهٔ کاراتری خواهد شد. اما برای بیشتر مسائل، از عمق محدودیت مناسب اطلاعی نداریم تا مسئله حل شود.

جستجو با عمق محدود شده می تواند با یک تغییر ساده در الگوریتم جستجوی درختی عمومی و یا با الگوریتم بازگشتی جستجوی اول- عمق پیاده سازی شود. در شکل ۲-۱۲ شبه کد الگوریتم بازگشتی جستجوی عمق محدود شده نشان داده شده است. توجه کنید که جستجوی عمق محدود شده می تواند با دو نوع خطا اتمام یابد: مقدار خطای استاندارد معرف عدم وجود پاسخ است، مقدار برش معرف عدم وجود پاسخ در محدودهٔ عمق معین شده است.

```
function DEPTH-LIMTTED-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 ۲-۱۲ پیاده سازی بازگشتی جستجوی عمق محدود شده

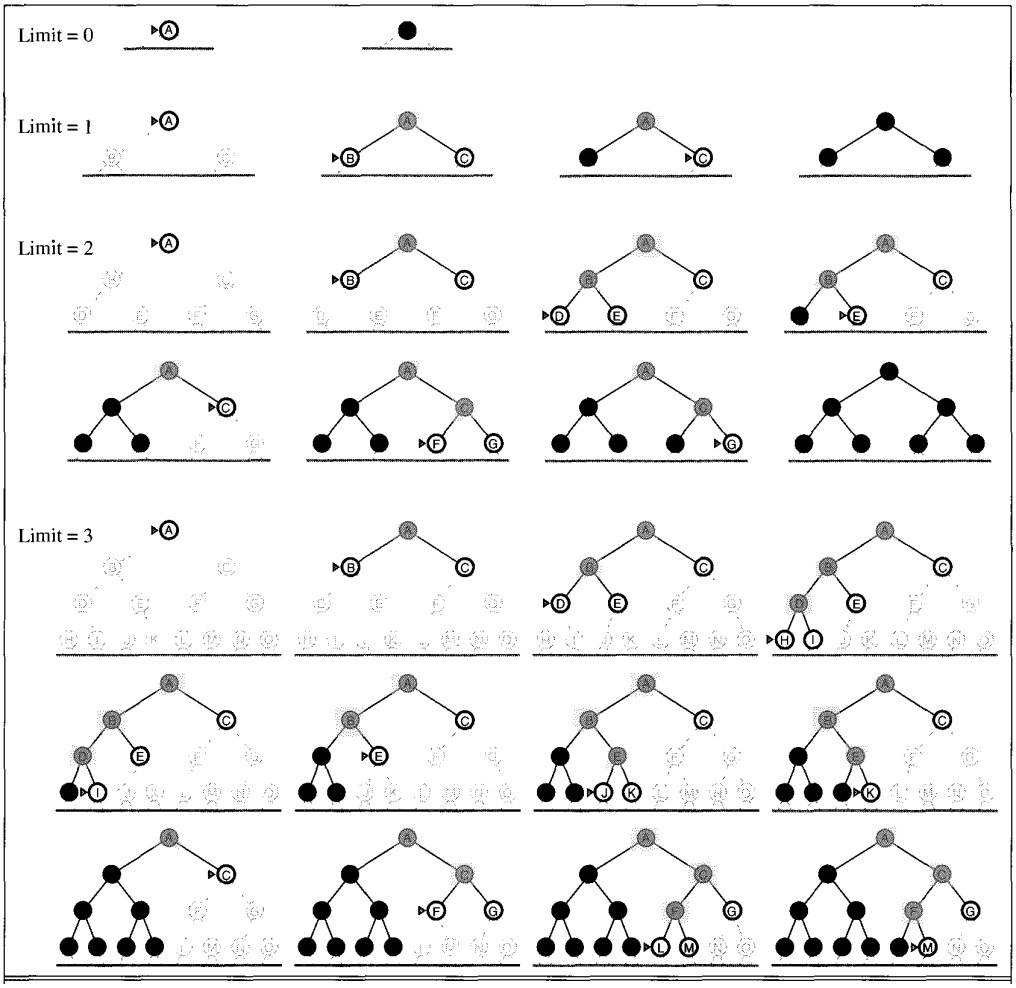
جستجوی عمیق شوندهٔ تکراری Iterative Deepening Search

جستجوی عمیق شوندهٔ تکراری استراتژی عمومی است که با ترکیب جستجوی اول- عمق، بهترین محدودیت عمق را می یابد. این کار از طریق افزایش تدریجی محدودیت صورت می گیرد- اول صفر، سپس ۱، سپس ۲، غیره- تا هدف کشف شود. این زمانی روی می دهد که محدودیت عمق به d برسد، عمق کم عمق ترین گره هدف. الگوریتم در شکل ۲-۱۳ نشان داده شده است. جستجوی عمیق شوندهٔ تکراری مزایای اول- عمق و اول- سطح را ترکیب می کند. همانند جستجوی اول- عمق، حافظهٔ مورد نیاز وابسته

به $O(bd)$ خواهد بود. همانند جستجوی اول- سطح، اگر فاکتور انشعاب متناهی باشد، کامل و بهینه است. شکل ۲-۱۴ چهار تکرار از الگوریتم را روی درخت دو دویی نشان داده که پاسخ در تکرار چهارم یافت شده است.

function ITERATIVE-DEEPENING-SEARCH (*problem*) **returns** a solution, or failure
input: *problem*, a problem
for *dept* $\leftarrow 0$ **to** ∞
    result $\leftarrow$ DEPTH-LIMITED-SEARCH (*problem*. *depth*)
    if *result* $\neq$ Cutoff **then return** *result*

شکل ۲-۱۳: الگوریتم جستجوی عمیق شوندهٔ تکراری، مکرراً جستجوی عمق محدوده شده را با افزایش محدودیت تکرار می کند. زمانی متوقف می شود که پاسخی یافت شود، و یا اگر جستجوی عمق محدود شده پاسخ شکست تولید کند که به معنی عدم وجود پاسخ است.



