CS2121 - Project 2 - 2019

Priority Queues and Search: Finding the Optimal Solution of a Maze

Out: March 20th, 2019

In: April 9th, 2019 at 11:55 pm via OWL

Note Extra Well:

Read over the contents carefully as there are some minor nuanced differences. You will also have quite a bit of freedom on how you go about doing the project. All is good as long as you hit the ideas (A*) and everything is correct.

Note Well:

You must take note of the following:

- **Before you start** you will need to install a Python package called PyGame. To do this, simply open up your terminal/cmd and type: `pip install pygame`. If this does not work because of permissions issues, instead type `pip install --user pygame` to install PyGame in your user directory. If you use anaconda at the command line, you should use `conda install pygame` instead.
- **If you are a 9643B student**, you have some additional requirements that are noted below and marked with "**(for 9643B only)**". The continuation of instructions for everyone will then be indicated.
- You are given four (4) .py files, two of which you leave unchanged and one you alter. You are also given a set of maze text files and an images directory that must remain unchanged. If you happen to accidentally change any of them, you can always re-download a fresh copy and replace the changed files/directory.
- **Make sure** to give yourself **ample time** to work on this project. You are given three weeks to work on this because it will take you time to develop different pieces of the project to get them working and then put it all together. **Do not** expect that you can pick this up a few days before the deadline and get it all working in time.
- Some of the descriptions are deliberately left somewhat vague. This gives you practice interpreting design instructions, but things will become clear as you get further on with your implementation. Part of the project is to figure out what the requirements mean and then plan your implementation to meet them.
- If there is something you are not sure how to do, you should first Google it or look it up in the Python documentation online: `https://docs.python.org/3/index.html`

Purpose

The purpose of this project is to implement a maze search using the A* (A-star) heuristic algorithm.

Summary

You will be writing a program to find the optimal solution to a maze using A*, which will require you to implement a priority queue data structure. Note that **your algorithm *is* expected to find the optimal solution to the maze**.

Provided:

You are provided with four python files, two of which that are already complete (`App.py`, `Maze.py`) and two that you must complete (`Astar.py` and `PriorityQueue.py`). You are provided with eight (8) text files (`m1.txt` through `m8.txt`) containing the raw maze data for eight mazes. And you are provided with an `images` folder containing images needed for the maze display to work correctly. Do not alter the location or content of this folder or the maze files.

Prior Work:

Feel free to use any of the code you developed for Project 1, but this should be your work, not copied from someone else.

`Maze` class

Nothing to do here. This class is complete and is just needed to help with visualization of the maze.

`App` class

Nothing to do here either. This class is also complete and needed to help with maze visualization. This class is not documented, but you are free to investigate the code if you like to try to figure out what it does. This is a good challenge, but not required.

Note that the `App` class is set up so that for an `App` object `a`, you can call `a.pause_for_key()` to force the app window to wait for the user to press a key to continue, and so that you can call `a.on_cleanup()` to allow the window close either by pressing any key or by clicking on the x button.

`PriorityQueue` class

You need to implement your own priority queue for the algorithm to operate. You can do this however you like, using whichever underlying data structure (list, linked list, min heap for a challenge, etc.) you prefer. Recall that the rules are:

- can enqueue things with a priority;
- dequeue will be done on the order of priority.[1]

Keep in mind that for our case, as in the cases we saw in lecture, *smaller value* of priority means you will be dequeued *earlier*.

`Astar` script

This is the code that will actually be used to solve the maze. You will need to complete this script so that it works to solve the maze automatically. Below are a few methods that I am recommending that you implement in order to make solving the problem easier.

Remember that breaking up a problem into manageable subtasks is an effective strategy for making problem solving simpler, and that doing this with code helps to write clearer, more reliable, and more maintainable code. Choosing what functions/methods to implement, thereby packaging certain code into a function/method, is a part of this general strategy.

Here are the functions I recommend you write:

1. `isLocValid(y, x)`
    - This method will take a coordinate in the maze and will tell us if that coordinate is within the bounds of the maze (`True`) or not (`False`).
    - For example, if we have a 10×10 maze, we can check if location $(5, 7)$ is valid (it is). If we check $(10, 7)$ or $(-1, 3)$, it would return `False`.
2. `getNextNeighbour(y, x)`
    - All places in the maze have a potential of 4 neighbours (up, right, down, left).
    - **Always** check neighbours in a clockwise direction starting with up (up, right, down, left).
    - This method will return the coordinates of the next valid and open neighbor cell.

[1] In class we discussed implementations such that FIFO is maintained for items with equal priority, which is fine to do here. However, if the equal priority items are handled using LIFO (like a stack), then your algorithm will behave like a depth-first search (which you did in Project 1), which avoids exploring more than one equally optimal solution. You are free to implement your priority queue in this way, but it is not required and you will not get any extra marks for doing so.

- Positions that are not valid, i.e., walls and already visited cells (past or presently in our path) should not be returned; only open positions and the end will ever be returned.
- If no such neighbour exists, return `None`.
- **Hint**: `isLocValid` might be handy here.

