گزارش پیاده‌سازی *pacman*

در این پیادسازی از لینک https://inst.eecs.berkeley.edu/~cs188/fa18/project1.html و کدهای آن استفاده شده است.

در ابتدا برای شبیه سازی نقشه داده شده در سوال فایل maze.lay ایجاد شده است که در آن '%' نماد دیوار و 'P' نماد pacman و '.' نماد نقطه (هدف بازی) است.

الگوریتم‌های خواسته شده برای یافتن مسیری به هدف ('.' مشخص شده در نقشه) در فایل search.py پیاده‌سازی شده اند که همگی تنها یک ورودی به نام problem دارند که اطلاعات سوال در آن قابل دسترس است. problem یک object از کلاس PositionSearchProblem است که در فایل searchAgent.py پیاده شده است و که خود از کلاس SearchProblem در فایل search.py ارث بری می‌کند. در این object متدهایی وجود دارد که به پیاده‌سازی الگوریتم‌های خواسته شده کمک به سزایی می‌کنند.

قبل از بررسی متدها به معرفی برخی از متغیرها می‌پردازیم.

1. منظور از state در همه این متدها یک مکان در نقشه است که به صورت یک tuple دوتایی نمایش داده می‌شود که به ترتیب شماره ستون از چپ و شماره سطر از پایین است. (که در الگوریتم ها همان نودهای گراف هستند.)
2. در این پروژه action ها در کلاس Directions تعریف شده اند: Directions.NORTH و Directions.SOUTH و Directions.WEST و Directions.EAST و Directions.stop. (که در الگوریتم ها همان یال‌های گراف هستند.)
3. منظور از cost هزینه یک action است که در حالتی که روح نداریم هزینه همه action ها یک در نظر گرفته می‌شود. (که در الگوریتم ها همان وزن یال‌های گراف هستند.)

متدهای استفاده شده عبارتند از:

★ متد getStartState: مکان اولیه pacman در نقشه(state اولیه) را برمی‌گرداند که با توجه به نقشه داده شده برابر (15, 1) است.

★ متد isGoalState: این متد یک ورودی state دارد که مشخص می‌کند آیا state داده شده همان state هدف است یا خیر که با توجه به نقشه هدف در (5, 15) قرار دارد بنابراین تنها وقتی True برمی‌گرداند که state برابر (5, 15) باشد.

★ متد getSuccessors: این متد یک ورودی state دارد و همه فرزندان مجاز state ورودی (مکان‌های اطراف مکان داده شده) را به صورت یک tuple سه تایی برمی‌گرداند که اولی مکان فرزند(state فرزند) در نقشه و دومی action لازم برای رسیدن به این فرزند از state ورودی و سومی هزینه رفتن به این فرزند از state ورودی است. برای مثال خروجی برای state = (1, 14) با توجه به نقشه که تنها به خانه بالایی و پایین می‌تواند برود برابر:

getSuccessors((1, 14)) = [((1, 15), Directions.NORTH, 1), ((1, 13), Directions.SOUTH, 1)]

در پیاده‌سازی الگوریتم‌ها از ساختمان داده‌هایی استفاده شده است که در فایل util.py قرار دارند:

1) **صف (Queue)**: در این ساختمان داده از یک لیست به نام list برای نگه‌داری داده‌ها استفاده می‌شود. در متد push که یک ورودی به نام item دارد item را در ابتدا یعنی ایندکس صفر list اضافه می‌کند. در متد pop که ورودی ندارد آخرین داده ذخیره شده در list را برمی‌گرداند. در متد isEmpty خالی بودن یا نبودن list برگردانده می‌شود.
2) **پشته (Stack)**: در این ساختمان داده از یک لیست به نام list برای نگه‌داری داده‌ها استفاده می‌شود. در متد push که یک ورودی به نام item دارد item را در انتها یعنی در یکی بیشتر از ایندکس آخر list اضافه می‌کند. در متد pop که ورودی ندارد آخرین داده ذخیره شده در list را برمی‌گرداند. در متد isEmpty خالی بودن یا نبودن list برگردانده می‌شود.
3) **صف الویت‌دار (PriorityQueue)**: در این ساختمان داده از متدهای کتابخانه heapq استفاده شده است تا پیچیدگی زمانی را کاهش دهد. این ساختمان داده شبیه به صف است با این تفاوت که در متد push داده در یک heap و در مکان مناسب با توجه به اولویت خود ذخیره می‌شود نه انتهای list و در متد pop ریشه ساختمان داده heap چون دارای کمترین اولویت است برداشته می‌شود نه ابتدای list. همچنین متد دیگری به نام update دارد، در صورتی که اولویت یک داده تغییر کند اولویت آن را تغییر داده و heap را طبق اولویت جدید دوباره مرتب می‌کند و در صورتی که این داده در heap وجود نداشت همانند متد push عمل می‌کند.

