روش های سرچ

در این پروژه، یک مسئله جستجو طرح شده که با دو روش جستجو ناآگاهانه ی bfs و ids و یک روش جستجوی آگاهانه ی A* آن را حل میکنیم.

شرح مسئله بدین صورت است که یک جدول شامل تعدادی بیمار و تعدادی بیمارستان با ظرفیت مشخص داریم که یک آمبولانس با هل دادن بیمار ها از پشت، آن ها را جابجا می کند تا به یک بیمارستان با ظرفیت مثبت برسد. همچنین در جدول موانعی وجود دارد که آمبولانس و بیمار قادر به عبور از آن ها نیستند.

```
##########
#     #1 P #
#   #A###  #
# P###    P#
#    P  #  #
# 2     # 1#
##########
```

برای هر 3 روش باید درخت مسئله را به وجود آورده و از یک نود(استیت) ابتدایی شروع کرده وآن را به لیست Frontier خود افزوده و تا جایی که لیست خالی نشده و یا به نود هدف نرسیده ایم، با یک سیاست خاص یک نود از لیست برداشته و آن را به لیست دیده شده ها (Explored) اضافه کرده و بچه های آن را به 4 اکشن مختلف Up، Right، Left و Down ساخته و به لیست Frontier اضافه میکنیم و این روند را همانطور که گفته شد تا خالی شدن لیست فرانتیر که نشانه عدم موفقیت در رسیدن به هدف و یا رسیدن به نود هدف که بیانگر رسیدن به هدف است ادامه می دهیم. جزئیات مربوط به هر الگوریتم را در بخش های بعد مفصلا توضیح می دهیم.

نود ها چه هستند؟

نود ها ی درخت باید بیانگر هر استیت یا وضعیت مسیر حرکت آمبولانس در جدول باشند. یعنی هر نود مجزا یک وضعیت خاص از جدول از زمان شروع به حرکت آمبولانس را نشان می دهد.

از آنجا که متغیر های مسئله از ابتدای شروع بازی مختصات آمبولانس، مختصات بیمار ها و تعداد باقی مانده آن ها و مختصات بیمارستان ها به همراه ظرفیت باقیمانده شان می باشد، هر استیت یا نود با یک لیست که متشکل از یک زیر لیست مختصات آمبولانس اون استیت، یک زیر لیست حاوی مختصات بیماران اون استیت و یه دیکشنری که کلید های آن مختصات بیمارستان ها و مقادیر آنها ظرفیت های بیمارستان های اون استیت هستند نمایش داده می شود.

اکشن ها چه هستن؟

هر نود پدر برای به وجود آوردن نود های بچه هایش باید آمبولانس خود را به 4 جهت مذکور حرکت دهد تا ببیند چه نود بچه ای تولید می شود (ممکن است اصن نود جدیدی تولید نشود به این معنا که نود تولید شده یا در لیست Frontier یا Explored باشد). پس اکشن ها شامل 4 حرکت چپ، راست، بالا و پایین آمبولانس می باشند که با L,R,D و U نشان می دهیم.

Initial State و Goal State:

Initial state استیت شروعی است که حاوی مختصات ابتدایی آمبولانس، مختصات ابتدایی بیمارها و ظرفیت های ابتدایی بیمارستان هاست که همگی در یک لیست جمع شده اند و از فایل ورودی خوانده میشوند.

Goal State استیت پایانی است که در آن زیر لیست مربوط به مختصات بیمارها خالی باشد.در هر زمان که آمبولانس با یک اکشن بتواند یک بیمار را به داخل یک بیمارستان دارای ظرفیت هل دهد، مختصات مربوط به آن بیمار از زیر لیست بیمارها حذف می شود، پس هدف استیتی است که این زیر لیست در آن خالی باشد.

حال الگوریتم ها را به ترتیب یکی یکی شرح می دهیم:

1)الگوریتم ناآگاهانه BFS:

در این الگوریتم از یک لیست Frontier و یک لیست Explored و یک لیست دیگر به نام Fifo استفاده کرده ایم که هر یک را شرح می دهیم.

Frontier و Explored دو دیکشنری با کلید مختصات خانه های غیر مانع جدول هستند که value هر کلید یک لیست است. در واقع هنگام append کردن یک نود در Frontier و Explored به جای قرار دادن آن در یک لیست کلی، آن نود را در لیستی قرار می دهیم که کلید آن مختصات آمبولانس آن نود (استیت) است تا بعدا هنگام چک کردن تکراری نبودن یک child با یکی از نود های Frontier و Explored، به جای چک کردن لیست شامل کل نود ها، فقط لیست با کلید مختصات آمبولانس child بررسی شود (اگر نود child قرار باشد با یکی از نود های این Frontier یا Explored یکی باشد، آن نود قطعا در مختصات آمبولانس با child نیز یکی است و قبلا در لیستی که کلید آن مختصات آمبولانس child است ذخیره شده، پس کافی است فقط آن را بررسی کنیم که این کار زمان را کمتر و بهینه تر می کند).

