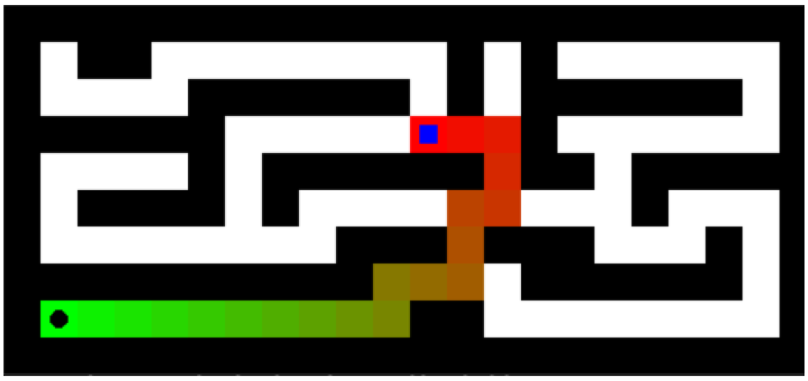
CS440/ECE448 Fall 2021

Assignment 1: Search



Due Wednesday, September 8th, 11:59pm.

Introduction

In this assignment, we will implement general-purpose search algorithms and use them to solve 'pacman'-like maze puzzles. This assignment has five parts.

1. Breadth-first search, with one waypoint.
2. A* search, with one waypoint.
3. A* search, with many waypoints.
4. Faster A* search, with many waypoints.

Throughout this assignment, the goal will be to find a path from a given **starting position** in a maze which passes through a given set of **waypoints** elsewhere in the maze. We will begin by finding a path from the starting position to a single destination waypoint. Then we will generalize the implementation to handle multiple waypoints. Finally, we will explore heuristics to handle large numbers of waypoints in a reasonable amount of time.

Environment

This assignment is written in **Python 3**. If you have never used Python before, a good place to start is the Python tutorial. We recommend using Python version 3.8 or later.

This assignment contains visualization tools which depend on **pygame**. You can install pygame locally using the **pip3** tool.

Getting Started

To get started on this assignment, download the template code. The template contains the following files and directories:

- `main.py`
- `maze.py`
- `search.py`
- `grade.py`

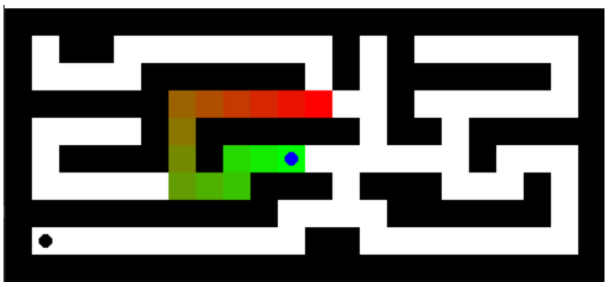
- key_s
- data/
    - data/part-1/
    - data/part-2/
    - data/part-3/

The data/ directory contains the maze files we will be using in this assignment. Each maze is a simple plaintext file, so you can easily make additional test mazes. Take a moment to open up one file and look at it.

The main.py file will be the primary entry point for this assignment. Let's start by running it as follows:

```
python3 main.py --human data/part-1/small
```

This will open a pygame-based interactive visualization of the data/part-1/small maze.



The blue dot represents the **agent**. You can move the agent, using the arrow keys, to trace out a **path**, shown in color.

> **note:** if the red-green gradient is hard for you to see, you can make the visualization use an alternative color scheme by specifying the --altcolor option.

The black dots represent the maze waypoints. Observe that this maze contains a single waypoint, in the lower left-hand corner. When you implement the search algorithms for this assignment, you will need to compute a path that takes the agent through all of the waypoints in the maze.

We also supply a script grade.py which you can use to test your code locally. The full Gradescope test suite is similar.

Let's see what happens when we run this script:

```
./grade.py

running in student mode (instructor key unavailable)
running in student mode (instructor key unavailable)
{'tests': ({'max_score': 2,
            'name': "part-1: `validate_path(_:)` for 'tiny' maze",
            'output': 'Your path validity is 0',
            'score': 0,
            'visibility': 'visible'},
           {'max_score': 1,
            'name': "part-1: not too many states explored for 'tiny' maze",
            'output': 'You explored 0 states, you should explore fewer than '
                      '1.1 * 15',
            'score': 1,
            'visibility': 'visible'},
           {'max_score': 2,
            'name': "part-1: correct path length for 'tiny' maze",
            'output': 'Your path length is 0, the correct length is 9',
            'score': 0,
            'visibility': 'visible'},
...
```

As you can see, we have a score of 0, since we have not implemented any of the functions in `search.py`. We will do this in the next few sections. Note: please do **not** change `key_s.grade.py` uses it and the local testing will break if you change it.

The Maze API

It is helpful to understand some of the infrastructure we have provided in the starter code. All the important functions and variables you should use can be found in `maze.py`. The `Maze` class is the abstraction that converts the input ASCII mazes into something whose properties you can access straightforwardly.