الگوریتم‌های جستجو:

- **الگوریتم breadth first search**: این الگوریتم از نود ریشه با عمق صفر شروع به جستجو می کند و سپس همه نود با عمق یک را جستجو می‌کند و سپس همه نودها در عمق دو را جستجو میکند و این کار را تا جایی

ادامه می‌دهد تا به نود هدف برسد. این الگوریتم درتابع breadthFistSearch پیاده شده است. برای پیاده‌سازی این الگوریتم از ساختمان داده Queue استفاده شده است و نام object آن queue گذاشته شده است که در ابتدا نود ریشه که همان مکان اولیه پک من است و از طریق متد problem.getStartState به دست می‌آید به queue از طریق متد push اضافه می‌شود. همچنین از یک لیست به نام explored برای نگه‌داری نودهایی تا این لحظه دیده شده‌اند، استفاده می‌شود تا نودهای تکراری بررسی نشوند و الگوریتم در حلقه بی‌نهایت نیفتد. همچنین از آنجایی که ما به مسیری که از طریق آن به نود هدف رسیده‌ایم نیاز داریم از یک dictionary به نام came_from استفاده می‌کنیم که در آن هر نود یافت شده به یک tuple دوتایی شامل پدر نود و action لازم برای رسیدن به نود map می‌شود. برای مثال اگر به کمک action با Directions.SOUTH از نود (15, 2) به (14, 2) برسیم:

came_from = {(2, 14): ((2, 15), Directions.SOUTH)}

مراحل زیر را انجام می کنیم.