در Fifo نود ابتدایی را قرار می دهیم و الگوریتم را بدین صورت ادامه می دهیم که تا زمانی که Fifo خالی نشده (عدم موفقیت) یا به استیت هدف (موفقیت) نرسیده ایم، نود اول Fifo را pop می کنیم، آن را برابر current node قرار داده و در Explored قرار می دهیم. حال به ازای هر اکشن، بچه ی current node را می سازیم (با تابع doAction) و در صورتی که در Frontier و Explore، در لیست مربوط به مختصات خانه آمبولانس بچه نود برابر با آن نبود، آن را در لیست مربوطه در Frontier اضافه می کنیم. البته قبل از بررسی مراحل مربوط به اضافه کردن بچه به Frontier، اگر بچه همان استیت هدف بود 1 را return می کنیم. اگر در هر بار ابتدای شروع مجدد حلقه بی نهایت، سایز Fifo صفر شده بود، 0 را return می کنیم.

2)الگوریتم ناآگاهانه IDS:

این الگوریتم را به روش بازگشتی پیاده سازی کردیم و شرح آن بدین صورت است:

تا زمانی که به استیت هدف نرسیده ایم، جستجوی dfs تا یک عمق ماکسیمم را به ترتیب صعودی انجام می دهیم. یعنی ابتدا یک dfs تا عمق 0 میزنیم. اگر به نود هدف رسیدیم،1 را return می کنیم. اگر

تلاش dfs برای رسیدن به هدف تا حداکثر عمق 0 موفقیت آمیز نبود، این بار dfs تا حداکثر عمق 1 انجام می دهیم و ... (برای جلوگیری از لوپ

اگر dfs با حداکثر عمق n میزنیم، مطمئن هستیم که نود هدف در عمق های 0 تا n - 1 نیست، چرا که اگر بود در dfs قبلی پیدا می شد. پس برای بهینه کردن زمان، بهتر است هرگاه dfs تا یک عمق مشخص می زنیم، نود های سطر آخر را که قبلا ندیده بودیم با doAction ساخته و در صورت هدف نبودن به لیست فرزندان پدرشان اضافه کنیم تا در dfs بعدی، دوباره به محاسبه آن ها نپردازیم و بچه ها را از لیست فرزندان پدر به دست آوریم (همواره نود های سطر آخر توسط تابع doAction که قبل توضیح داده شد ساخته می شوند چون اولین بار است که تا آن عمق سرچ می کنیم و قبلا آنها را در لیست فرزندان پدرشان نداریم ولی مابقی فرزندان در dfs های قبلی با doAction ساخته شده اند و در لیست فرزندان پدرشان هستند و با یک for میتوانیم روی لیست ای فرزندان حرکت کنیم).

پس برای یک سطح مشخص n،

dfs(n) را صدا میزنیم. اگر n > 1 بود یعنی فرزندانش قبلا ساخته شده اند و در لیست فرزندانش قرار دارند پس به ازای هر یک از آن ها با حلقه for، دوباره dfs(n -1) را صدا میزنیم تا به صورت شاخه ای در عمق پایین برویم. اگر n مساوی 1 شد، یعنی به سطر یکی مانده به آخر رسیده ایم و احتیاج است فرزندان را تک تک با doAction ساخته و در لیست فرزندان پدر اضافه کنیم و مانند قبل به ازای هر فرزند dfs(n -1) را صدا بزنیم که چون n - 1 برابر 0 شده است، پس به سطر آخر رسیده ایم و باید هدف بودن این نود سطر آخر را بررسی کنیم (گفتیم که هدف قطعا در سطر آخر است).

در صورت هدف بودن true و در غیر آن صورت false را برمیگردانیم.

توجه شود که کلاس نود دارای یک فیلد لیست Children است که در طی dfs های مختلف فرزندان هر پدر در صورت لزوم به آن اضافه می شوند، هم چنین چون تابع dfs بازگشتی تعریف شده، در یک شاخه پایین می رویم تا به سطر آخر آن برسیم که در صورت false بودن نود سطر آخر پله پله بالا می آییم و منطق dfs پیاده سازی می شود.

3) الگوریتم A*

این الگوریتم همانند bfs است با این تفاوت که در هر مرحله به جای pop کردن اولین نود ، نودی را pop می کنیم که دارای شرایط خاصی باشد که در پایین توضیح می دهیم.

