جزوه درس هوش مصنوعی - جلسه پنجم

مروری بر فصل جست‌وجو و مسئله‌یابی

یادآوری از جلسه قبل

در جلسه گذشته وارد **فصل سوم: جست‌وجو (Search)** شدیم. ایده اصلی این فصل آن است که برخی مسائل را می‌توان به‌صورت یک **مسئله جست‌وجو** فرموله کرد؛ یعنی:

- یک **حالت آغازین (Initial State)** داریم،
- مجموعه‌ای از **اقدام‌ها (Actions)** در اختیار عامل است،
- یک **حالت هدف (Goal State)** یا مجموعه‌ای از هدف‌ها داریم،
- و می‌خواهیم **دنباله‌ای از اقدام‌ها** پیدا کنیم که ما را از شروع به هدف برساند.

جست‌وجوی آفلاین (Offline Search)

در این فصل عمدتاً با **جست‌وجوی آفلاین** سروکار داریم. یعنی عامل ابتدا در ذهن یا مدل خود، مسیرهای ممکن را بررسی می‌کند، **راه‌حل را پیشاپیش پیدا می‌کند** و سپس آن را اجرا می‌کند.

به بیان دیگر:

- اول **برنامه‌ریزی (Planning)** انجام می‌شود،
- بعد از یافتن راه‌حل مناسب، آن راه‌حل **اجرا (Execution)** می‌شود.

نوع عامل مناسب در این فصل

چون در این مسائل، هدف اصلی رسیدن به یک **هدف مشخص (Goal)** است، مناسب‌ترین مدل عاملی که تا اینجا با آن آشنا شده‌ایم، **عامل هدف‌محور (Goal-Based Agent)** است.

تمایز مهم: State-Space Graph و Search Tree

دو نمایش متفاوت از مسئله

در جست‌وجو باید میان دو مفهوم بسیار مهم تمایز قائل شویم:

1. **گراف فضای حالت (State-Space Graph)**
   نمایش مفهومی و اصلیِ مسئله است؛ در آن هر **حالت (State)** فقط یک‌بار وجود دارد.
2. **درخت جست‌وجو (Search Tree)**
   نمایش اجرایی و پیاده‌سازی‌شده‌ی فرایند جست‌وجو است؛ در آن ممکن است یک حالت بارها و بارها در قالب **گره‌های مختلف (Nodes)** ظاهر شود.

تفاوت State و Node

- **State**: یک وضعیت واقعی در فضای حالت مسئله
- **Node**: و اطلاعات جانبی دیگر state یک موجودیت پیاده‌سازی‌شده در درخت جست‌وجو که معمولاً حاوی یک است

چرا یک state ممکن است چند node داشته باشد؟

چون در پیاده‌سازی، هر node فقط خود state را نگه نمی‌دارد؛ بلکه معمولاً اطلاعات دیگری هم همراه آن است، مثل:

- عمق گره (**Depth**)
- هزینه مسیر تا آن گره (**Path Cost**)
- والد (**Parent**)
- اقدامی که باعث رسیدن به آن شده (**Action**)
- و سایر اطلاعات لازم برای اجرای الگوریتم

تکرار یک state در درخت جست‌وجو

ممکن است در **گراف فضای حالت**، حالت `G` فقط یک‌بار وجود داشته باشد، اما در **درخت جست‌وجو** همان حالت `G` از دو مسیر متفاوت به‌دست آید؛ در نتیجه، درخت جست‌وجو دو node متفاوت خواهد داشت که هر دو متناظر با همان state هستند.

Graph Search در برابر Tree Search

🕮 Tree Search

در **جست‌وجوی درختی (Tree Search)** اجازه می‌دهیم حالت‌های تکراری در روند جست‌وجو ظاهر شوند.
یعنی اگر از دو مسیر مختلف دوباره به یک state برسیم، هر دو را به‌عنوان nodeهای جدید می‌پذیریم.