1. اگر queue خالی نباشد (queue.isEmpty مقدارش False) یعنی تا زمانی که همه نودها را ندیده باشیم به مرحله 2 می‌رویم در غیر این صورت به مرحله 7 می‌رویم.
2. قدیمی‌ترین نودی که به queue اضافه شده است را به کمک queue.pop در متغیر current می‌ریزیم تا همواره نودهای با عمق کمتر زودتر بررسی شوند. به مرحله 3 می‌رویم.
3. به کمک problem.isGoalState چک می‌کنیم که current همان نود هدف است یا خیر. اگر current همان نود هدف باشد به مرحله 6 می‌رویم در غیر این صورت به مرحله 4 می‌رویم.
4. اگر current در explored موجود باشد یعنی قبلا بررسی شده باشد به مرحله 1 می‌رویم در غیر این صورت ابتدا آن را به explored اضافه می‌کنیم سپس به مرحله 5 می‌رویم.
5. روی لیستی از فرزندان current به همراه action مورد نیاز برای رسیدن به هر یک آن‌ها و هزینه این action که توسط problem.getSuccessors به دست می‌آید حرکت می‌کنیم و به ازای هر فرزند چک می‌کنیم که اگر در explored وجود دارد دوباره بررسی نشود و به فرزند بعدی برویم اما اگر در explored وجود نداشته باشد آن را به queue به کمک متد queue.push اضافه می‌کنیم و این نود را به current به عنوان پدرش و action مورد نیاز برای رسیدن به آن map کرده و در came_from ذخیره می‌کنیم تا اگر در مسیر رسیدن به هدف بود بتوانیم مسیر رسیدن به آن را بازیابی کنیم. به مرحله 1 می‌رویم.
6. در این مرحله مسیر رسیدن به هدف یافت شده است تنها باید آن را بازیابی کنیم که این کار را به کمک تابع reconstruct_path انجام می‌دهیم. این تابع came_from و نود current که مسیر رسیدن به آن را می‌خواهیم را می‌گیرد و لیستی از action های مورد نیاز برای رسیدن به current را برمی‌گرداند. در came_from هر نود به پدرش و action مورد نیازش map شده است. بنابراین مراحل زیر را انجام می‌دهیم و الگوریتم با برگرداندن مقداری که تابع reconstruct_path برمی‌گرداند به پایان می‌رسد.
   a. یک لیست به نام total_actions برای نگه‌داری action های مورد نیاز برای رسیدن به current می‌سازیم و همچنین یک لیست به نام path برای نگه‌داری نودهایی که در مسیر رسیدن به current وجود دارند می‌سازیم.
   b. اگر current در came_from وجود داشته باشد یعنی برای آن پدر و action ذخیره شده باشد یا به عبارتی دیگر به نود آغازین نرسیده باشیم به مرحله b می‌رویم و در غیر این صورت به مرحله d می‌رویم.
   c. پدر current را به جای خود متغیر current می‌گذاریم و action آن را به total_actions و current را به path اضافه می‌کنیم. (در واقع در حال ذخیره action ها و نودهای در مسیر را به صورت وارونه هستیم.) به مرحله b می‌رویم.
   d. چون action های مورد نیاز برای رسیدن به هدف و نودهای موجود در مسیر رسیدن به هدف به صورت وارونه به ترتیب در total_actions و path ذخیره شده‌اند معکوس آن‌ها را چاپ می‌کنیم و وارونه total_action را برمی‌گردانیم.
7. مسیری برای رسیدن به هدف وجود ندارد.

- **الگوریتم depth first search**: این الگوریتم از نود ریشه با عمق صفر شروع به جستجو می کند و سپس همه نودهای یک شاخه را تا اخرین عمق ممکن جستجو می‌کند و سپس از شاخه باز می‌گردد و همه نودها در تا آخرین عمق ممکن در شاخه بعدی جستجو می‌کند و این کار را تا جایی ادامه می‌دهد تا به نود هدف برسد. این الگوریتم درتابع depthFistSearch پیاده شده است. برای پیاده‌سازی این الگوریتم از ساختمان داده Stack استفاده

شده است و نام object آن stack گذاشته شده است که در ابتدا نود ریشه که همان مکان اولیه پک من است و از طریق متد problem.getStartState به دست می‌آید به stack از طریق متد push اضافه می‌شود. همچنین از یک لیست به نام explored برای نگه‌داری نودهایی تا این لحظه دیده شده‌اند، استفاده می‌شود تا نودهای تکراری بررسی نشوند و الگوریتم در حلقه بی‌نهایت نیفتد. همچنین از آنجایی که ما به مسیری که از طریق آن به نود هدف رسیده‌ایم نیاز داریم از یک dictionary به نام came_from استفاده می‌کنیم که در آن هر نود یافت شده به یک tuple دوتایی شامل پدر نود و action لازم برای رسیدن به نود map می‌شود. برای مثال اگر به کمک action با Directions.SOUTH از نود (2 ,15) به (2 ,14) برسیم:

came_from = {(2, 14): ((2, 15), Directions.SOUTH)}

مراحل زیر را انجام می کنیم.