هر نود دارای یک heuristic است که فاصله آن نود تا نود هدف را تخمین میزند. هر نود همچنین دارای یک هزینه از نود ابتدایی تا خودش است که آن را با g نشان می دهیم و مجموع heuristic و g برای هر نود را f نامیده و در هر مرحله انتخاب از Fifo نودی را انتخاب می کنیم که f کمتری دارد.

Heuristic انتخاب شده باید admissible باشد بدین معنا که برای هر نود مقدارش از هزینه واقعی آن نود تا نود هدف کمتر باشد.

همانطور که گفتیم الگوریتم تقریبا شبیه bfs است. یکی از تفاوت های آن با bfs این است که اگر یک نود پدر بعد از پاپ شدن از Fifo و اضافه شدن به Explored، با هر اکشن بچه ای تولید کرد باید نگاه کنیم و ببینیم که در لیست فرانتیر اگر نود مانند بچه وجود داشت ولی f بیشتری داشت، مقدار f آن را با f بچه که کمتر و بهتر است آپدیت کنیم.

هیوریستیک admissible انتخاب شده برای این پروژه، مجموع فاصله (manhattan distance) بیمار ها تا نزدیک ترین بیمارستان بهشون می باشد.

این هیوریستیک admissible است به این علت که در هر استیت یا نود، برای رسیدن به نود هدف کمترین و ایده آل ترین کار رسوندن هر بیمار به نزدیک ترین بیمارستان بهش از مسیر manhattan distance بینشان می باشد (این حالت بهترین و حداقل ترین کار لازم است چون در بین راه موانع وجود دارند و ظرفیت بیمارستان ها محدود است از هزینه واقعی قطعا کمتر است پس admissible بودن آن ثابت می شود).

در هر اکشن، حداکثر یک بیمار استیت پدر یک خانه به یک جهت هول داده می شود، پس مجموع فواصل بیمار ها تا نزدیک ترین بیمارستان بهشون نهایتی یکی زیاد یا کم می شود. حال بسته به آنکه بیمار هول داده شده به بیمارستان نزدیک به خود یک قدم نزدیکتر یا دورتر شود، هیوریستیک پدر به ترتیب یکی کم یا زیاد می شود و در صورتی که بیمار بعد از یک خانه هول داده شدن به یک بیمارستان دیگر نزدیک تر

شود (نزدیکترین بیمارستان به آن تغییر کند)، پس قطعا نسبت به حالت قبل خود که به بیمارستان قبلی نزدیکترین بود به بیمارستان حدید نزدیک تر است و هیوریستیک پدر یکی کم میشود.

در نهایت هزینه رسیدن از پدر به هر بچه را 1 در نظر گرفته ایم و مبنای آن را یک خانه حرکت کردن آمبولانس گرفتیم. پس بدین ترتیب هیوریستیک و هزینه (g) هر نود بچه حساب می شود و در صورتی که بچه قبلا در لیست فرانتیر یا اکسپلورد نبود، به فرانتیر اضافه شده و برای راند بعد در داوری برای انتخاب کمترین f بین اعضای fifo قرار میگیرد.

توجه شود که نیازی برای بررسی وجود یک عضو تکراری با بچه در لیست Frontier نیست که علت آن هم این میباشد که f نود بچه قطعا از f نود مانند آن در Frontier بیشتر است. علت این است که دو نود هیوریستیک برابر دارند(چون هیوریستیک طبق تعریف به تعداد و مختصات بیماران آن نود(استیت) ربط دارد و دو نود مشاه ازین نظر برابرند).و g بچه قطعا از g نود مشابه در Frontier بیشتر است (چون هزینه هر استپ از پدر به فرزندی یک است و این نود بچه بعد تر از نود مشابه در Frontier دیده شده پس در عمق بیشتری می باشد و g بیشتری دارد) پس در مجموع f بچه از f نود مشابه کمتر است و نیاز به آپدیت نود قبلی نیست.

تفاوت دیگر الگوریتم با bfs در این است که در اینجا هدف بودن یک نود را بعد از پاپ کردن از Fifo و قبل بسط بچه هایش بررسی می کنیم اما در bfs این عمل در هنگام ساختن هر بچه و قبل افزودن آن به لیست Frontier انجام میگیرد.

هیوریستیک دیگر که admissible نیست، مجموع فواصل آمبولانس تا بیمار ها می باشد که به وضوح admissible نیست(به این علت که شاید بیمارستانی خود را به یک بیمار رساند و در بیمارستانی قرارش داد اما مثلا بیمار های دیگر درست در نزدیکی بیمارستان باشند و آمبولانس آن ها را نیز داخل بیمارستان کند در حالی که هیوریستیک گفته شده هزینه را بیشتر از واقعیت محاسبه می کند و جواب بهینه نمیدهد.