🕮 Graph Search

در **جست‌وجوی گرافی (Graph Search)** مجموعه‌ای از stateهای بازدیدشده را در حافظه نگه می‌داریم و اجازه نمی‌دهیم stateهایی که قبلاً دیده‌ایم دوباره گسترش یابند.

⚠ اثر این تفاوت

تفاوت Tree Search و Graph Search بسیار مهم است، زیرا:

- Tree Search ساده‌تر است،
  - اما ممکن است باعث **تکرار زیاد، اتلاف زمان و حتی رشد نامتناهی درخت جست‌وجو** شود،
  - در حالی که Graph Search با ذخیره‌سازی stateهای دیده‌شده، از این تکرارها جلوگیری می‌کند.

🔒 جمع‌بندی

- در **Search Tree** ممکن است stateهای تکراری ببینیم.
- در **Graph Search** معمولاً با نگه‌داشتن یک مجموعه‌ی visited از تکرار جلوگیری می‌کنیم.

فرمول‌بندی یک مسئله جست‌وجو

🕮 اجزای اصلی یک مسئله جست‌وجو

هر مسئله جست‌وجو معمولاً با مؤلفه‌های زیر تعریف می‌شود:

1. **Initial State**: حالت آغازین
2. **Actions**: اقدام‌های مجاز در هر حالت
3. **Transition Model / Result Function / Successor Function**: مدلی که مشخص می‌کند با اجرای یک action از یک state به کجا می‌رویم
4. **Goal Test**: تابعی بولی که مشخص می‌کند آیا یک state هدف است یا نه
5. **Step Cost / Action Cost**: هزینه‌ی رفتن از یک state به یک state با یک action خاص دیگر

نسبت Transition Model و Successor Function

این اصطلاح‌ها در بسیاری از بحث‌ها نزدیک به هم به‌کار می‌روند:

- **Transition Model** معمولاً بر توصیف رابطه‌ی گذار تأکید می‌کند.
- **Result Function** چیست state در یک action می‌گوید نتیجه‌ی اجرای یک.
- **Successor Function** مجموعه‌ی جانشین‌های ممکن را تولید می‌کند.

در مسائل **قطعی (Deterministic)** این مفاهیم بسیار به هم نزدیک‌اند، اما در محیط‌های **تصادفی (Stochastic)** ممکن است با توصیف‌های احتمالاتی غنی‌تری مواجه شویم.

Goal Test

تابع **Goal Test** روی هر state اجرا می‌شود و به‌صورت `True/False` پاسخ می‌دهد که آیا آن حالت، حالت هدف است یا نه.

Goal Test می‌تواند صریح یا ضمنی باشد

تعریف هدف ممکن است:

- **صریح (Explicit)** باشد؛ مثلاً رسیدن به یک state با شماره مشخص
- یا **ضمنی (Implicit)** باشد؛ مثلاً «هیچ نقطه کثیفی باقی نمانده باشد» یا «دمای هوا کمتر از 25 درجه باشد»

Step Cost

تابع **Step Cost** هزینه‌ی حرکت از یک state به state دیگر را با انجام یک action می‌دهد.
هزینه می‌تواند به عوامل مختلفی وابسته باشد، مثل:

- فاصله
- زمان
- مصرف انرژی
- یا هر معیار مسئله‌محور دیگر

مفهوم راه‌حل (Solution)

راه‌حل در مسائل جست‌وجو

راه‌حل (Solution) یک **دنباله از اقدام‌ها (Action Sequence)** است که از حالت آغازین شروع می‌شود و ما را به یک حالت هدف می‌رساند.

راه‌حل هنوز اجرا نشده است

در جست‌وجوی آفلاین، این دنباله‌ی اقدام‌ها در ابتدا فقط یک **برنامه‌ی آینده** است؛ یعنی:

- هنوز اجرا نشده،
- ولی عامل آن را پیدا کرده،
- و بعداً آن را اجرا خواهد کرد.