1. اگر stack خالی نباشد (stack.isEmpty مقدارش False) یعنی تا زمانی که همه نودها را ندیده باشیم به مرحله 2 می‌رویم در غیر این صورت به مرحله 7 می‌رویم.
2. جدیدترین نودی که به stack اضافه شده است را به کمک stack.pop در متغیر current می‌ریزیم تا همواره نودهای با عمق بیشتر زودتر بررسی شوند. به مرحله 3 می‌رویم.
3. به کمک problem.isGoalState چک می‌کنیم که current همان نود هدف است یا خیر. اگر current همان نود هدف باشد به مرحله 6 می‌رویم در غیر این صورت به مرحله 4 می‌رویم.
4. اگر current در explored موجود باشد یعنی قبلا بررسی شده باشد به مرحله 1 می‌رویم در غیر این صورت ابتدا آن را به explored اضافه می‌کنیم سپس به مرحله 5 می‌رویم.
5. روی لیستی از فرزندان current به همراه action مورد نیاز برای رسیدن به هر یک آن‌ها و هزینه این action که توسط problem.getSuccessors به دست می‌آید حرکت می‌کنیم و به ازای هر فرزند چک می‌کنیم که اگر در explored وجود دارد دوباره بررسی نشود و به فرزند بعدی برویم اما اگر در explored وجود نداشته باشد آن را به stack به کمک متد stack.push اضافه می‌کنیم و این نود را به current به عنوان پدرش و action مورد نیاز برای رسیدن به آن map کرده و در came_from ذخیره می‌کنیم تا اگر در مسیر رسیدن به هدف بود بتوانیم مسیر رسیدن به آن را بازیابی کنیم. به مرحله 1 می‌رویم.
6. در این مرحله مسیر رسیدن به هدف یافت شده است تنها باید آن را بازیابی کنیم که این کار را به کمک تابع reconstruct_path انجام می‌دهیم که روش کار آن بالاتر توضیح داده شده است و الگوریتم با برگرداندن مقداری که تابع reconstruct_path برمی‌گرداند به پایان می‌رسد.
7. مسیری برای رسیدن به هدف وجود ندارد.

- **الگوریتم uniform cost search**: این الگوریتم از نود ریشه شروع به جستجو می کند و سپس هر بار از میان نودهایی تا کنون پدرشان بررسی شده است نودی که برای رسیدن به آن از ریشه هزینه کمتری باید بدهیم به عنوان نود بعدی انتخاب می‌شود تا بررسی شده و فرزندانش به لیست نودهایی که در مرحله بعد باید از میان آن‌ها انتخاب کنیم اضافه می‌شوند یا اگر فرزندی در لیست وجود دارد در صورتی که مسیر جدید یافت شده هزینه کمتری نسبت به قبلی داشته باشد مقدار هزینه آن آپدیت می‌شود. این کار را انقدر ادامه می‌دهیم تا نودی که انتخاب می‌شود نود هدف باشد. این الگوریتم درتابع uniformCostSearch پیاده شده است. برای پیاده‌سازی این الگوریتم از ساختمان داده PriorityQueue استفاده شده است و نام object آن frontier گذاشته شده است که در آن اولویت همان هزینه رسیدن به نود مدنظر از نود ریشه است. که در ابتدا نود ریشه که همان مکان اولیه پک من است و از طریق متد problem.getStartState به دست می‌آید به همراه اولویت(هزینه) 0 به frontier از طریق متد push اضافه می‌شود. همچنین در یک dictionary به نام g_score هزینه رسیدن از ریشه به هر نود تا این لحظه را نگه می‌داریم تا مطمئن شویم به مسیری با کمترین هزینه می‌رسیم و در ابتدا مقدار g_score نود ریشه(problem.getStartState) را برابر 0 قرار می‌دهیم. همچنین از آنجایی که ما به مسیری که از طریق آن به نود هدف رسیده‌ایم نیاز داریم از یک dictionary به نام came_from استفاده می‌کنیم که در آن هر نود یافت شده به یک tuple دوتایی شامل پدر نود و action لازم برای رسیدن به نود map می‌شود. برای مثال اگر به کمک action با Directions.SOUTH از نود (2 ,15) به (2 ,14) برسیم:

  came_from = {(2, 14): ((2, 15), Directions.SOUTH)}

  مراحل زیر را انجام می دهیم.