شکل ۲-۱۴ چهار تکرار جستجوی عمیق شونده تکراری در درخت دو دویی

جستجوی عمیق شوندهٔ تکراری به نظر اتلاف کننده می آید، زیرا وضعیت ها چندین بار تولید می شوند. می توان نشان داد که این چندان هزینهٔ مهمی ندارد. علت آن است که در درخت جستجو با فاکتور انشعاب یکسان (یا نزدیک یکسان) در هر سطح، بیشتر گره ها در سطح پایینی قرار دارند. بنابراین اهمیتی ندارد که سطوح بالایی چندین بار تولید شوند. در جستجوی عمیق شوندهٔ تکراری، گره های سطح انتهایی (عمق d) یکبار تولید می شوند، یک سطح بالاتر دوبار و به همین ترتیب تا بچه های ریشه که d بار تولید می شوند. پس تعداد کل گره های تولید شده:

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

که به ما پیچیدگی زمانی $O(b^d)$ می دهد. این را می توانیم با گره های تولیدی توسط جستجوی اول سطح مقایسه کنیم:

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

توجه کنید که جستجوی اول- سطح، برخی گره ها در سطح $d+1$ را توسعه می دهد، حال آنکه عمیق شوندۀ تکراری این کار را نمی کند. نتیجه اینکه جستجوی عمیق شوندۀ تکراری عملاً سریع تر از جستجوی اول- سطح بر خلاف تولید مکرر حالات است. برای مثال، اگر $b = 10$ و $d = 5$ باشد، تعداد برابر است با:

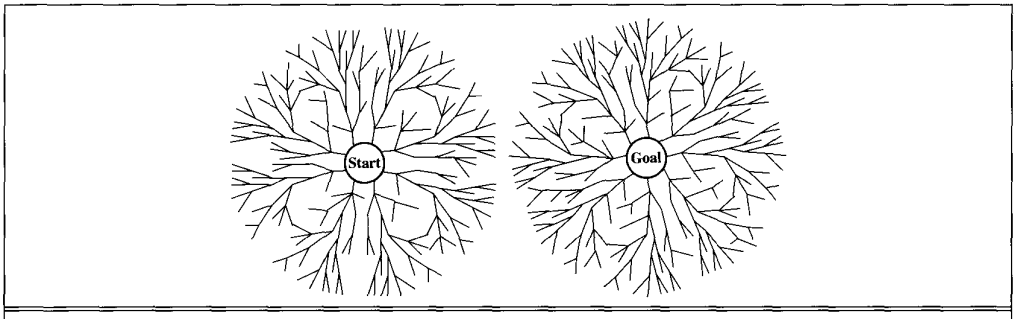
$$N(IDS) = 50 + 400 + 3/000 + 20/000 + 100/000 = 123/450$$

$$N(BFS) = 10 + 100 + 1/000 + 10/000 + 999/990 = 1/111/100$$

به طور کلی، عمیق شوندۀ تکراری روش جستجوی غیرآگاهانه برتر زمانیست که فضای حالت بزرگ بوده و عمق پاسخ نامعین است.

جستجوی دو طرفه (Bidirectional Search)

