گزارش پروژه میانترم درس مبانی هوش مصنوعی

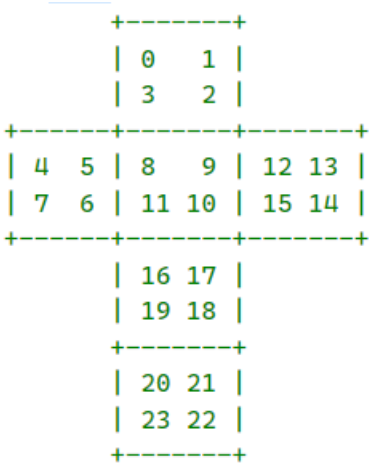
محمدرضا غفرانی ۹۶۳۱۰۵۳

شیوه ذخیره کدها و فایل‌ها به این شکل است که در پوشه با نام `p1` تمامی کد های مربوط به سوال ۱ و در پوشه با نام `p2` تمام کدهای مربوط به سوال ۲ قرار داده شده است. همچنین در پوشه `p1` سه پوشه `p1a`، `p1b` و `p1c` قرار دارد که هر یک متعلق به هر یک از قسمت‌های سوال یک هستند. همچنین در پوشه `p2` نیز نمودار های قسمت اول در `p2a` و نمودارها و جواب‌های قسمت دوم در `p2b` قرار دارد

سوال ۱

قسمت اول

در این قسمت مکعب روبیک به صورت یک آرایه ۳۶ تایی مدل سازی شده‌است. ترتیب قرار گرفتن اعضا کنار هم طبق شکل زیر است:



پیاده‌سازی این قسمت مطابق با شبه کدی که در اسلایدهای درس در رابطه با `IDS` بود، انجام شده‌است. بدین صورت که یک تابع به نام `depth_limited_search` تعریف شده است. این تابع سه ورودی می‌گیرد، روبیکی که باید روی آن کار کند، حداکثر عمقی که باید برود، و جوابی که تاکنون برای مسئله یافته است. این تابع در واقع روی روبیک مورد نظر `dfs` می‌زند با این تفاوت که از ابتدا عمق را بررسی می‌کند که به عمق مورد نظر رسیده است یا خیر و همچنین تا زمانی که به حداکثر عمق مجاز نرسیده بود کار را ادامه داده و خود را دوباره صدا می‌زند. در هر بار فراخوانی دوباره یک واحد از `limit` کم می‌کند. در پیاده‌سازی این بخش به صورت درختی پیش رفته‌ایم زیرا که حالت تکراری را با یک واحد کاستن از `limit` صدا می‌زند و در نتیجه احتمال افتادن در یک حلقه بینهایت وجود ندارد. همچنین اگر پیاده سازی این بخش به صورت درختی انجام نمی‌شد، دیگر الگوریتم کامل نبود و ممکن بود از مسیر دورتری به هدف برسد و یا اصلا نتواند هدف را بیاید.

قسمت دوم

در این قسمت یک کلاس `Node` تعریف شده است، که مقادیر زیر را ذخیره می‌کند:

- `Cube`: مکعب روبیک مربوط آن گره که همانند قسمت قبلی مقادیر روبیک را در یک آرایه ۳۶تایی نگهداری می‌کند.

- **Parent**: والدی که با چرخاندن آن **Node** فعلی به وجود آمده است.
- **Phase**: وجهی از **Node** والد که با چرخاندن آن در جهت **direction** نود فعلی به وجود آمده است.
- **Direction**: جهتی که اگر آن **Phase** از **Node** والد را درآن جهت بچرخانیم **Node** فعلی به وجود می‌آید.

عمده کار در قسمت **bidirectional search** رخ می‌دهد. این تابع که براساس شبه کدی که در اسلاید های درس وجود دارد پیاده‌سازی شده است، بدین شکل عمل می‌کند که تمامی نود های هدف را به یک نود اولیه وصل کرده که با گسترش دادن آن تمامی حالت های هدف تولید می‌شود. منظور از **qf** همان Q_f و منظور از **qb** همان Q_b در اسلاید های درس است. همچنین **ef** و **eb** به ترتیب **explored set** مربوط به قسمت پیش‌رونده و عقب‌گرد است. شیوه کار نیز به صورت **BFS** غیر همزمان است که در آن ابتدا تمامی گره های مربوط به یک حالت ساخته شده و اگر در **explored set** مربوطه نباشند به **qf** و با **qb** بر اساس این که مربوط به قسمت پیش‌رونده و یا عقب‌گرد هستند، اضافه می‌شوند. سپس یک **Node** از **qf** برداشته می‌شود و چک می‌شود که در **eb** وجود دارد یا خیر. اگر وجود داشت قسمت مربوط به یافتن مسیر فعال می‌شود و در غیر این صورت همین عملیات برای **qb** و **ef** تکرار می‌شود تا به جواب برسیم.