راه‌حل خوب چیست؟

اگر چند راه‌حل وجود داشته باشد، مطلوب است الگوریتم جست‌وجو راه‌حلی را برگرداند که:

- کوتاه‌تر باشد،
- یا کم‌هزینه‌تر باشد،
- یا با معیار مطلوب مسئله بهترین باشد.

مثال کلاسیک: دنیای جاروبرقی (Vacuum World)

صورت مسئله

دنیای دو اتاقه

فرض کنید محیطی شامل دو اتاق داریم: **چپ** و **راست**.

عامل جاروبرقی:

- می‌تواند مکان خود را تشخیص دهد،
- می‌تواند تشخیص دهد اتاق فعلی تمیز است یا کثیف،
- و سه اقدام دارد:

- `Left`
- `Right`
- `Suck`

فضای حالت این مسئله

چون:

- مکان عامل دو حالت دارد،
- وضعیت هر اتاق نیز دو حالت دارد (تمیز / کثیف)،

در نتیجه کل فضای حالت این مسئله شامل **۸ حالت گسسته (Discrete States)** است.

چرا ۸ حالت؟

چون همه‌ی ترکیب‌های ممکنِ:

- موقعیت عامل
- تمیزی/کثیفی اتاق چپ
- تمیزی/کثیفی اتاق راست

را در نظر می‌گیریم:

$$8 = 2 \times 2 \times 2$$

تعریف هدف

Goal Test در Vacuum World

هدف این است که **هیچ اتاق کثیفی باقی نمانده باشد.**
بنابراین معمولاً دو حالت هدف داریم:

- عامل در اتاق چپ باشد و هر دو اتاق تمیز باشند
- عامل در اتاق راست باشد و هر دو اتاق تمیز باشند

جایگاه نهایی عامل مهم نیست

برای کارفرما معمولاً مهم نیست جاروبرقی در پایان دقیقاً در کدام اتاق بایستد؛ مهم این است که **خانه تمیز تحویل داده شود.**

گذارها در گراف حالت

رفتار اقدام‌ها

در این جهان:

- اگر عامل در چپ باشد و `Left` انجام دهد، در همان‌جا می‌ماند.
- اگر در چپ باشد و `Right` انجام دهد، به اتاق راست می‌رود.
- اگر `Suck` انجام دهد، اتاق فعلی تمیز می‌شود.
- اگر اتاق فعلی از قبل تمیز باشد و `Suck` انجام شود، تغییری در تمیزی رخ نمی‌دهد.

مقایسه‌ی دو محیط کاری: Observable و Unobservable

حالت اول: محیط قابل مشاهده

عامل با دو سنسور

اگر عامل:

- **سنسور مکان** داشته باشد،
- و **سنسور تشخیص کثیفی** هم داشته باشد،

آنگاه محیط برای او تا حد زیادی **Observable / Fully Observable** است و می‌تواند دقیق‌تر تصمیم بگیرد.

اثر مشاهده‌پذیری

اگر عامل بداند دقیقاً در کدام state قرار دارد، می‌تواند با تعداد اقدام‌های کمتری به هدف برسد.
مثلاً اگر در state مشخصی باشد، شاید فقط با توالی زیر به هدف برسد:

- `Right`
- `Suck`

حالت دوم: محیط غیرقابل مشاهده

عامل بدون سنسورهای اصلی

حالا فرض کنید عامل:

- نه سنسور مکان داشته باشد،
- نه سنسور تشخیص کثیفی.

در این حالت محیط برای عامل **Unobservable** می‌شود.

آیا چنین عاملی هنوز می‌تواند خانه را تمیز کند؟

بله.
نکته مهم این است که عامل اگرچه state دقیق فعلی را نمی‌داند، اما می‌داند که در یکی از stateهای ممکنِ فضای حالت قرار دارد.

