University of Alberta
Computing Science Department
CMPUT 366 - Fall 2022

Assignment 2
Due date: October 07
12 marks

Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit your code on eClass as a zip file 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 eClass for marking. Please submit the entire folder of the project so that it is easier to run your implementation.

Overview

In this assignment you will implement A*, Bidirectional A* (Bi-A*), and MM 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 the cost of 1.0 and each action in one of the four diagonal directions has the 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 eClass 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 be implemented in Python, however.

Heap Tutorial (0 Marks)

You can skip this section if you read the heap tutorial in Assignment 1.

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).

Starter Code Difference Between Assignments 1 and 2. The main difference between the starter code of Assignment 1 and the starter code of this assignment is the implementation of the `State` class. Since you will be implementing algorithms that require different cost functions, class `State` now has an attribute called `cost` and the less-than operation is overloaded to account for it. That way, if you store the f-value of a node in `cost`, then the heap will be 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.

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-value, and the cost value 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 cost of the nodes. It is up to you to decide which value to store in `cost` of `State`. Please see the heap tutorial to understand why the "less than" operator needs to be implemented.

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. For example, the image below shows the map and states in both CLOSED lists of MM. The lighter areas are traversable regions, while gray areas represent walls. The darker areas represent the states expanded in search. If you zoom in you will be able to see two pixels with a lighter color in the middle of the darker area; one represents the start state and the other the goal state. Here is an example of use of this function.

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

In this example, `map` is the map object, `CLOSED` is the union of MM's CLOSED lists, and `name_file` is the name of the file in which the image will be saved.

Bringing Map and State Together

We consider the same 30 test instances used in Assignment 1 for the `brc000d` map. The test instances (start and goal states) are read in the main file. 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 comments starting with "Replace None, None with a call..." in main.py for where you need to insert the calls to your implementation of A*, Bi-A*, and MM).

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 --testinstances`. Copy and paste might not work properly because `--testinstances` could be pasted as `-testinstances`.

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 A* and what was expected for the problem:
Start state:  [26, 123]
Goal state:  [59, 12]
Solution cost encountered:  None
Solution cost expected:  176.0

There is a mismatch in the solution cost found by MM and what was expected for the problem:
Start state:  [26, 123]
Goal state:  [59, 12]
Solution cost encountered:  None
Solution cost expected:  176.0

There is a mismatch in the solution cost found by Bi-A* and what was expected for the problem:
Start state:  [26, 123]
Goal state:  [59, 12]
Solution cost encountered:  None
Solution cost expected:  176.0
```

Implement A* (3 Marks)

In this assignment 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. The maximum

number of diagonal moves we can perform is given by $\min(\Delta x, \Delta y)$; the remaining values can be corrected by equation $|\Delta x - \Delta y|$. Here, Δx and Δy are the differences in distance in the x-axis and in the y-axis, respectively, between the evaluated state and the goal state. Octile distance can then be written as follows.

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

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

Implement Bi-A* (2 Marks)

Implement the Bi-A* algorithm. Bi-A* performs an A* search from both ends of the search problem: one starting from the initial state and another from the goal state. You will also use the Octile distance as the heuristic function to estimate the cost to go from start to goal and goal to start. In each iteration of Bi-A* you will expand the node with lowest f-value in either searches. Bi-A* stops once the cost of the best solution encountered is less-than or equal to the minimum f-value in either `OPEN` lists.

Implement MM (3 Marks)

Implement the MM algorithm with the Octile distance as the heuristic function to estimate the cost to go from start to goal and goal to start. In our implementation we will use a simpler stopping condition. Namely, MM will stop its search once the cost of the best solution it has encountered is less than or equal to

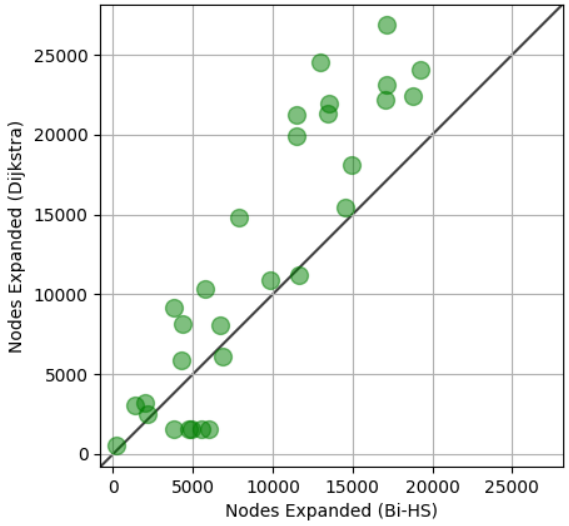
$$\min(p_{minf}, p_{minb}) \,.$$

Here, p_{minf} and p_{minb} are the smallest p-value in the forward and backward `OPEN` lists, respectively.

Your implementation must be correct, i.e., it must find an optimal solution for the search problems, for all algorithms (A*, Bi-A*, and MM). The algorithms must return the solution cost and the number of nodes it expands to find a solution. If the problem has no solution, they 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 of the algorithms must be efficient, i.e., it should use the correct data structures.

Analyzing the Scatter Plots (4 Marks)

Once you have implemented A*, Bi-A*, and MM, run the code with the "plots" option enabled: `python3 main.py --testinstances --plots`. Your program will generate two scatter plots: one comparing MM with A* and another comparing MM with Bi-A*. Each point in a scatter plot represents a search problem and each axis an algorithm. The plot below shows the scatter from Assignment 1, which compares the number of expansions Bi-BS and Dijkstra's algorithm perform on the same test instances used in this assignment.



Regarding the scatter plot shown above and the two scatter plots you code generated, answer the following questions. You do not need to justify your answer if the question does not ask for a justification. Please include the scatter plots your code generates to substantiate your answers.

1. (0.5 Mark) In general, does the use of a heuristic function increase or decrease the number of nodes the algorithms need to expand to find a solution?

2. (1.5 Mark) What do you expect to observe in terms of the running time of the algorithms that use a heuristic function in comparison to the algorithms that do not use a heuristic function? Select one of the following options and justify your answer.

   Hint: You should consider the differences in terms of computational cost of generating a set of children and adding them to the `OPEN` list for uninformed and informed algorithms.

   a) The running time will increase exactly in the same proportion in which the number of expansions increases. For example, if A* expands 27% more states than Dijkstra's algorithm, then A* will be exactly 27% slower than Dijkstra's algorithm.

   b) The running time will decrease exactly in the same proportion in which the number of expansions decreases. For example, if A* expands 27% fewer states than Dijkstra's algorithm, then A* will be exactly 27% faster than Dijkstra's algorithm.

   c) The running time will increase, but not as much as the number of expansions increases. For example, if A* expands 27% more states than Dijkstra's algorithm, then A* will be a bit less than 27% slower than Dijkstra's algorithm.

   d) The running time will decrease, but not as much as the number of expansions decreases. For example, if A* expands 27% fewer states than Dijkstra's algorithm, then A* will be a bit less than 27% faster than Dijkstra's algorithm.

3. (0.5 Mark) Does MM tend to perform more or fewer expansions than Bi-A*?

4. (0.5 Mark) Did the heuristic-guided bidirectional algorithms deliver their promise of substantially reducing the number of expansions one needs to perform to solve a problem?

5. (1.0 Mark) Speculate on an explanation for the distribution of points in the scatter plot that compares A* and MM. Since you are being asked to speculate, you will receive full marks in this question as long as your answer does not contain conceptual mistakes.