You can inspect the ASCII maze cells programatically using the `__getitem__(_:_:)` subscript.

```
cell = maze[row, column]
```

> **warning:** this subscript uses matrix notation, meaning the first index is the row, not the column. Spatially, this means the y-coordinate comes before the x-coordinate.

In the rest of this guide, we will use i to refer to a row index, and j to refer to a column index.

The maze size is given by the `size` member. The `size.x` value specifies the number of columns in the maze, and the `size.y` value specifies the number of rows in the maze.

```
rows    = maze.size.y
columns = maze.size.x
```

Keep in mind that the coordinate order in `size` is reversed with respect to the two-dimensional indexing scheme!

Each cell in the maze is represented by a single character of type `str`. There are four kinds of cells, which should be self-explanatory:

- **wall cells**
- **start cells**
- **waypoint cells**
- **empty cells**

For obvious reasons, a maze will only ever contain one starting position, but it can contain an arbitrary number of waypoint cells. However, for Part 1, you can assume there is only one waypoint cell in the maze.

You can determine what kind of cell a particular maze cell is by using the **maze legend**, available in the `legend` property.

```
# `True` if the cell is a wall cell
cell == maze.legend.wall

# `True` if the cell is the starting position
cell == maze.legend.start
```

```
# `True` if the cell contains a waypoint
cell == maze.legend.waypoint
```

You can assume that any cell that is not a wall, start position, or waypoint is empty.

The `Maze` type also supports the following interfaces, which may or may not be useful to you:

- `start : (int, int)`

  Gives the (i, j) coordinate of the starting position.

- `waypoints : [(int, int)]`

  A tuple containing a sequence of (i, j) coordinates specifying the positions of all the waypoints in the maze.

- `indices() : () -> [(int, int)]`

  Returns a generator traversing the coordinates of all the cells in the maze, in row-major order.

- `navigable(_:_:) : (int, int) -> bool`

  Takes (i, j) coordinates (as separate arguments), and returns a `bool` indicating if the corresponding cell is navigable (meaning the agent can move into it). All cells except for wall cells are navigable.

- `neighbors(_:_:) : (int, int) -> [(int, int)]`

  Takes (i, j) coordinates (as separate arguments), and returns a tuple containing a sequence of the coordinates of all the navigable neighbors of the given cell. A cell can have at most 4 neighbors.

- `states_explored : int`

  Keeps track of the number of cells visited in this maze. **Each call to** `neighbors(_:_:)` **increments this value by 1.** We will use this value to test if you are expanding the correct number of states, so do not call `neighbors(_:_:)` any more than necessary.

- `validate_path(_:) : ([(int, int)]) -> str?`

  Validates a path through the maze. This method returns `None` if the path is valid, and an error message of type `str` otherwise.

You will become very familiar with the `Maze` class as you implement the parts in the next sections. Remember, do **not** modify `maze.py`.

Part 1: Breadth First Search

For Part 1 of this assignment, we will implement breadth-first search (BFS) for a single waypoint. Specifically, we will implement a function `bfs(_:)` in `search.py` with the following signature:

```
# search.py
def bfs(maze):
```

Your `bfs(_:)` implementation should return a maze path, which should be a sequence of (i, j) coordinates. The first vertex of the path should be `start`, and the last vertex should be `waypoints[0]`.

> **hint:** the `deque` type, available in the `collections` module, may be useful.

You can view your generated path, and some interesting statistics about it, by running `main.py` as follows:

```
python3 main.py data/part-1/small --search bfs
```

You can test the other mazes by replacing the specified maze file `small` (in `data/part-1/`) with one of `tiny`, `small`, `no_obs`, or `open`.

```
python3 main.py data/part-1/tiny --search bfs
python3 main.py data/part-1/small --search bfs
python3 main.py data/part-1/no_obs --search bfs
python3 main.py data/part-1/open --search bfs
```

Note that there are additional hidden tests on the autograder on gradescope, but if you pass all tests for all parts locally you will get at least 55% of the possible credit.

Improving your BFS implementation

For the first couple parts of this MP, it is ok to use an (x,y) position to represent the state of your agent. However, later parts of the assignment will require passing through multiple waypoints. To solve those, your state must include information about which waypoints you have reached (or, alternatively, which waypoints you have not yet reached). It will be easier to write your code if you browse those sections now and design your code with the full state representation in mind.

Since the goal is close to the starting point, your code should explore only a small number of states before it finds the waypoint and returns. Take a look at how many states your code explore - the 'no_obs' maze is a good one to check you are not exploring too many.

Part 2: A*

For Part 2 of this assignment, solve a similar set of mazes as in the previous part, but this time using the A* search algorithm instead.

Write your implementation as a function `astar_single(_:)` in `search.py` with the following signature:

```
def astar_single(maze):
```

> **hint:** the `heapq` module may be useful.