شبه‌حالت (Belief-like / Set State)

در چنین وضعیتی، عامل به‌جای اینکه بداند «الان دقیقاً در کدام state هستم»، با **مجموعه‌ای از stateهای ممکن** کار می‌کند.
این مجموعه را می‌توان به‌صورت یک **شبه‌حالت** در نظر گرفت.

✨ آغاز کار در محیط غیرقابل مشاهده

اگر عامل هیچ مشاهده‌ای نداشته باشد، در ابتدا فقط می‌داند که در یکی از **۸ حالت ممکن** قرار دارد.
پس دانش اولیه او از جهان برابر است با:

«من در یکی از این ۸ state هستم.»

کاهش عدم قطعیت با انجام عمل

💡 *خودِ عمل هم اطلاعات می‌سازد*

حتی اگر عامل چیزی را حس نکند، اجرای یک action باعث می‌شود مجموعه‌ی حالت‌های ممکن تغییر کند.

مثلاً اگر عامل `Right` انجام دهد:

- دیگر نمی‌تواند در اتاق چپ باشد،
- پس بسیاری از حالت‌های ممکن حذف می‌شوند.

✨ تضمین رسیدن به هدف بدون مشاهده

در این مسئله، عاملی که هیچ حسگری ندارد هم می‌تواند با یک توالی تضمینی مثل:

- `Right`
- `Suck`
- `Left`
- `Suck`

تضمین کند که هر دو اتاق تمیز خواهند شد.

🔥 هزینه‌ی نداشتن سنسور

عاملِ بدون سنسور معمولاً:

- اقدام‌های بیشتری انجام می‌دهد،
- راه‌حل بلندتری دارد،
- اما هنوز می‌تواند به هدف برسد.

نتیجه

بودن محیط می‌تواند روی **Unobservable بودن** یا **Observable**:

- طول راه‌حل،
- هزینه راه‌حل،
- و نوع استدلال عامل

اثر مستقیم بگذارد.

برنامه‌ریزی بد و افتادن در حلقه

بودن تضمین کافی نیست Observable ⚠

حتی اگر محیط observable باشد، باز هم اگر عامل **بد برنامه‌ریزی شده باشد** ممکن است در **حلقه (Loop)** بیفتد.

تفاوت محیط و برنامه عامل

- **Observable / Unobservable** ویژگی محیط یا رابطه عامل با محیط است **بودن**.
- اما **حلقه افتادن یا نیفتادن** تا حد زیادی به طراحی خود الگوریتم جست‌وجو و برنامه عامل بستگی دارد.

یک راه جلوگیری از حلقه

اگر عامل بتواند stateهایی را که قبلاً دیده است به خاطر بسپارد و دوباره بی‌دلیل به آن‌ها برنگردد، احتمال افتادن در حلقه کمتر می‌شود.

مثال‌های دیگر از فضای حالت

پازل ۸ (Puzzle-8)

در پازل ۸، هر state یک چیدمان خاص از خانه‌ها و مربع خالی است.
اقدام‌ها بر اساس جابه‌جایی خانه‌ی خالی تعریف می‌شوند.
در این مسئله نیز می‌توان از **Graph Search** استفاده کرد تا stateهای دیده‌شده دوباره بررسی نشوند.

✨ مسئله شهرهای رومانی

در مسئله‌ی معروف رومانی:

- هر **شهر** یک state است،
- هر **حرکت به شهر همسایه** یک action است،
- **Transition Model** مشخص می‌کند با انتخاب یک جاده به کدام شهر می‌رویم،
- **Goal Test** مثلاً رسیدن به بخارست است،
- و **Action Cost** معمولاً فاصله‌ی بین شهرهاست.

🔥 مبدأ و مقصد هر دو مهم‌اند

در مسئله جست‌وجو:

- **حالت شروع** مهم است، چون مسیر از آنجا آغاز می‌شود.
- **حالت هدف** هم مهم است، چون تعیین می‌کند چه زمانی باید متوقف شویم.

