Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit your code on Canvas as a zip file (the entire "starter" folder) and the answers to the questions of the assignment as a pdf. The pdf must be submitted as a separate file so we can more easily visualize it on Canvas for marking.

Overview

In this assignment, you will implement Dijkstra's algorithm and A* to solve pathfinding problems on video game maps. We will consider a grid environment where each action in the four cardinal directions (north, south, east, and west) has a cost of 1.0, and each action in one of the four diagonal directions has a cost of 1.5. A video game map, a start location, and a goal location define each search problem. The assignment package available on Canvas includes many maps from movingai.com, but you will use a single map in our experiments; feel free to explore other maps if you like.

The assignment package already implements most of the code you need. In the next section, we detail some of the key functions you will use from the starter code. You can reimplement all these functions if you prefer; their use isn't mandatory. However, the assignment must be implemented in Python.

Heap Tutorial (0 Marks)

Run the file heap_tutorial.ipynb on Jupyter Notebook (see instructions on how to install Jupyter Notebook here: https://jupyter.org/install). The Notebook file is a tutorial about Python's heapq library, as you will need it to implement the OPEN lists in the assignment. Note that we assume a minimum knowledge of Python to complete the assignment. For example, we assume that you are familiar with dictionaries and lists in Python. If you are not familiar with the language and its basic structures, please seek help during office hours and labs. You will also need to be familiar with Python for the other course assignments. If you are not familiar with the language, you should see this course as a good learning opportunity.

Starter Code (0 Marks)

The starter code comes with a class implementing the map and another implementing the nodes in the tree. We also provide the code for running the experiments (see main.py for details about the experiments).

State Implementation

The State class (see algorithms.py) implements the nodes in the search tree. It contains the following information: x and y coordinates of the state in the map, the g- and cost-values of the node, and a parent pointer. We also include the width (W) of the map. This is because we use the map width to compute the following hash function for a state with coordinates x and y: $y \times W + x$ (see method state_hash of State). This is a perfect hash function, i.e., each state is mapped to a single hash value. We will leave it as an exercise for you to understand this hash function. Note that you can use the function without understanding it.

The "less than" operator for State is already implemented to account for the attribute cost of the nodes. Please see the heap tutorial to understand why the "less than" operator needs to be implemented. The use of this cost attribute in the class allows us to easily implement both Dijkstra's algorithm and A* with the same State class. That is, if you store the f-value of a node in cost of class State, then the heap will be automatically sorted according to the f-values; if you store the g-value of a node in cost, then the heap will be sorted according to the g-values, and so on. It is thus your responsibility to decide which information is added to cost, depending on the algorithm you are implementing.

The g-value of the states is already computed by the Map implementation, as explained below. As you implement Dijkstra's algorithm and A*, you will have to set the cost and the parent of the nodes in search.

Map Implementation

Most of the functions in the map implementation are called internally or in main.py, so you will not have to worry about them. In main.py we create an instance of the map used in the experiments as follows: gridded_map = Map("dao-map/brc000d.map"). This instance must be passed to your search algorithms, so they can access the transition function of the state space defined by the map.

The most important method you will need to use from map.py is successors. This method receives a state s as input and returns a set of states, the children of s. The children of s are returned with their correct g-values (see State Implementation above for details). For example, children = gridded_map.successor(start) generates all children of start and stores them in a list called children. One can then iterate through the children as one does with any list in Python: for child in children.

The Map class also offers a method called plot_map for plotting the map and the states in CLOSED after completing a search. This method can be helpful to visualize the search and possibly help you find bugs. For example, the image below shows the map and states in the CLOSED list of A* (left) and CLOSED list of Dijkstra's algorithm (right) for the same search problem. The white areas are traversable regions while black areas represent walls. The gray areas represent the states generated in the search. If you zoom in you will be able to see a pixel with a lighter color in the gray region; this circle represents the initial state.

```
map.plot_map(CLOSED, start, goal, 'name_file')
```

In this example, map is the map object, CLOSED is the CLOSED list (of either A* or Dijkstra's algorithm after the search is completed), and name_file is the name of the file in which the image will be saved.

Bringing Map and State Together

We consider 30 test instances from the file testinstances.txt for the brc000d map. The test instances (start and goal states) are read in main.py. All you need to do is to pass the start and goal states as well as the map instance to your search algorithms (see the lines starting with "Replace None, None, None..." in main.py for where you need to insert the calls to your implementation of Dijkstra's algorithm and A*).

Here is a code excerpt that assumes the existence of a state called start and a map called map (see the Map Implementation above) and it creates a dictionary whose keys are given by the hash function.

```
CLOSED = {}
CLOSED[start.state_hash()] = start
children = gridded_map.successors(start)
    for child in children:
        hash_value = child.state_hash()
        if hash_value not in CLOSED:
            CLOSED[hash_value] = child
```

How to Run Starter Code

Follow the steps below to run the starter code (instructions are for Mac and Linux).

- Install Python 3.
- It is usually a good idea to create a virtual environment to install the libraries needed for the assignment. The virtual environment step is optional.
  - virtualenv -p python3 venv
  - source venv/bin/activate
  - When you are done working with the virtual environment you can deactivate it by typing deactivate.