قسمت سوم

در این قسمت نیز کلاس **Node** همانند قسمت قبلی است. در این قسمت الگوریتم **A*** مطابق با شبه‌کد موجود در اسلایدهای درس پیاده‌سازی شده است. نکته قابل توجه در اینجا آن است که هزینه هر گام برابر ۸ در نظر گرفته شده است، تا هیوریستیک داده شده قابل قبول باشد. در این روش نیز یک لیست الویت‌دار ساخته می‌شود که در اینجا ما یک لیست ساده در نظر گرفته‌ایم ولی هنگام برداشتن از آن **Node** را انتخاب می‌کنیم که کمترین **f** را داشته باشد. به همین شیوه ادامه می‌دهیم تا به جواب برسیم و با دیگر **Node** در لیست ما وجود نداشته باشد و ما هنوز به هدف نرسیده باشیم که در این صورت می‌گوییم مسئله جواب نداشته است.

- در قسمت اول تعداد گره های تولید شده در تابع **rotate** یک واحد زیاد می‌شود چون هنگامی که این تابع فراخوانی می‌شود یک مکعب جدید با چرخاندن مکعب قبلی تولید می‌شود. در قسمت دوم و سوم این عبارت را در **constructor** مربوط به کلاس **Node** گذاشته‌ایم گرچه می‌شد در قسمت **rotate** نیز قرار داد.
- در قسمت اول تعداد گره‌های بسط داده شده هنگامی شمرده می‌شود که تابع **depth_limited_search** فراخوانی می‌شود چون همانطور که گفته شد، این تابع یک مکعب را گرفته و شروع به پردازش آن می‌کند، بنابراین باید در این قسمت باشد. در قسمت دوم بعد از هنگامی که یک **Node** از **qb** و یا **qf** برداشته شد یک واحد زیاد می‌شود چون در اینجا عملیات پردازش روی آن **Node** آغاز می‌شود. در قسمت سوم هنگامی که یک **Node** از مجموعه **frontier** برداشته شد به تعداد گره های بسط داده شده یک واحد اضافه می‌شود چرا که در اینجا نیز عملیات پردازش **Node** آغاز می‌شود.
- عمق جواب در قسمت اول در برابر **limit**در تابع **depth_limited_search_decorator** است. و هنگامی که مقدار آن زیاد می‌شود مقدارش زیاد می‌شود. در قسمت دوم و سوم چون با استفاده از **parent** به مسیری که منجر به حل روبیک می‌شود می‌رسیم بنابراین در آن تابع (یعنی تابع **find_solution**) در هر حلقه یک بار به مقدار عمق جواب اضافه می‌کنیم.

- حداکثر تعداد گره‌های ذخیره شده در بخش اول برابر مقدار `limit` است چون در هر بار فراخوانی تابع صرفا یک حالت در حافظه ذخیره می‌شود و تابع بازگشتی فراخوانی می‌شود. در بخش دوم حداکثر تعداد گره‌های ذخیره شده برابر جمع تعداد اعضای `eb` و `ef` است. از آنجا یک `Node` جدید هم به `q` و هم به `e` مربوط به آن بخش اضافه می‌شود بنابراین در `e` شمرده می‌شود و نیازی به شمردن دوباره در `q` نیست. همچنین چون عضوی که وارد `e` شد از آن خارج نمی‌شود، بنابراین هنگامی که یک عضو به این مجموعه اضافه می‌شود مقدار حداکثر تعداد گره‌های ذخیره شده را یکی زیاد می‌کنیم. در بخش سوم یک مجموعه `frontier` داریم که `Node`های موجود در حافظه را نگهداری می‌کند هنگامی که `Node` جدیدی را به آن اضافه می‌کنیم بر حداکثر تعداد گره‌های موجود در حافظه یکی اضافه می‌کنیم و هنگامی که از `frontier` یک `Node` را برای گسترش انتخاب کردیم از حداکثر تعداد گره‌های موجود در حافظه یکی کم می‌کنیم چرا که دیگر آن مقدار در حافظه ذخیره نمی‌شود.