Expand Function و Successor Function

🕮 Expand Function

وقتی در درخت جست‌وجو روی یک node عمل **expand** انجام می‌دهیم، فرزندان آن node تولید می‌شوند.

💡 *نقش Successor Function*

این تولید شدن فرزندان، بر اساس **Successor Function** یا همان مدل مسئله انجام می‌شود.
یعنی خود مسئله مشخص می‌کند از یک state چه جانشین‌هایی می‌توان ساخت.

🔥 تولید شدن با بازدید شدن فرق دارد

مهم است توجه کنیم که:

- با node کردن یک **Generate / Expand**
- کردن آن **Visit / Explore**

یکسان نیست.

ممکن است چند node تولید شده باشند، اما هنوز فقط یکی از آن‌ها را برای ادامه مسیر انتخاب کرده باشیم.

مرز جست‌وجو / Fringe / Frontier

Frontier

در هر لحظه از جست‌وجو، برخی گره‌ها:

- تولید شده‌اند،
- اما هنوز گسترش نیافته‌اند.

به این مجموعه می‌گوییم:

- **Frontier**
- یا **Fringe**
- یا **مرز جست‌وجو**

نقش frontier

در هر مرحله، الگوریتم با توجه به **استراتژی جست‌وجو** یکی از اعضای frontier را انتخاب می‌کند و آن را expand می‌کند.

دو مجموعه مهم در جست‌وجو

از یک نگاه کلی، گره‌ها را می‌توان به دو دسته تقسیم کرد:

- **Expanded / Explored**: گره‌هایی که قبلاً گسترش یافته‌اند
- **Frontier**: گره‌هایی که تولید شده‌اند ولی هنوز گسترش نیافته‌اند

استراتژی، شکل frontier را تعیین می‌کند

این‌که frontier چگونه رشد کند و کدام node بعدی از آن انتخاب شود، کاملاً به **استراتژی جست‌وجو** وابسته است.

الگوریتم کلی Tree Search

🕮 چارچوب عمومی جست‌وجوی درختی

در حالت کلی، الگوریتم Tree Search چنین رفتاری دارد:

1. درخت جست‌وجو را از **initial state** آغاز می‌کنیم.
2. اگر nodeای برای گسترش وجود نداشته باشد، **failure** برمی‌گردانیم.
3. با توجه به **strategy** یک **leaf node** را از frontier انتخاب می‌کنیم.
4. اگر این node حالت هدف باشد، **راه‌حل متناظر** را برمی‌گردانیم.
5. در غیر این صورت آن را **expand** می‌کنیم و فرزندانش را به درخت/مرز جست‌وجو اضافه می‌کنیم.
6. این فرایند تکرار می‌شود.

⚠ سرچ عمومی هنوز استراتژی را مشخص نکرده

این الگوریتم فقط اسکلت کلی را می‌دهد.
تفاوت اصلی الگوریتم‌ها در این است که:

- کدام leaf انتخاب شود،
- با چه ساختاری نگه‌داری شود frontier،
- و آیا stateهای تکراری مجاز باشند یا نه.

نخستین استراتژی: جست‌وجوی عمقی (Depth-First Search)

🕮 DFS

جست‌وجوی عمقی (Depth-First Search / DFS) همیشه **عمیق‌ترین گره موجود در frontier** را برای گسترش انتخاب می‌کند.

🔥 ساختار داده مناسب برای DFS

برای پیاده‌سازی frontier در DFS معمولاً از **پشته (Stack)** استفاده می‌شود:

- **LIFO**
- Last In, First Out

رفتار شهودی DFS

مثل عاملی است که می‌گوید DFS:

- یک مسیر را تا جایی که می‌شود ادامه بده،
- اگر به بن‌بست رسیدی، برگرد و مسیر بعدی را امتحان کن.