1. اگر frontier خالی نباشد (frontier.isEmpty مقدارش False باشد.) یعنی تا زمانی که همه نودها را ندیده باشیم به مرحله 2 می‌رویم در غیر این صورت به مرحله 6 می‌رویم.
2. از میان frontier نودی که هزینه رسیدن به آن از همه کمتر است را به کمک متد frontier.pop برداشته و در متغیر current ذخیره می‌کنیم. به مرحله 3 می‌رویم.
3. به کمک problem.isGoalState چک می‌کنیم که current همان نود هدف است یا خیر. اگر current همان نود هدف باشد به مرحله 5 می‌رویم در غیر این صورت به مرحله 4 می‌رویم.
4. روی لیستی از فرزندان current به همراه action مورد نیاز برای رسیدن به هر یک آن‌ها و هزینه این action که توسط problem.getSuccessors به دست می‌آید حرکت می‌کنیم و به ازای هر فرزند هزینه رسیدن به آن را از طریق current محاسبه می‌کنیم که برابر مجموع هزینه رسیدن به current و هزینه رسیدن از current به فرزند است(g_score[current]+cost) و در متغیر temp ذخیره می‌کنیم. سپس چک می‌کنیم که اگر برای فرزند در g_score مقداری وجود ندارد(به معنای آن که این اولین مسیر یافت شده برای فرزند است.) یا اگر وجود دارد مقدار آن از temp بیشتر است(به معنای آن که مسیری با هزینه کمتر برای فرزند یافت شده است.) current وaction که از طریق آن از current به فرزند می‌رسیم را در came_from ذخیره می‌کنیم و مقدار g_score فرزند را به مقدار temp آپدیت می‌کنیم. همچنین نیاز است تا اولویت فرزند در frontier را به مقدار temp آپدیت کنیم. پس از انجام این کار روی همه فرزندان به مرحله 1 می‌رویم.
5. در این مرحله مسیر رسیدن به هدف یافت شده است تنها باید آن را بازیابی کنیم که این کار را به کمک تابع reconstruct_path انجام می‌دهیم که روش کار آن بالاتر توضیح داده شده است.
6. مسیری برای رسیدن به هدف وجود ندارد.

- **الگوریتم A***: این الگوریتم از نود ریشه شروع به جستجو می کند و سپس هر بار از میان نودهایی تا کنون پدرشان بررسی شده است نودی که مجموع هزینه رسیدن از نود ریشه به آن با هزینه تخمین زده شده برای رسیدن از آن نود به نود هدف کمتر باشد را به عنوان نود بعدی انتخاب می‌کنیم تا بررسی شده و فرزندانش به لیست نودهایی که در مرحله بعد باید از میان آن‌ها انتخاب کنیم اضافه می‌شوند یا اگر فرزندی در لیست وجود دارد در صورتی که مسیر جدید یافت شده هزینه تخمینی کمتری نسبت به قبلی داشته باشد مقدار هزینه آن آپدیت می‌شود. این کار را انقدر ادامه می‌دهیم تا نودی که انتخاب می‌شود نود هدف باشد. برای این الگوریتم به تابع heuristic نیاز است تا هزینه از هر نود را به نود هدف تخمین بزند. در این پیاده‌سازی ما از manhattan distance نود مدنظر تا نود هدف برای این تخمین استفاده کردیم. پیاده‌سازی manhattan distance در تابع manhattanHeuristic آمده است که در فایل searchAgents.py قرار دارد. این تابع دو ورودی اصلی می‌گیرد یکی نود مدنظر و دیگری problem را تا از طریق آن نود هدف را بیابد و عددی که برمی‌گرداند در واقع مجموع قدرمطلق فاصله سطری و ستونی نود مدنظر با نود هدف است. در واقع اگر مانعی سر راه یک من نباشد با توجه به آن که یک من تنها افقی یا عمودی حرکت می‌کند حداقل با این تعداد حرکت می‌تواند به هدف برسد بنابراین مقداری که این تابع تخمین می‌زند همواره کوچک‌تر مساوی هزینه واقعی است، در نتیجه manhattan distance قابل قبول است و به کمک آن می‌توان مطمئن بود که الگوریتم A* همواره بهترین مسیر با کمترین هزینه را می‌یابد. این الگوریتم در تابع aStartSearch پیاده شده است که علاوه بر problem ورودی دیگری دارد که آن تابع heuristic است. در نتیجه اگر بخواهیم از تابع heuristic دیگری استفاده کنیم تنها کافی است این ورودی را تغییر دهیم. اما ما در این پیاده سازی از همان manhattan distance استفاده می‌کنیم. برای پیاده‌سازی این الگوریتم از ساختمان داده PriorityQueue استفاده شده است و نام object آن frontier گذاشته شده است که در آن اولویت همان هزینه تخمینی رسیدن به نود هدف از طریق نود مدنظر است. که در ابتدا نود ریشه که همان مکان اولیه پک من است و از طریق متد problem.getStartState به دست می‌آید به همراه اولویت(هزینه) 0 به frontier از طریق متد push اضافه می‌شود. همچنین در یک dictionary به نام g_score هزینه رسیدن از ریشه به هر نود تا این لحظه را نگه می‌داریم تا مطمئن شویم به مسیری با کمترین هزینه می‌رسیم و در ابتدا مقدار g_score نود ریشه(problem.getStartState) را برابر 0 قرار می‌دهیم. همچنین از آنجایی که ما به مسیری که از طریق آن به نود هدف رسیده‌ایم نیاز داریم از یک dictionary به نام came_from استفاده می‌کنیم که در آن هر نود یافت شده به یک tuple دوتایی شامل پدر نود و action لازم برای رسیدن به نود map می‌شود. برای مثال اگر به کمک action با Directions.SOUTH از نود (15, 2) به (14, 2) برسیم:

