University of Alberta
Computing Science Department
CMPUT 366 - Winter 2023

Assignment 1
Due date: January 27
12 marks

Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit your code on eClass 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 eClass for marking.

Overview

In this assignment you will implement Dijkstra's algorithm and Bidirectional Search (Bi-BS) 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)

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 and the g-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 g-values of the nodes. 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 Bi-BS. The white areas are traversable regions while black areas represent walls. The gray areas represent the states expanded in search. If you zoom in you will be able to see a pixel with a lighter color in the middle of the two circles; 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 Bi-BS's CLOSED lists (or Dijkstra's algorithm CLOSED list), 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 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 lines starting with "Implement here..." in main.py for where you need to insert the calls to your implementation of Dijkstra's algorithm and Bi-BS).

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 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 Bi-BS
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 "Implement here 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. If 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. You can test the correctness of your implementation of Dijkstra's algorithm by running `python3 main.py --testinstances`. You may also use the plotting function of the `Map` class to visualize the result of your search.

Implement the Bi-BS Algorithm (4 Marks)

Implement the Bi-BS algorithm and call your implementation in the line marked with the comment "Implement here the call to your Bi-BS'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. If 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. You can test the correctness of your implementation by running `python3 main.py --testinstances`. You may also use the plotting function of the `Map` class to visualize the result of your search. Recall that you must take the union of the states in both CLOSED lists in order to use the plotting function of `Map`.

Analyzing the Scatter Plot (4 Marks)

Once you have implemented both Dijkstra's algorithm and Bi-BS, run the code with the "plots" option enabled: `python3 main.py --testinstances --plots`. Your program will generate one scatter plot: `nodes_expanded.png`. Each point in the scatter plot represents a search problem and one of the axis represents Dijkstra's algorithm and the other Bi-BS. Regarding the scatter plot, explain the following.

1. (0.5 Mark) Why is the overall distribution of points in the plot the way it is?
2. (1.5 Marks) Why some of the points are clearly **below** the main diagonal?
3. (2.0 Marks) Why some of the points are clearly **above** the main diagonal?

Hint: It might be helpful to plot the results of the searches (see the explanation of function `plot_map` from class `Map` in section "Map Implementation") to answer numbers 2 and 3 above.