ویژگی حافظه‌ای DFS

حذف شاخه‌های بی‌فایده از حافظه

در DFS وقتی به گرهی می‌رسیم که:

- نه هدف است،
- و نه ادامه‌ی امیدبخشی دارد،

نگه‌داشتن آن در حافظه لازم نیست.
بنابراین DFS بسیاری از شاخه‌های بی‌فایده را از حافظه خارج می‌کند.

نتیجه

از نظر حافظه معمولاً **کم‌مصرف** است، چون مجبور نیست همه‌ی گره‌های سطوح قبلی را نگه دارد DFS.

ارزیابی DFS

چهار معیار ارزیابی الگوریتم‌های جست‌وجو

برای ارزیابی استراتژی‌های جست‌وجو معمولاً چهار ویژگی را بررسی می‌کنیم:

1. **Completeness**: آیا اگر جواب وجود داشته باشد، حتماً پیدا می‌شود؟
2. **Optimality**: آیا بهترین جواب را پیدا می‌کند؟
3. **Time Complexity**: چند گره باید تولید شوند؟
4. **Space Complexity**: چند گره باید در حافظه نگه داشته شوند؟

پارامترهای رایج تحلیل

معمولاً از نمادهای زیر استفاده می‌کنیم:

- **Branching Factor** : `b` یا میانگین/حداکثر تعداد فرزندان هر گره
- `m` : بیشینه عمق درخت جست‌وجو
- `d` یا `s` : عمق کم‌عمق‌ترین جواب

پیچیدگی زمانی DFS

اگر بیشینه عمق را `m` بگیریم، پیچیدگی زمانی DFS معمولاً:

`O(b^m)`

است؛ زیرا ممکن است تا عمق زیاد پیش برود و تعداد زیادی گره تولید کند.

پیچیدگی فضایی DFS

پیچیدگی فضایی DFS معمولاً:

`O(bm)`

است؛ چون فقط مسیر جاری و تعدادی از siblingهای لازم را نگه می‌دارد.

⚠ کامل بودن DFS

فقط در شرایطی **کامل** است که DFS:

- عمق فضای جست‌وجو متناهی باشد،
- و مسیر بی‌نهایت یا حلقه‌های مسئله‌ساز نداشته باشیم.

⚠ بهینگی DFS

DFS **بهینه نیست**.

ممکن است اولین جوابی که پیدا می‌کند:

- کم‌عمق‌ترین جواب نباشد،
- یا کم‌هزینه‌ترین جواب نباشد.

دومین استراتژی: جست‌وجوی عرضی (Breadth-First Search)

🕮 BFS

جست‌وجوی عرضی (Breadth-First Search / BFS) همیشه **کم‌عمق‌ترین گره frontier** را برای گسترش انتخاب می‌کند.

ساختار داده مناسب برای BFS

برای پیاده‌سازی frontier در BFS معمولاً از **صف (Queue)** استفاده می‌شود:

- **FIFO**
- First In, First Out

رفتار شهودی BFS

BFS لایه‌به‌لایه جلو می‌رود:

- ابتدا عمق 0
- سپس همه‌ی گره‌های عمق 1
- بعد همه‌ی گره‌های عمق 2
- و به همین ترتیب

ویژگی حافظه‌ای BFS

⚠ BFS حافظه‌بر است

برخلاف DFS، در BFS نمی‌توان به‌راحتی گره‌های سطوح قبلی را کنار گذاشت؛ چون الگوریتم برای ادامه‌ی پیمایش لایه‌ای به آن‌ها و فرزندانشان نیاز دارد.
به همین دلیل BFS معمولاً حافظه‌ی زیادی مصرف می‌کند.