  came_from = {(2, 14): ((2, 15), Directions.SOUTH)}

  مراحل زیر را انجام می دهیم.

1. اگر frontier خالی نباشد (frontier.isEmpty مقدارش False باشد.) یعنی تا زمانی که همه نودها را ندیده باشیم به مرحله 2 می‌رویم در غیر این صورت به مرحله 6 می‌رویم.
2. از میان frontier نودی که هزینه تخمینی رسیدن به هدف از طریق آن از همه کمتر است را به کمک متد frontier.pop برداشته و در متغیر current ذخیره می‌کنیم. به مرحله 3 می‌رویم.
3. به کمک problem.isGoalState چک می‌کنیم که current همان نود هدف است یا خیر. اگر current همان نود هدف باشد به مرحله 5 می‌رویم در غیر این صورت به مرحله 4 می‌رویم.
4. روی لیستی از فرزندان current به همراه action مورد نیاز برای رسیدن به هر یک آن‌ها و هزینه این action که توسط problem.getSuccessors به دست می‌آید حرکت می‌کنیم و به ازای هر فرزند هزینه رسیدن به آن را از طریق current محاسبه می‌کنیم که برابر مجموع هزینه رسیدن به current و هزینه رسیدن از current به فرزند است(g_score[current]+cost) و در متغیر temp ذخیره می‌کنیم. سپس چک می‌کنیم که اگر برای فرزند در g_score مقداری وجود ندارد(به معنای آن که این اولین مسیر یافت شده برای فرزند است.) یا اگر وجود دارد مقدار آن از temp بیشتر است(به معنای آن که مسیری با هزینه کمتر برای فرزند یافت شده است.) current وaction که از طریق آن از current به فرزند می‌رسیم را در came_from ذخیره می‌کنیم و مقدار g_score فرزند را به مقدار temp آپدیت می‌کنیم. همچنین نیاز است تا اولویت فرزند در frontier را به مقدار temp به علاوه هزینه تخمینی رسیدن به هدف از فرزند ((g_score[current]+heuristic(child))آپدیت کنیم. پس از انجام این کار روی همه فرزندان به مرحله 1 می‌رویم.
5. در این مرحله مسیر رسیدن به هدف یافت شده است تنها باید آن را بازیابی کنیم که این کار را به کمک تابع reconstruct_path انجام می‌دهیم که روش کار آن بالاتر توضیح داده شده است.
6. مسیری برای رسیدن به هدف وجود ندارد.