Since all the test mazes contain only a single waypoint, you can use the **Manhattan Distance** from the agent's current position to the singular waypoint as the A* **heuristic function**. For two grid coordinates `a` and `b`, the manhattan distance is given by:

```
abs(a.i - b.i) + abs(a.j - b.j)
```

You can test your implementation by running `main.py` as follows, replacing `data/part-2/small` as needed:

```
python3 main.py data/part-2/small --search astar_single
```

You should find that the path lengths returned by A* are the same as those computed by breadth-first search, but A* explores fewer states.

Part 3: A* for Multiple Waypoints

Now, consider the more general and harder problem of finding the shortest path through a maze while hitting multiple waypoints. As suggested in Part 1, your state representation, goal test, and transition model should already be adapted to deal with this scenario. The next challenge is to solve different mazes using A* search with an appropriate admissible heuristic that takes into account the multiple waypoints.

There is a beautiful heuristic that is will not only find the optimal path through all the waypoints, but is admissable as well. It is based on a data structure you hopefully have seen before: the Minimum Spanning Tree (MST). Instead of computing the distances to each waypoint from the current position, it would be more helpful to obtain an estimate of the cost of reaching all the rest of the unreached waypoints once we have reached one. Obtaining this estimate can be done with an MST.

Specifically, we construct a graph where the vertices are the waypoints and each edge connecting `w_i` to `w_j` has weight `manhattan_distance(w_i, w_j)` for all pairs of vertices `(w_i, w_j)`, the MST represents the approximate lowest cost path that connects all the waypoints. Since it strictly underestimates the cost of going through all the waypoints, this is an admissable heuristic.

To help you, we have included a function `MST(_:)` in `search.py` that will generate an MST for you. It is up to you to use this to obtain the information you want, depending on how many waypoints remain.

So, suppose you are in the middle of a search. You're at some location (x,y) with a set of S dots still to reach. Your heuristic function h should be the sum of the distance from (x,y) to the nearest dot D, plus the MST length for the dots in S. The MST length is an underestimate of how long it will take to travel through all the waypoints once you have reached D.

Now, suppose that you're far from the remaining waypoints and you move the agent. The distance to the nearest waypoint D will change. However, the set of remaining waypoints is the same. So you'll end up asking for the same MST value again.

Search paths from a completely different part of your A* queue might also need the MST for the same set of waypoints. So it's a good idea to cache (memoize) MST values once you have computed them.

Write your implementation as a function `astar_multiple(_:)` in `search.py` with the following signature:

```
def astar_multiple(maze):
```

You can test your implementation by running `main.py` as follows, replacing `data/part-3/corner` as needed:

```
python3 main.py data/part-3/corner --search astar_multiple
```

Some of the mazes are special cases intended to test edge cases. Find a shortest path through these mazes by hand. Is your code finding a valid path (e.g. going through all the waypoints)? Is its path as short as the one you found by hand? If not, look back through the previous instructions to find the design bug.

One common problem is to have an inadequate representation of the agent's state, e.g. just the agent's (x,y) coordinates but not which waypoints it has/hasn't traversed.

Another common design bug is to use a "greedy" algorithm rather than the MST heuristic described above. Specifically, the greedy algoirthm calls A to find the nearest waypoint, then goes to that waypoint and calls A again. One of the above examples will produce an overly long path if you use this approach.

Part 4: Fast Heuristic Search for Many Waypoints

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. Write a suboptimal search algorithm that will do a good job on large complex mazes.
Your algorithm could either be A* with a non-admissible heuristic, or something different altogether (e.g. the greedy algorithm described above).

Specifically, your function must solve our hidden test maze in less than 6 minutes. Assuming that it finishes in that amount of time, grading will be based on the length of your returned path and its validity.

Why would a non-admissible heuristic be faster? Think of your heuristic as a bias for your algorithm. If it's biased towards taking the route that currently seems most promising, then it may find some solution faster but miss the best solution. One standard way to create a non-admissible heuristic is to multiply your original admissible heuristic h(x) by w, where w is a weight greater than 1. This is called "weighted A* search".

We're not supplying a test maze for this part. However, you can test your algorithm locally by seeing if it can solve the Part 3 mazes faster than your Part 3 code. Also, you can also make some mazes of your own, with more edges and/or waypoints than the Part 3 mazes. Aim for mazes that are too complex for your Part 3 code to solve in a reasonable amount of time.

Write your implementation as a function `fast(_:)` in `search.py` with the following signature:

```
def fast(maze):
```

You can test your implementation by running `main.py` as follows:

```
python3 main.py data/part-3/corner --search fast
```

Submitting to Gradescope

Submit this assignment by uploading `search.py` to Gradescope. You can upload other files with it, but only `search.py` will be retained by the autograder.

We strongly encourage you to submit to Gradescope early and often as you will be able to see your final score there. The local `grade.py` is purely for your convenience and does not include the full suite of tests you will be graded on. Moreover, if something should go wrong (e.g. you get sick), we'll be able to see a record of your earlier partial solutions.

Policies

You are expected to be familiar with the general policies on the course syllabus (e.g. academic integrity) and on the top-level MP page (e.g. code style). In particular, notice that this is an individual assignment.