ارزیابی BFS

پیچیدگی زمانی BFS

اگر کم‌عمق‌ترین جواب در عمق `s` باشد، BFS معمولاً تا آن عمق تقریباً همه‌ی گره‌ها را تولید می‌کند:

```
O(b^s)
```

پیچیدگی فضایی BFS

پیچیدگی فضایی BFS نیز معمولاً:

```
O(b^s)
```

است؛ چون frontier در لایه‌های پایین می‌تواند بسیار بزرگ شود.

📖 کامل بودن BFS

و وجود جواب، **کامل** است branching factor در صورت متناهی بودن BFS.

📖 بهینگی BFS

اگر همه‌ی step costها یکسان باشند، BFS **بهینه** است؛ چون کم‌عمق‌ترین جواب را پیدا می‌کند.

چرا Time Complexity را با تعداد گره‌های تولیدشده می‌سنجیم؟

💡 *معیار محاسباتی طبیعی*

در الگوریتم‌های جست‌وجو، هزینه‌ی زمانی تا حد زیادی صرف این می‌شود که:

- nodeها ساخته شوند،
- expand شوند،
- و اطلاعات مربوط به آن‌ها پردازش شود.

بنابراین تعداد nodeهای تولیدشده، معیار طبیعی و مناسبی برای سنجش زمان است.

🔥 تفسیر شهودی

هرچه الگوریتم:

- گره‌های بیشتری تولید کند،
- آزمون هدف بیشتری انجام دهد،
- و فرزندان بیشتری بسازد،

طبیعتاً زمان بیشتری مصرف می‌کند.

مقایسه شهودی DFS و BFS

✨ چه زمانی BFS بهتر است؟

اگر از قبل بدانیم که جواب:

- نزدیک به ریشه است،
- و در عمق کم پیدا می‌شود،

آن‌گاه BFS گزینه‌ی خوبی است.

✨ چه زمانی DFS بهتر است؟

اگر:

- جواب‌ها در عمق‌های زیاد باشند،
- یا حافظه محدود باشد،

آن‌گاه DFS معمولاً مناسب‌تر است.

⚠ مسئله‌ی عملی BFS

با وجود مزیت کامل‌بودن و بهینگی در هزینه‌های یکنواخت، BFS ممکن است در عمل به‌علت مصرف حافظه بسیار بالا غیرقابل استفاده شود.

یک مقایسه‌ی عددی مهم

رشد انفجاری هزینه‌ها

فرض کنید:

- branching factor برابر `10` باشد،
- در هر ثانیه بتوانیم `6^10` گره تولید کنیم،
- و هر گره به `1000` بایت حافظه نیاز داشته باشد.

در این صورت با افزایش عمق جواب، هزینه‌های زمانی و فضایی BFS به‌سرعت انفجاری می‌شوند.

✨ پیامد عملی

اگر جواب در عمق‌های بالا مثل 10، 12، 14 یا 16 باشد:

- زمان محاسبه بسیار زیاد می‌شود،
- و مهم‌تر از آن، حافظه‌ی لازم ممکن است به مقادیر غیرعملی مانند ترابایت و پتابایت برسد.

⚠ نتیجه‌ی مهندسی

بنابراین در انتخاب استراتژی جست‌وجو فقط **درست‌بودن نظری** مهم نیست؛ بلکه باید ببینیم:

- آیا از نظر زمان عملی است؟
- آیا از نظر حافظه شدنی است؟
- و آیا اصلاً با منابع موجود قابل اجراست؟

Depth-Limited Search

📖 جست‌وجوی عمقی محدودشده

برای کاهش مشکلِ گیر افتادن DFS در عمق زیاد، می‌توانیم یک حد عمق تعریف کنیم.
در **Depth-Limited Search** به DFS می‌گوییم:

«فقط تا عمق مشخصی پایین برو.»

✨ ایده‌ی حد عمق

اگر حد عمق را مثلاً `5` بگذاریم:

- DFS فقط تا عمق 5 پیش می‌رود،
- و پایین‌تر از آن را بررسی نمی‌کند.