ایده جستجوی دو طرفه در واقع شبیه سازی جستجو 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 می شود (شکل ۲-۱۵). برای مسائلی که فاکتور انشعاب در دو جهت b است، جستجوی دو طرفه تفاوت بزرگی را ایجاد می کند. اگر ما فرض بگیریم که راه حلی در عمق d وجود دارد، این راه حل در مراحل $O(2b^{d/2}) = O(b^{d/2})$ پیدا خواهد شد. زیرا جستجو به سمت عقب و جلو، می بایست فقط نیمی از راه را طی کند. برای اثبات این امر: برای $b = 10$ و $d = 6$، جستجوی سطحی ۱۱/۱۱۱/۱۰۰ گره تولید می کند، در حالی که جستجوی دو طرفه اگر هر دو طرف از جستجوی اول سطح استفاده کنند، که هر جهت در عمق ۳ است، موفق می شود و در این حالت ۲۲/۲۰۰ گره تولید می کند. موارد زیادی قبل از اینکه الگوریتم بتواند پیاده سازی شود، نیاز به پاسخگویی دارند.



شکل ۲-۱۵ جستجوی دو طرفه

- سؤال اصلی این است که، جستجو از سمت هدف به چه معنی است؟ ماقبل های (predecessors) یک گره n را گره هایی در نظر می گیریم که n مابعد (successor) آنها باشد. جستجو به سمت عقب بدین معناست که تولید ماقبل ها از گرهٔ هدف آغاز شود.
- زمانی که تمام عملگرها، قابل وارونه شدن باشند، مجموعه ماقبل ها و مابعدها یکسان هستند. برای بعضی از مسائل، بهرحال، محاسبه والدها ممکن است بسیار مشکل باشد.
- چه کار می توان کرد زمانی که هدفهای متفاوتی وجود داشته باشد؟ اگر لیست صریحی از حالتهای هدف وجود داشته باشد، می توانیم یک تابع ماقبل برای مجموعه حالت تقاضا کنیم در حالی که تابع مابعد یا (جانشین) در جستجوی مسائل چند وضعیت به کار می رود. اگر ما فقط تعریفی از مجموعه داشته باشیم، ممکن خواهد بود که شرحی از «مجموعه حالاتی که مجموعه هدف را تولید می کنند» حاصل شود، اما این یک عمل زیرکانه خواهد بود. برای مثال، چه حالتی، والدهای هدف کیش و مات در شطرنج هستند؟
- باید یک آزمون 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 تصمیم بگیریم که چه نوع جستجویی در هر نیمه قصد انجام دارد. برای مثال شکل ۲-۱۵ دو جستجوی سطحی را نشان می دهد. آیا این جستجو بهترین انتخاب است؟

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

مقایسهٔ استراتژی های غیرآگاهانه

شکل ۲-۱۶ استراتژی های جستجو را بر مبنای چهار ملاک اشاره شده در بخش قبل را با هم مقایسه می کند.

