Project 1: Graph Search

CS 6370 / ME EN 6225 Motion Planning
University of Utah

Due: 27 Sept 2015 by 11:55pm

Instructions

All solutions should be uploaded via the assignment on Canvas.

Naming: Your upload should be named in the format <*uid*>.zip where <*uid*> is your Utah uid.

Code Organization: You should have a text file called README in the root directory of your submitted archive which explains how to run the code submitted. Any output generated and returned should be clearly labeled using he convention Proj<number>-<question number>-<question sub>-counter. Output should be stored as text files or if generating an image be saved in a common format (i.e. JPG, PNG, TIFF).

Written Answers: Place all written answers to questions in a single document of a standard format (txt / pdf / doc / odt). This should be clearly named Proj<*number*>-answers.[txt/pdf], where number is the Project number (i.e. 1 for this assignment).

Forward Graph Search (125 total points)

The lectures so far have focused on performing search on grids and graph representations. This project will focus on implementing and analyzing some of the algorithms we've discussed.

To help you with getting started there is an included python file `graph_serach.py` which has some classes defined to help you focus on the algorithm implementation. There is an additional file named `PythonPointers.pdf` on Canvas which gives some links and simple examples to help people who are new to python. The code also contains function headers for the search methods you need to implement. Feel free to change the function arguments necessary to make it easier for you to solve the problems.

In addition to the code file included there are a few text files with included maps which we will use for evaluating our algorithms. If you write you own code you'll need to have it read this kind of file so that it can be evaluated on new maps once you've submitted it for grading. The map is defined as follows:

0 - denotes clear cell

x - denotes obstacle

g - denotes goal location (clear)

i - denotes init location (clear)

For now the robot has actions $A = \{\texttt{left}, \texttt{right}, \texttt{up}, \texttt{down}\}$. The robot can move to a free cell, but if the action attempts to move to an occupied cell or off the map the action is not valid and the robot remains in the same state.

1 Depth First Search (40 pts)

We will start by implementing depth first search. I'll walk you through this one.

Implement DFS You may find the Python `list` class helpful. For a list instance `l = []` you can push to the end of the list by performing `l.append(n)` and pop from the end using the function call `x = l.pop()`

The provided class `GridMap` reads in a map file and sets up a transition function for the grid world. It also provides methods for testing if the goal location has been reached. I've provided psuedocode for a version of depth first search below:

```
def DFS(init_state, f, is_goal, actions):
    frontier = [] # Search stack
    n0 = SearchNode(init_state, actions)
    visited = []
    frontier.push(n0)
    loop_count = 0
    while len(frontier) > 0:
        # Peak last element
        n_i = frontier.pop()
        if n_i not in visited:
            visited.add(n_i.state)
            if is_goal(n_i.state):
                return (path(n_i), visited)
            else:
                for a in actions:
                    s_prime = f(n_i.state, a)
                    n_prime = SearchNode(s_prime, actions, n_i, a)
                    frontier.push(n_prime)
    return None
```

I should be able to run your code using a map file as described above and it should be clear how to change the transition function and actions used.

1.1 (10 pts) Run DFS on `map0.txt`. Report the path and set of states visited. Is the path found the shortest path? Explain why the algorithm did or did not find the shortest path.

1.2 (5 pts) Perform the same test and analysis for `map1.txt`.

1.3 (5 pts) Perform the same test and analysis for `map2.txt`.

1.4 (4 pts) The DFS algorithm explained in the psuedocode above adds search nodes for all actions from a given node at once. What issues may arise from doing this? How could this be alleviated?

1.5 (6 pts) Reverse the order of the actions explored by DFS and run it again on `map1.txt` and `map2.txt`. Does anything change in your results? What? Why is this the case?

1.6 (10 pts) Extend DFS to iterative deepening DFS. You need to add an argument to your DFS function that it only explores to a maximum depth m during execution.

Run your iterative deepening implementation on `map1.txt` and `map2.txt`. Did your algorithm find the shortest path? Explain why it did or did not.

2 Breadth First Search (25 pts)

Implement BFS Now implement breadth first search. Remember we want to use a queue as our frontier data structure. In Python we can implement this by again using the `list` data structure. For a given list `l = []` we can push to the end of the queue using `l.append(n)` and pop from the front using `n = l.pop(0)`. Note the parameter 0 provided to pop, this tells the list to pop element 0 from the list (the front). You could pop the second element using `l.pop(1)` or explicitly from the rear of the list by using `l.pop(-1)`

2.1 (5 pts) Run BFS on `map0.txt`. Report the path and set of states visited. Is the path found the shortest path? Explain why the algorithm did or did not find the shortest path.

2.2 (5 pts) Perform the same test and analysis for `map1.txt`.

2.3 (5 pts) Perform the same test and analysis for `map2.txt`.

2.4 (10 pts) Compare the performance of your algorithm to DFS and iterative deepening DFS above. How do the paths found differ? How do the states visited differ? Which was easier to implement and why? Discuss anything else you found interesting or difficult in implementing and evaluating the three algorithms.

Searching with Costs

We will now switch to examining algorithms for planning with costs associated to actions. To begin with all actions will have the same cost of 1. We will change this below.

3 Uniform Cost Search (15 pts)

Implement Uniform Cost Search Remember that for implementing uniform cost search you need a priority queue to access the lowest cost path currently in the frontier. I have provided the python class `PriorityQ`. This function is setup to work with the `SearchNode` class and I have not used it with other classes. The class has the functions `push()`, `pop()`, and `peak()`, which respectively allow you to add an element with an associated cost to the queue, remove the lowest cost element, and see what the lowest cost element is. There is also a replace function for updating an element with the same state as the element added,

however you can also access this function by calling `push()` with a SearchNode that has a state already in the queue.

You can also get the length of a `PriorityQ` instance q using $len(q)$ or test if a state s is in the priority queue by using $sinq$ which will return True if the state s is currently in the queue.

3.1 (5 pts) Run uniform cost search on `map0.txt`. Report the path and set of states visited. Is the path found the lowest cost path? Is the path found the shortest path? Explain why the algorithm did or did not find the lowest cost path.

3.2 (5 pts) Perform the same test and analysis for `map1.txt`.

3.3 (5 pts) Perform the same test and analysis for `map2.txt`.

4 A* Search (40 pts)

Implement A* You now need to implement A* search by incorporating heuristics. You should make it easy to swap out heuristic functions as we will be evaluating a few below.

4.1 (10 pts) Implement a euclidean distance heuristic for evaluating A*. Run your algorithm on maps `map0.txt`, `map1.txt`, and `map2.txt` and give the results. How does the path you get compare to uniform cost search? How do the states explored compare to uniform cost search? Did you notice any other differences in comparing the two?

4.2 (10 pts) Now implement a Manhattan distance heuristic. Run it on the same three maps and perform the same analysis.

4.3 (20 pts) Now extend the set of actions to allow the robot to move diagonally on the grid. Make diagonal action cost 1.5 as opposed to 1 for lateral and vertical moves. Run the same tests and analysis from 4.1 and 4.2 using both heuristics. Are these heuristics still admissable for our new problem? Why or why not? What is the effect of the new actions being added compared to the previous problems? What is the effect of the heuristic being admissable or not on the solutions your implementation gives?

5 Self Analysis (5 pts)

5.1 (1 pt) What was the hardest part of the assignment for you?

5.2 (1 pt) What was the easiest part of the assignment for you?

5.3 (1 pt) What problem(s) helped further your understanding of the course material?

5.4 (1 pt) Did you feel any problems were tedious and not helpful to your understanding of the material?

5.5 (1 pt) What other feedback do you have about this homework?