3. `loadMaze(filename)`
   - Just load the data from the maze text file and return a `numpy` array of the maze.
   - If you have never used the `numpy` package, look it up online. You only need to figure out how to construct a 2D array consisting of characters, which are Python `str` objects. (Note that you should not use the deprecated `numpy.chararray` function to do this).
4. `heuristic(start, end)`
   - This is given to you. It is what enables us to know the right direction to go.

Instructions

Our maze search will use the heuristic algorithm A* to find the optimal path through a maze. So you are actually going to use some AI to intelligently find your way through the maze! Simply, we will search multiple paths in parallel looking for the shortest path until one (or more) of the paths reaches the end. Along each path we consider we count how many steps we are from the start, and use a heuristic to estimate how far we are from the goal. At each step of the algorithm, we consider the shortest path taken so far (based on the heuristic) and then consider how we do by moving one step from our current position in one of the possible directions we can move. It is the fact that we can measure exactly the distance between a point in the maze and the exit that guarantees that we find an optimal path through the maze.

In a bit more detail, you will be assigning a priority to each position x (which is actually a tuple) along a path that you consider based upon the **number of steps** you have taken to get to x, this number is called $g(x)$, and a heuristic **distance measure** of how far x is from the end of the maze, this distance measure is called $h(x)$. The **priority** we assign to each position is then $f(x) = g(x) + h(x)$. For more details on the A* algorithm, you are highly encouraged to take a look through the A* wikipedia page.

The function `heuristic` provides the straight-line distance from the current position to the end, which is what you will use to evaluate $h(x)$ from above.

(for 9643B only): You will have to implement the A* search strategy (described fully below) as well as another strategy that is a variant of it. See below for details on what you need to do in addition to what everyone is required to do.

(for everyone) The maze is just a grid (2D array of characters) of positions. Each position in this grid can either be a wall (black) or an open space (light grey) (including the start (blue) and end (red)). The search cannot go through walls and should only ever travel through open positions. We want to mark where in the maze our search currently is (yellow). Additionally, we want to make sure we're keeping track of the path we've taken in our maze. The positions in the maze that are in our current path will be blue (now those positions still in the priority queue) and the positions that we've looked at, but are no longer active, will be dark grey. The status/type of position in the cell can be represented by a character in our 2D character array representing the maze (e.g., we change the value in the character array to `'c'` when it is the current position we are in). Note that you don't have to track an actual path taken, but you do need to track what is in the priority queue, what is currently being examined, and what has been discarded from the queue.

A list of position statuses (character values in the 2D character array) is listed below along with the character representing position status. **Note that there are some changes here from Project 1**:

- `'p'`: **things in the priority queue** (blue circle)
- `'c'`: the current position (yellow circle)
- `'m'`: empty cell (white cell – ignore this possibility)
- `'E'`: end/exit of the maze (red)
- `'f'`: end/exit is found (yellow circle on red)
- `'x'`: **already examined (not current and out of queue)** (dark grey circle)
- `' '`: open cell (light grey)
- `'S'`: start of the maze (blue – You might never really see this pop up once the search starts)
- `'#'`: wall cell (black)

For our purposes, treat each position as a tuple representing the coordinate of the position in the array/grid. We can then use this tuple to easily access the contents in maze array/grid. By the way you can actually index a 2D `numpy` array with a tuple, so that could come in handy, but be warned that the order of the tuple matters: (x, y) vs. (y, x). When you read in the maze files, they will just be a text file containing spaces, `#`, `S`, and `E`.

Some tips/hints that are likely to help you:

- Do not just enqueue the positions. Rather, you should enqueue tuples containing the position, the number of steps taken to get to that position, and the priority (steps taken + heuristic).
  - `((x, y), numSteps, priority)`: for example, for the start position you might have `((2, 6), 1, 11.9)`
- You can implement the priority queue however you like. You can do a linear search on enqueue or a linear search on dequeue.

- If you know that the tuple always has the priority in the last position, then just use that for your enqueue and dequeue operations.

The following is pseudocode for the procedure you should use for solving the maze:

1. Open maze file and store the character data as a 2D `numpy` array.
2. Create an `App` object with the maze.
3. The maze should now be set up.
4. Find the start of the maze. This position will be where the `'S'` is in the maze. **You must** treat positions in the maze as tuples. So, for example, (x, y) coordinates – or possibly (y, x) if that works better for you.
5. Create a priority queue object.
6. Add the start position to the queue (plus the number of steps, i.e. 1, and the priority, i.e., $f(x)$).
7. `while` the queue is not empty, and we have not yet found the end, do the following:
   (a) Dequeue.
   (b) Is it the end?
      - If so, print out the number of steps to the end.
      - **Do not** worry about actually showing the path.
   (c) Otherwise, check all of the nearest neighbours and add them to the priority queue based on $f(x)$.
   (d) Render the maze and update the positions accordingly (not necessarily in that order).