Criterion	Breadth Frist	Uniform Cost	Depth Firdt	Depth Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^{a}	$Yes^{a,b}$	No	No	Yes^{a}	$Yes^{a,d}$
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^{m})$	$O(b^{\ell})$	$O(b^{d})$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^{c}	Yes	No	No	Yes^{c}	$Yes^{c,d}$

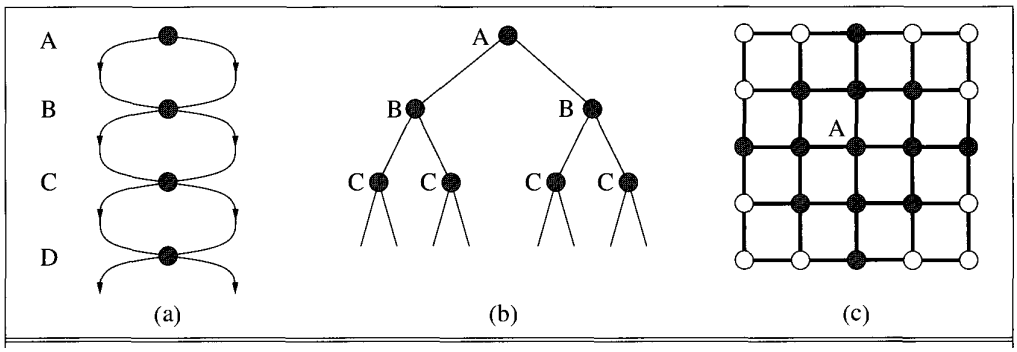
شکل ۲-۱۶: ارزیابی استراتژی های جستجو. b فاکتور انشعاب، d عمق کم عمق ترین پاسخ، m عمق حداکثر درخت جستجو و ℓ محدودیت عمق است. بالانویس ها به قرار زیرند: ^{a} کامل است اگر b متناهی باشد، ^{b} کامل است اگر هزینهٔ مرحلهٔ بزرگتر یا مساوی ε باشد. ^{c} بهینه است اگر هزینه های مراحل تماماً یکسان باشند. ^{d} اگر هر دو جهت از جستجوی اول- سطح استفاده کرده باشند.

اجتناب از حالات تکراری

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

برای مسائل زیادی، حالات تکراری غیر قابل اجتناب هستند این شامل تمام مسائلی می شود که عملگرها قابل وارونه شدن باشند، مانند مسائل مسیریابی. درخت های جستجو برای این مسائل نامحدود هستند، اما اگر تعدادی از حالات را حذف کنیم، می توانیم 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 می دهد، باقی بماند.

حتی زمانی که درخت محدود است، اجتناب از وقوع حالات تکراری می تواند موجب کاهش نمایی در هزینه جستجو شود. مثال کلاسیک آن در شکل ۲-۱۷ نشان داده شده است. فضا فقط شامل $m+1$ حالت می شود، جایی که m حداکثر عمق است. به علت اینکه درخت هر مسیر ممکنی را در فضا شامل می شود، 2^m شاخه دارد.



شکل ۲-۱۷: فضای حالت هایی که درختان جستجوی بزرگتر نمایی دارند.

(a) یک فضای حالت که در آن دو عمل منجر به رسیدن از A به B، دو عمل منجر به رسیدن از B به C ، به همین ترتیب وجود دارد. فضای حالت شامل d+1 وضعیت است که در آن d عمق حداکثر است.

(b) درخت جستجوی وابسته که در آن 2d مسیر در فضا وجود دارد.

(c) یک فضای شبکه ای مربعی، حالات کلی دو مرحله از وضعیت آغازین (A) خاکستری شده اند.