این الگوریتم‌ها روی مثال‌های مختلفی اجرا شده است. نتایج اجرای آن‌ها در فایل‌های `txt` به همان نامی که کد به آن نام نام‌گذاری شده است، می‌باشد. تعداد گره‌های ذخیره شده در قسمت اول نسبت به قسمت دوم و سوم کمتر است چرا که در این قسمت به صورت `dfs` عمل می‌کنیم ولی در سایر روش ها به روشی مشابه `bfs` و همانطورکه می‌دانیم حافظه مورد استفاده برای ذخیره‌سازی `dfs` به صورت خطیست در حالی که حافظه مورد نیاز برای `bfs` به شکل نمایی است. با توجه به آنکه تمامی الگوریتم های داده شده کامل و بهینه هستند بنابراین عمق جوابی که هر یک از الگوریتم‌ها پیدا کرده‌اند یکسان بوده و تفاوتی ندارند. تعداد گره‌هایی که در قسمت `ids` تولید شده‌اند و همچنین تعداد گره‌های گسترش یافته نسبت به `bidirectional` در ابتدا کمتر است ولی در ادامه ممکن است بیشتر نیز بشود چرا که `ids` تا یک عمق مشخص پیش می‌رود و حداکثر می‌تواند همان‌ها را تولید کند ولی در ادامه چون اولا هر دفعه گراف را از ابتدا می‌سازد و با توجه به آنکه گراف دارای حجم گسترده‌تری است بنابراین تعداد گره های تولید شده این الگوریتم افزایش می‌یابد. الگوریتم `ucs` نیز در ابتدا نسبت به `bidirectional` تعداد گره‌های کمتری تولید می‌کند ولی در ادامه تعداد گره‌های تولید شده توسط این روش افزایش می‌یابد. زمانی که تعداد مراحل رسیدن به هدف کم است با توجه به اینکه الگوریتم `bidirectional` در ابتدا تمامی حالت‌های هدف را ساخته و یک به یک شروع به بسط دادن می‌کند بنابراین در ابتدا تعداد گره‌های تولید شده و گسترش داده‌شده نسبت به الگوریتم `usc` بیشتر خواهد بود ولی با افزایش عمق جواب چون الگوریتم `bidirectional` از دو طرف سعی می‌کند یک مسیر به هدف را بیابد، در نتیجه گره‌های کمتری را گسترش می‌دهد. ولی در الگوریتم `usc` که در این‌جا چون هزینه هر گام ثابت است، مانند `bfs` ساده عمل می‌کند و حتی امکان دارد یک گره را چند بار بسازد و گسترش دهد در نتیجه تعداد گره‌های تولید شده آن نسبت به `bidirectional` بیشتر می‌شود.

سوال ۲

قسمت اول

نمودار هایی که خواسته شده است در پوشه `p2a` ذخیره شده است. شیوه نامگذاری فایل با یک مثال توضیح داده می‌شود. اگر اسم فایل `g#50,p#10,t#2,m#0.01` منظور آن است که تعداد نسل‌ها برابر ۵۰، اندازه جمعیت برابر ۱۰، `tornumentSize` برابر ۲ و نرخ جهش برابر ۰.۰۱ است. همانطور که از نمودار ها برمی‌آید هر چقدر تعداد نسل‌هایی که برنامه به ازای آن‌ها اجرا می‌شود را بیشتر کنیم جمعیت رفته رفته بهتر می‌شود، اما به نظر می‌رسد که جمعیت پس از مدتی به مقدار مشخص همگرا می‌شود و نمی‌توان تمام افراد نامناسب را از جمعیت حذف کرد. یکی از علل این کار آن است که امکان جهش در جمعیت وجود دارد و همواره ممکن است یک عضو نامناسب از والدین اولیه به وجود آید. دیگر عامل تاثیرگذار در جمعیت، اندازه جمعیت است که همانطور که در نمودارها دیده می‌شود هر

اندازه، طول جمعیت بیشتر باشد، همان قدر جامعه زود تر و بهتر رشد می‌کند. علت این رخداد آن است که در جمعیت‌های کوچک اگر یک عوض نامناسب به وجود آید، با توجه به اینکه طول جمعیت کم است، احتمال خراب کردن جمعیت را بیش‌تر از زمانی است که طول جمعیت زیاد است. عامل دیگر دخیل در این کار اندازه تورنمت است. این عامل نیز مانند عوامل قبلی هر اندازه برزگ‌تر باشد جامعه زودتر به سمت کمال حرکت می‌کند. اگر اندازه تورنمت کم باشد احتمال اینکه یک عضو نامناسب در تورنمت وجود داشته باشد و همچنین به عنوان یک عضو برنده انتخاب شود، زیاد خواهد بود و در نتیجه باعث می‌شود که جامعه دیرتر به کمال برسد. عامل دیگر موثر در این روند نرخ جهش است. همانطور که نمودارهای کشیده شده نیز بیانگر این موضوع هستند. این افزایش این عامل برخلاف عوامل قبلی موجب حرکت سریع‌تر جامعه به سمت کمال نمی‌شود. البته این احتمال نیز وجود دارد که چنین اتفاقی نیز بیفتد و ناگهان یک جهش در یک جامعه رخ دهد. ولی این عامل از دو جنبه عمل می‌کند: اول اینکه ممکن است یک جهش رخ داده و موجب تولید فرزندان بهتر شود و از طرف دیگر ممکن است جهش موجب از بین رفتن یک نمونه خوب بشود. اما همانطور که از نمودار ها برمی‌آید این عامل بیشتر در جهت عکس عمل کرده و مانع رسیدن جامعه به کمال می‌شود. همچنین تاثیر این عامل در جامعه هایی با اندازه کوچک تر بهتر دیده می‌شود و موجب شکستگی در نمودار می‌شود.