   - **Note**: each time through the loop, at some point, you're going to want to tell the `App` to re-draw itself. To do this, call the `App`'s `on_render()` method. If you do not do this, your maze will not look like it's doing anything.
   - **Note**: You **must** add a delay to your program. Without this, things will happen *way* too fast. Check out `time.sleep(someNumOfSeconds)` (there's an example in the `DFS` file you got for Project 1). A good place to call this is right next to the render call.
8. In the end, print out the total number of steps in the path to the end. Remember that your path does not need to be updated at this point.

(for 9643B only): Once you have this all working, add a third argument `epsilon` to the `heuristic` function that can be a non-negative float. Set up your `heuristic` function so that when `epsilon` is 0 your code works as before, but when `epsilon` is greater than zero, the heuristic is instead computed by an ε-admissible heuristic method of your choosing, such as static or dynamic weighting (see the Bounded Relaxation section of the A* wikipedia page).[2]. Note that the version of `heuristic` I gave you provides an *admissible heuristic*, which is

[2]Note that be careful if you use static weighting, as the equation given on wikipedia assumes $\varepsilon > 1$, whereas you are to assume only $\varepsilon > 0$, meaning that you need to redefine ε so that the equation is valid for $\varepsilon > 0$. The dynamic weighting equation does not have this defect.

what ensures that we obtain an optimal path, and that the length of the path that your ε-admissible heuristic provides is at worst $(1 + \varepsilon)$ times the length of the optimal solution path.

Your ε-admissible heuristic should provide a more efficient computation of a solution (smaller time-complexity) potentially at the cost of a sub-optimal path (more steps in the maze solution). Your code should provide a comparison of the running time (measured by the number of steps the algorithm takes to finish) and the number of steps taken in the shortest path found for each of the mazes `m5.txt` through `m10.txt`. Do this comparison by producing one plot for each maze using `matplotlib.pyplot`.

For each plot, you need to run the maze solver for a range of exponentially increasing values of ε between 0.1 and 100. To do this you can use the code:

```
epsilons = np.logspace(-1,2,n,10)
np.insert(a,0,0)
```

The first line produces `n` evenly spaced values (on a log scale) between $0.1 = 10^{-1}$ and $100 = 10^{2}$. The second line inserts `0` at the beginning of the array, which makes sure that you include the case $\varepsilon = 0$. A good choice for `n` is `25`, but you can increase this if you want to make a smoother plot. If your code is efficient you should be able to increase `n` to well over `100`, or possibly even `1000` or more, and still produce the plots in a reasonable amount of time.

Note well: For these tests *make sure to* **set the delay to 0** *and* **turn off creation of the App object**, and any method calls to it, to turn off any display functionality. This will **significantly** speed up your code. Also, make sure to keep track of both the number of steps in the shortest path found along with the number of steps that the algorithm takes to finish (this is the number of times you dequeue). This is the data you will plot for each maze: one curve for the steps the algorithm takes; and a second curve for the steps in the path found.

You may find the following code useful for making clear plots with two differently scaled time series:

```
plt.clf()
fig, ax1 = plt.subplots()

color = 'tab:blue'
ax1.semilogx(epsilons, totalSteps, color=color)
ax1.set_xlabel('epsilon')
ax1.set_ylabel('algorithm steps', color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()

color = 'tab:orange'
ax2.plot(epsilons, pathSteps, color=color)
```

```
ax2.set_ylabel('path steps', color=color)
ax2.tick_params(axis='y', labelcolor=color)

plt.title('Algorithm Efficiency versus Solution Size for Maze '+filename[1:])

plt.savefig(filename+'.pdf')
```

Finally, **make sure** to comment in your code (`Astar.py`) on the following things:

- Point out any notable or interesting features in each of your in your plots;
- Discuss what you can say about the best choice of ε given your experiments? Explain your reason for choosing an optimal value of ε.

(for everyone): Your program will also have to run with command line arguments. Think of these as like parameters in a method, but rather that being for a method, we have parameters for the whole program. The command line arguments you will pass to the program will be the file name of the maze we want to solve and the delay. This should work in the same way as for Project 1.

(for 9643B only): In addition to the maze filename and delay, your code should take a third optional parameter, a non-negative *float* indicating the value of ε to be passed to the `heuristic` function.

Cool ideas for a challenge (not for extra marks) (for everyone)

If you feel like your display should show an actual path through the maze, go ahead and show one. If there is more than one path just show one.

Also, your implementation only checks up, right, down and left as possible directions to move. If you want to be slick, you can allow diagonals, and see the difference it makes.

Submitting the Assignment (for everyone)

You will submit the project via OWL, under the assignments tab. **For CS2121 and DH2221 only submit your complete python files as text files** `Astar.txt` **and** `PriorityQueue.txt`, **For CS9643 submit your two complete python files as text and your six plots as .pdf files** via OWL. Please follow the instructions carefully and do what is asked, but not more, to ensure that you do not unnecessarily lose marks. Make sure not to modify the function names.