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

- به حالتی که هم اکنون از آن آمده اید، برنگردید. داشتن تابع بسطی (یا مجموعه عملگرها) که از تولید مابعدهایی که مشابه حالتی هستند که در آنجا نیز والدین این گره ها وجود دارند، جلوگیری کند.
- از ایجاد مسیرهای دوار بپرهیزید. داشتن تابع بسطی (یا مجموعه عملگرها) که از تولید مابعدهای یک گره که مشابه اجداد آن گره است، جلوگیری کند.
- حالتی را که قبلاً تولید شده است، مجدداً تولید نکنید. این مسئله باعث می شود که هر حالت در حافظه نگه داری شود، و 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، الگوریتم های جستجو اغلب از جدول پراکندگی که تمامی گره های تولیدی را ذخیره می کند، استفاده می کنند. این امر کنترل وضعیت های تکراری را بخوبی امکان پذیر می سازد. بده- بستان بین هزینه مرتب سازی و کنترل و هزینه اضافه جستجو بسته به مسئله دارد، فضاهای حالت حلقه ای تر بیشتر به کنترل تمایل دارند.

برای جستجوی اول- عمق تنها گره های موجود در حافظه، آنهایی هستند که روی مسیر از ریشه تا گره جاری قرار دارند. مقایسهٔ آن گره ها با گره جاری می تواند به الگوریتم امکان کشف مسیرهای حلقه ای را داده و به سرعت حذفشان کند. این برای گراف های حالت متناهی مناسب است تا تبدیل به درختان جستجوی نامتناهی به خاطر حلقه ها نشوند، متأسفانه این نمی تواند مانع از مجزا سازی نمایی مسیرهای فاقد حلقه در مسائلی از قبیل شکل ۲-۱۷ شود. تنها راه برای اجتناب آن است که گره های بیشتری را در حافظه نگهداری کنیم. این یک مصالحهٔ اساسی بین زمان و فضا است. الگوریتم هایی که تاریخچهٔ خود را فراموش می کنند محکوم به تکرار هستند.

اگر الگوریتمی هر حالتی را که ملاقات کرده به یاد بیاورد، می تواند به صورت پویشگری که مستقیماً گراف فضای حالت را کاوش می کند نشان دهد. می توانیم الگوریتم tree-search را به گونه ای تغییر دهیم که شامل ساختار داده ای که لیست بسته شده نام دارد، گردد که در آن هر گره توسعه یافته ای ذخیره می شود. اگر گره جاری، تطبیقی با لیست بسته شده یافت، به جای توسعه دور انداخته می شود. الگوریتم جدید graph-search (شکل ۲-۱۸) نامیده می شود. با مسائلی که حالت مکرر تکراری دارند، graph-search بسیار کاراتر از tree-search است. در بدترین شرایط زمان و فضای مورد نیاز وابسته به اندازهٔ فضای حالت دارد. این ممکن است بسیار کمتر از $O(b^d)$ باشد.

```
function GRAPH-SEARCH (problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT (MAKE-NODE (INITIAL-STATE [problem]), fringe)
Ioop do
    if EMPTY (fringe) then return failure
    node ← REMOVE-FIRST (fringe)
    if GOAL-TEST [problem](STAT[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 ۲-۱۸: الگوریتم عمومی جستجوی گراف مجموعه بسته شده می تواند از طریق جدول پراکندگی پیاده سازی شود تا اجازه کنترل کارا برای حالات تکراری را بدهد. این الگوریتم فرض می کند که اولین مسیر به حالت S کمترین هم است.

توجه کنید استفاده از لیست بسته شده به معنی آن است که برای جستجوی اول- عمق و عمیق- شونده تکراری دیگر فضای خطی کفایت نمی کند. الگوریتم graph-search هر گرهی را در حافظه نگه می دارد و برخی محققان این کار را منطقی نمی دانند، چرا که محدودیت حافظه به وجود می آید.