⚠ مشکل اصلی

اگر جواب واقعی در عمق بیشتر از حد تعیین‌شده باشد، الگوریتم ممکن است **Not Found** برگرداند، در حالی که جواب واقعاً وجود داشته است.

چالش مهم

پرسش اساسی این است:

حد عمق را چند بگذاریم؟

- ؟5
- ؟20
- ؟100

اگر خیلی کم باشد، جواب‌های عمیق را از دست می‌دهیم.
اگر خیلی زیاد باشد، دوباره به مشکلات DFS نزدیک می‌شویم.

Iterative Deepening Search

جست‌وجوی تعمیق تکراری

Iterative Deepening Search (IDS) راه‌حلی برای مشکل انتخاب limit است.
ایده این است که DFS محدودشده را چند بار پشت سر هم اجرا کنیم:

- یک بار با limit = 1
- سپس limit = 2
- سپس limit = 3
- و به همین ترتیب

شهود IDS

این روش می‌خواهد:

- از **مزیت حافظه‌ای** DFS استفاده کند،
- و در عین حال به **رفتار لایه‌ای** BFS نزدیک شود.

روند اجرا

در IDS:

- ابتدا عمق 1 را کامل می‌گردیم،
- اگر جواب نبود، دوباره از ریشه با عمق 2 شروع می‌کنیم،
- بعد عمق 3،
- و همین‌طور ادامه می‌دهیم تا جواب پیدا شود.

⚠ پرسش طبیعی

در نگاه اول ممکن است به نظر برسد IDS دارد کارهای تکراری انجام می‌دهد، چون گره‌های بالایی را بارها دوباره تولید می‌کند.
این پرسش کاملاً طبیعی است و از نکات مهم تحلیلی IDS به شمار می‌رود.

نکته‌ی کلیدی

با وجود تکرار بعضی گره‌های سطوح بالا، IDS در بسیاری از مسائل هنوز بسیار کارآمد است، چون:

- تعداد گره‌های نزدیک ریشه کم است،
- و بار اصلی تعداد گره‌ها معمولاً در لایه‌های عمیق‌تر قرار دارد.

جمع‌بندی پایانی جلسه

آنچه در این جلسه آموختیم

در این جلسه با این مفاهیم کلیدی آشنا شدیم:

- مرور فصل جست‌وجو و مفهوم **Offline Search**
- تفاوت **State-Space Graph** و **Search Tree**
- تمایز **Tree Search** و **Graph Search**
- اجزای فرمول‌بندی یک مسئله جست‌وجو
- مفهوم **Solution** به‌عنوان دنباله‌ای از actionها
- تحلیل دقیق‌تر **Vacuum World**
- تفاوت محیط‌های **Observable** و **Unobservable**
- تعریف **Frontier / Fringe**
- ساختار کلی **Tree Search**
- آشنایی با **DFS**
- آشنایی با **BFS**
- مقایسه‌ی زمانی و فضایی DFS و BFS
- ایده‌ی **Depth-Limited Search**
- و معرفی **Iterative Deepening Search**

برداشت مفهومی مهم جلسه

انتخاب استراتژی جست‌وجو به‌شدت به ویژگی‌های مسئله وابسته است.
هیچ الگوریتمی برای همه‌ی مسائل بهترین نیست.
باید هم‌زمان به این‌ها توجه کرد:

- عمق جواب
- branching factor
- حافظه
- زمان
- کامل بودن
- و بهینگی

برای فکر کردن

1. چرا در محیط **Unobservable** هم ممکن است بتوان به هدف رسید؟
2. چرا **BFS** با وجود کامل‌بودن و بهینگی، در عمل همیشه بهترین انتخاب نیست؟
3. در چه شرایطی **DFS** ممکن است بسیار بد عمل کند؟
4. چرا **Iterative Deepening** تلاش می‌کند مزیت‌های DFS و BFS را با هم ترکیب کند؟