قسمت دوم

در این قسمت باید تابع شبیه‌سازی ذوب فلزات را انجام می‌دادیم. بر اساس اسلایدهای درس این الگوریتم ترکیبی از الگوریتم `Random Walk` و `Hill Climbing` است. در ابتدا که دما زیاد است احتمال آنکه از یک حالت خوب به یک حالت بد برود زیاد است و با این روش از ماکسیمم های محلی فرار می‌کند ولی در ادامه و با کاهش دما تنها حالت‌های بهتر را برمی‌گزینند و بدین طریق سعی می‌کند که خود را به بیشینه سراسری برساند. فایل های این مجموعه در فایل `p2b` به صورت زیر ذخیره می‌شود:

```
sched#{sched},T0#{T0},alpha#{ALPHA}.png
```

که در اینجا آرگومان اول نوع تابع استفاده شده، دومین آرگومان `T0` که در تابع به عنوان ثابت در نظر گرفته‌شده‌است و سومین مقدار ثابت `α` در توابع دمای داده شده است.

شکل زیر نمودار چهار تابع داده شده است. همان طور که مشاهده می‌شود تابع اول به طور ملایمی پایین می‌آید ولی سایر توابع به خصوص تابع دوم از یک جایی به بعد آن قدر کند تغییر می‌کنند که می‌شود گفت از یک جایی به بعد به خط راست تبدیل می‌شوند. در نمودارهای مربوط به تابع اول در ابتدا که دما زیاد است امکان انجام انتخاب یک حالت بد به عنوان حالت بعدی وجود دارد و به طور متوسط در هر یک، در نمودار یک شکستگی دیده می‌شود که ناشی از همان حرکت تصادفی است. همانطور که گفته شد نمودار تابع دوم از یک جایی به بعد خطی می‌شود و تا از آن‌جا به بعد تابع تغییر چندانی پیدا نمی‌کند بنابراین اگر آن‌جا کم‌تر از آستانه‌ای باشد که ما از آنجا به بعد را صفر در نظر می‌گیریم، بنابراین الگوریتم در طی چند حرکت به پایان می‌رسد و اگر این گونه نباشد به علت بالا بودن نسبی دما احتمال انتخاب حالت‌های نامناسب زیاد می‌شود و در نتیجه نمودار شکستگی زیادی خواهد داشت و همچنین نقاط زیادی بررسی خواهد شد. تابع سوم نسبت به تابع اول شیب نزولی بیشتری دارد و پس از مدتی مانند آن رفتار خطی‌گونه‌ای از خود نشان می‌دهد ولی به اندازه آن سریع به حالت خطی نمی‌رسد و در حالت خطی نیز شیب نزول بیشتری نسبت به آن دارد. بنابراین در نمودارهای مربوط به آن در ابتدا شکستگی‌هایی دیده می‌شود ولی در ادامه این شکستگی‌ها رفته‌رفته کم شده و به صفر می‌شود و نمودار حالت خطی در ماکسیسم پیدا می‌کند. تابع چهارم نیز همان تابع قسمت سوم است با این تفاوت که مخرج آن سریع‌تر رشد می‌کند و در نتیجه اگر آستانه‌ای را که از آن پایین‌تر

را صفر در نظر می‌گیریم را همانند قسمت قبلی در نظر بگیریم، تعداد حالت‌های کمتری نسبت به حالت قبلی بررسی می‌شود و در نتیجه احتمال رسیدن به حالت بهینه در زمان‌هایی که با تنظیم شدن α و $T0$ شیب نزولی‌بودن الگوریتم به شدت زیاد می‌شود خیلی‌کم می‌شود چرا که تعداد نقاط کمتری را بررسی می‌کند.