تفاوت ها و شباهت ها

- تفاوت bfs با ids در نوع حرکت به سمت هدف است (bfs افقی اما dfs در شاخه و در عمق حرکت می کند و در صورت عدم موفقیت پله پله برمیگردد و بالا می آید)

- Bfs time complexity = o(b.pow(d)) Bfs space = o(b.pow(d))
- IDS time complexity = o(b.pow(d)) IDS space = o(b.pow(d))
- A* time complexity = depends on Heuristic chosen
- A* space = o(b.pow(d))

Bfs همه نود ها را در یک مموری نگه می دارد. پس اگر branching factor را با b و عمق جواب را که بهینه هم می باشد با d نشان دهیم، حداکثر مقدار نود های سطر جواب ((b.pow(d)) پیچیدگی حافظه و زمان است (bfs سطر سطر نود ها را ویزیت کرده و بچه هایشان را به مموری اضافه می کند و تا عمق جواب پیمایش را ادامه می دهد پس پیچیدگی حافظه و زمان یکسان و برابر مقدار گفته شده دارد).

A* هم کارکرد مشابه به bfs دارد با این تفاوت که بجای آنکه اولین نود مموری را پاپ کند(اولویت را به سطر سطر پیش رفتن بدهد)، بهترین نود انتخاب شده توسط معیار جمع هزینه و هیوریستیک را پاپ می کند(به سمت جواب پیش می رود) اما پیچیدگی زمان آن وابسته به هیوریستیکی است که انتخاب می کنیم و نمیتوان فرمولی برای آن گفت. در مورد حافظه، مانند bfs از یک مموری استفاده می کنیم که فرزندان هر پدر در هر مرحله به آن افزوده می شوند و حافظه o(b.pow(d)) دارد.

نکته: با مشاهده تعداد استیت های مشاهده شده می توان فهمید که الگوریتم A* در زمان بهتری نسبت به بقیه کار می کند اما همچنان مشکل حافظه را تا حدی دارد.

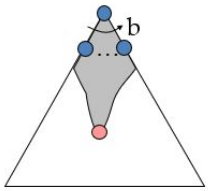
در مورد ids هم از آنجا که یک لیست داریم که نود های ویزیت شده را در آن نگه میداریم تا از لوپ جلوگیری کنیم، در زمانی که به سطر جواب (عمق d) میرسیم، همه نود های سطر های قبل رو در لیست نگه داشته ایم پس پیچیدگی حافظه ی o(b.pow(d)) داریم و هزینه زمان آن:

$$(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d$$

می باشد که یعنی قبل رسیدن به جواب سطر 0 را 1 + d بار، سطر 1 را d بار و .... و سطر جواب را یک بار ویزیت و iterate کرده ایم (پیچیدگی زمانی کل همان o(b.pow(d)) می شود)

نکته: مزیت ids نسبت به Bfs در حالت کلی حافظه مصرفی کم آن است. اگر برای پروژه زمان در اولویت نبود، فقط با نگهداری حافظه شاخه ای از درخت که در آن هستیم می توانستیم در زمان زیاد اما با حافظه کم به جواب برسیم (روش اولی که به کار بردم این بود که در هر بار صدا زدن تابع بازگشتی نود ویزیت شده را به لیست اد کنم و موقع بازگشت آن نود از تابع آن را از لیست پاپ کنم که حافظه همان شاخه را نگه می داشت اما از نظر زمانی الگوریتم مناسبی نبود).

A*



رفتن به سمت هدف در الگوریتم A*

حال جدول را کامل می کنیم.

	فاصله جواب	تعداد استیت دیده شده	تعداد استیت مجزای دیده شده	زمان اجرا
BFS	تست 1: 11 تست 2: 27 تست 3: 39	تست1: 989 تست 2: 37834 تست 3: 86483	تست 1: 469 تست 2: 16181 تست 3: 34466	تست 1: 0:00:00.0129 تست2: 0:00:04.6555 تست3: 0:00:17.2684
IDS	تست1: 11 تست2: 27 تست3: 39	تست1: 989 تست 2: 37834 تست 3: 86483	تست1: 469 تست 2: 16181 تست 3: 34466	تست 1: 0:00:00.0180 تست 2: 0:00:10.5111 تست 3: 0:00:31.8933
A*(1)	تست1: 11 تست 2: 27 تست 3: 39	تست1: 434 تست 2: 20862 تست 3: 31165	تست1: 210 تست 2: 9142 تست 3: 13039	تست1: 0:00:00.0152 تست2: 0:00:04.7499 تست3: 0:00:07.1651
A*(2)	تست1: 11 تست 2: 27 تست 3: 39	تست1: 379 تست 2: 17036 تست 3: 28521	تست1: 191 تست 2: 7607 تست 3: 12596	تست1: 0:00:00.0124 تست2: 0:00:04.1697 تست3: 0:00:07.6684