- Run pip install -r requirements.txt to install the libraries specified in requirements.txt.

You are now ready to run the starter code by typing: python3 main.py.

If everything goes as expected, you should see several messages as shown below. These messages are the result of running a set of test cases. Naturally, if you haven't implemented the search algorithms, then all test cases will return with a "mismatch." You will not see any of these mismatch messages once you have correctly implemented what is being asked.

```
There is a mismatch in the solution cost found by Dijkstra
and what was expected for the problem:

Start state: [108, 26]
Goal state: [105, 67]
Solution cost encountered: None
Solution cost expected: 42.5
Is the path correct? True

There is a mismatch in the solution cost found by A*
and what was expected for the problem:

Start state: [108, 26]
Goal state: [105, 67]
Solution cost encountered: None
Solution cost expected: 42.5
Is the path correct? True
```

The messages explain the problem instances for which your implementation failed to find an optimal solution. It also verifies whether the solution path returned is correct in the sense that it connects the initial state to the goal state. Note that the verification of correctness for the solution paths only works for problems with a solution path (some of the problems we consider do not have a solution). That is why the messages above state "True" for path correctness, despite the path being simply "None".

Implement Dijkstra's Algorithm (5 Marks)

Implement Dijkstra's algorithm and call your implementation in the line marked with the comment "replace None, None, None with the call to your Dijkstra's implementation" in main.py. The implementation must be correct, i.e., it must find an optimal solution for the search problems. The algorithm must return the solution path, the solution cost, and the number of nodes it expands to find a solution. If the problem has no solution, it must return "None" for the path and -1 for the cost. The solution path should be a list starting at the initial state and including all states on the path to the goal (inclusive).

The implementation must be efficient, i.e., it should use the correct data structures. You can test the correctness of your implementation of Dijkstra's algorithm by running python3 main.py. You may also use the plotting function of the Map class to visualize the result of your search.

You can implement the algorithm as a function or as a class, whichever is more convenient for you. Your implementation can be in a new file, in main.py, or in any other file you prefer.

Implement A* (4 Marks)

Implement A* and call your implementation in the line marked with the comment "replace None, None, None with the call to your A* implementation" in main.py. We will use the Octile distance with our implementation of A*. The octile distance is a version of the Manhattan distance function we have seen in class that accounts for diagonal moves. Intuitively, if we consider a map free of obstacles, the agent will perform as many diagonal moves as possible because a diagonal move allows one to progress in both the x and y coordinates toward the goal. Let Δx and Δy be the absolute differences in the distance in the x-axis and the y-axis, respectively, between the evaluated state and the goal state. The maximum number of diagonal moves we can perform is given by $\min(\Delta x, \Delta y)$ and each move costs 1.5; the values that cannot be corrected with diagonal moves are corrected with regular cardinal moves, where each move costs 1.0, and there are $|\Delta x - \Delta y|$ of them. Octile distance can be written as

$$h(s) = 1.5 \min(\Delta x, \Delta y) + |\Delta x - \Delta y|,$$

The Octile distance is consistent and thus admissible. Since the heuristic is consistent, you do not have to implement the re-opening and re-expansion of nodes we discussed in class.

You can implement the algorithm as a function or as a class, whichever is more convenient for you. Your implementation can be in a new file, in main.py, or in any other file you prefer. Similarly to the A* implementation, you can implement the Octile distance anywhere you prefer in the code.

Answering Questions (4 Marks)

Your implementation of Dijkstra's algorithm and A* are finished, look at the two scatter plots your code generates: nodes_expanded.png and running_time.png. Each point in the scatter plot represents a search problem, and one of the axes represents Dijkstra's algorithm and the other A*. One of the plots compares the number of nodes expanded in search, while the other compares the running time in seconds of the two algorithms.

Note that different implementations of A* might result in a slightly different number of expansions. For example, one correct way to implement A* reinserts into OPEN copies of a state s if a cheaper path to s is found. Depending on the specific implementation, the more expensive copies of s might also be counted as expansions as they exit OPEN. These differences could result in two correct implementations expanding a slightly different number of nodes during the search. We will not penalize your assignment as long as the number of states expanded is approximately comparable to what we expect.

1. (2 Marks) Regarding the two scatter plots, answer the following questions. You should include the scatter plots your code generated in your answer.
   a) (1 Mark) Why is the overall distribution of points in the plot the way it is? That is, what is the overall story the distribution of points tells?
   b) (1 Mark) Explain the difference in the distribution of points between the expansions plot and the running time plot. In particular, what causes the shift in distribution?
2. (1 Mark) What happens when you multiply the heuristic value for A* by 2? Explain what you observe. It can be helpful to plot a scatter plot of the number of expansions for A* as you had originally implemented (in one axis) and A* with the heuristic multiplied by 2 (in the other axis). Note that if you decide to generate this plot, you need to modify main.py.

3. (1 Mark) What happens when you change your Dijkstra's and A* implementations so that the search does not update the cost and parent pointer to a node *n* once it finds a better path to *n*. Please discuss what you observe in terms of solution cost and solution path.