روح‌ها

برای این بخش از نقشه maze_with_ghosts.lay استفاده می‌شود که تنها تفاوتش اضافه شدن دو روح است. برای نمایش روح ها در نقشه از نماد 'G' استفاده می‌شود. این روح‌ها به صورت رندوم از میان خانه‌های مجاز اطرافشان خانه‌ای را برای حرکت انتخاب می‌کنند. برای آن که یک من مسیر خود را با توجه به روح‌ها انتخاب کند هزینه هر action(وزن هر یال گراف) را به جای آنکه همیشه یک باشد برابر میانگین manhattan distance روح ها با نود که به آن می‌رویم قرار می‌دهیم برای پیاده‌سازی این مورد در متد __init__ در کلاس PositionSearchProblem در فایل searchAgents.py چک می‌کنیم که اگر تعداد روح‌ها در نقشه غیر از صفر بود(یا به عبارتی روح وجود داشت) تابع costFn که هزینه هرنود را مشخص می‌کند را به جای آنکه همواره برابر یک قرار دهد برابر میانگین manhattan distance روح ها با آن نود قرار دهد. برای محاسبه میانگین manhattan distance روح ها تا یک نود خاص از تابع manhattanDistance در فایل searchAgents.py استفاده می‌شود. از میان الگوریتم‌های بالا dfs و bfs تفاوتی در مسیر پیشنهادی آن‌ها ایجاد نمی‌شود زیرا این دو به وزن یک یال توجهی ندارند اما در الگوریتم‌های ucs و a* با توجه به آن که وزن یال در آن‌ها تاثیر می‌گذارد مسیر با توجه به مکان روح‌ها ممکن است تغییر کند.

برای اجرای برنامه می توانید به دستورات موجود در فایل commands.txt مراجعه کنید.
برای قسمت الف پروژه نودهایی که در مسیر انتخاب شده وجود دارند به شکل یک لیست در کنسول چاپ می‌شوند و همچنین action های مورد نیاز برای رسیدن به هدف در یک لیست چاپ می‌شوند. همچنین اطلاعاتی نظیر هزینه رسیدن به مسیر و تعداد نودهای بررسی شده(expanded) چاپ می‌شوند.
برای قسمت به می‌توانید به صورت گرافیکی حرکت پک من به سمت هدف را ببینید همچنین شما برخی خانه‌ها را رنگ شده مشاهده می‌کنید که در واقع نودهای بررسی شده(expanded) در حین اجرا الگوریتم برای یافتن بهترین مسیر هستند که هرچه این نودها در ابتدا اجرای برنامه بررسی شده باشند قرمزتر و هرچه دیرتر بررسی شده باشند تیره تر هستند.

نتیجه

می‌توان مشاهده کرد که در این نقشه dfs با بررسی 48 نود به جواب رسید اما در یک نقشه دیگر با چند مسیر برای رسیدن به هدف dfs هرچند سریع است اما نمی‌تواند تضمین کند که مسیر بهینه را پیشنهاد می‌دهد. bfs با بررسی 116 نود به هدف رسید. این الگوریتم کند است اما اگر وزن همه یال‌ها یکی باشد (همان طور که در حالت بدون روح هزینه برای همه یال‌ها یک بود.) تضمین می‌کند که ما را به جواب بهینه می‌رساند. ucs نیز با بررسی 116 نود به هدف رسید، این الگوریتم نیز به نسبت کند است اما حتی اگر وزن یال‌ها متفاوت باشد(مثل حالت با روح) می‌تواند تضمین کند که مسیر بهینه را می‌یابد. a* با بررسی

100 نود به هدف رسید سرعت این الگوریتم به تابع heuristic آن بستگی دارد که در اینجا با تابع manhattanHeuristic به نسبت بهتر است. همچنین این‌که الگوریتم ما را به جواب بهینه می‌رساند نیز به تابع heuristic آن بستگی دارد، اگر این تابع قابل قبول باشد ما به جواب بهینه می‌رسیم و از آنجایی که manhattanHeuristic قابل قبول است ما به جواب بهینه می‌رسیم. بنابراین از میان الگوریتم‌های بررسی شده a* ما را به سرعت به نسبت خوب به همراه تضمین آنکه به جواب بهینه می‌رسیم، می‌رساند و از بقیه بهتر است.