University of Alberta
Department of Computing Science
CMPUT 366 - Fall 2024

Assignment 1
Due date: September 27
8 marks

Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit your code on Canvas as a zip file (the entire "starter" folder).

Overview

In this assignment, you will implement Dijkstra's algorithm and A* for solving 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. Each search problem is defined by a video game map, a start location, and a goal location. The assignment package available on Canvas includes a large number of maps from `movingai.com`, but you will use a single map in our experiments; feel free to explore other maps if you like.

Most of the code you need is already implemented in the assignment package. 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. The assignment must, however, 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 aren't familiar with the language and its basic structures, please seek help during office hours and labs. You will need to be familiar with Python for the other course assignments as well. If you aren't 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. 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.

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.

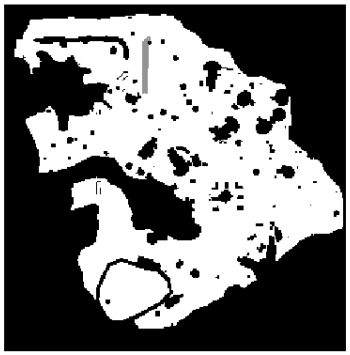
The most important method you will need to use from `map.py` is `successors`. This method receives a state s as input and returns a set of states, the children of s. The children of s are returned already with their correct g-values (see `State` Implementation above for details). For example, `children = 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`.

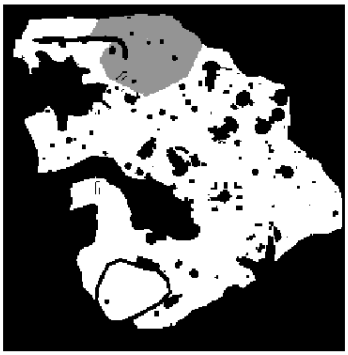
The `Map` class also offers a method called `plot_map` for plotting the map and the states in `CLOSED` after completing a search. This method can be helpful to visualize the search and possibly help you find bugs in your implementation; it is also great to gain intuition on what the search algorithms are doing. For example, the image below shows the map and states in the `CLOSED` list of A* (left) and the `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 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.

Consider the following instruction for plotting such a map.

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

Here, `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..." 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

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
```

Implement Dijkstra's Algorithm (4 Marks)

Implement Dijkstra's algorithm and call your implementation in the line marked with the comment "replace 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 cost and the number of nodes it expands to find a solution. When the problem has no solution, it must return -1 for the cost. There is no need to recover the optimal path the algorithm encounters, but only report the cost and number of expansions.

The implementation must be efficient, i.e., it should use the correct data structures. Note that your implementation can be correct in the sense that it finds the optimal solution to each test problem, but still be inefficient. It is your responsibility to ensure that your implementation uses the correct data structures, as discussed in class.

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 with the call to your A* implementation" in `main.py`. We will use the Octile distance with our implementation

of A*. Octile distance is a version of the Manhattan distance function we have seen in class that accounts for diagonal moves. Intuitively, if we are considering 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 at a cost that is cheaper than moving horizontally and then vertically. Let Δx and Δy be the absolute differences in distance in the x-axis and in 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. As a suggestion to gain intuition, draw a small grid on a piece of paper and manually compute the Octile distance between different states.

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.

Gaining Intution (0 Marks)

In this section, we suggest experiments you could run to gain intuition on how Dijkstra's algorithm and A* work. Feel free to play around with the code and run experiments other than the ones we suggest.

Once you have implemented both Dijkstra's algorithm and A*, run the code with the "plots" option enabled: `python3 main.py --plots`.[1] Your program will generate two scatter plots: `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 compare the number of nodes expanded in search, while the other compares the running time in seconds of the two algorithms. Study these plots and try to understand what the distributions of points in the scatter plots mean.

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.

In addition to the scatter plot, generate plots with the CLOSED lists of Dijkstra's algorithm and A* for different problems, as explained in section "Map Implementation". For example, how does the search look for problems with no solution?

As a final suggestion, what happens as you multiply the heuristic values by 2 in the A* search? What happens to the solution cost and the number of expansions of the resulting algorithm?

[1] Note that copy and paste might fail as the pdf might carry some hidden characters that will break your code. You should type "`--plots`" directly in the command line instead.