CSC421: Artificial Intelligence
Assignment 1

Rafael Solorzano

May 23, 2018

1 Introduction

This assignment of solving a map problem using uniformed and informed search algorithms. The map is a 2D 100x100 grid with 26 cities labeled 'A' to 'Z'. Each city is randomly placed in a non-repeated place in the grid. Furthermore, for each city the closes 5 cities based on Euclidean distance is calculated from which between 1 and 4 of them are chosen to be connected randomly. Experimentation was run against a bi-directional cities in which a path is added from any missing if it is included in one, and one in which the path is only considered as it was randomly at first, this is further discussed in section 2. An implementation discussion follows in section with the entire code shown and detailed in section 4.

1.1 Formulation of Problem as Search Problem

The map search problem can be formally defined as follows:

- Initial State: any city from the 26 possible cities chosen randomly.
- Action: The only possible actions is moving from one a city to an immediately adjacent one, incurring its cost. Therefore the actions available at one particular state are the immediately adjacent cities to that state.
- Transition Model: The only transition possible is go to city from city. For example In(A) Go(B) will result going from city A to city B, note that is only possible if B is in the adjacent list of A.
- Goal State: any city from the 26 possible cities chosen randomly, Initial State equaling Goal State is allowed.
- Goal Test: Checks if the current state is the goal state.
- Path Cost: The path cost is the euclidean distance between the adjacent cities.
- Solution: The list of cities to travel to reach from initial state to goal state.

The problem is represented using two dictionaries. The first contains a mapping between each city and its X and Y coordinates in the 2D grid. The second dictionaries contain a mapping between each city and another dictionary containing its immediate neighbours with the respective cost incurred by transition to that specific city.

2 Experimental Results

Results obtained from experimentation with the problem generator, uniformed search algorithms and informed search algorithms are presented in this section.

2.1 Problem Generation

The map problem is generated using Python. First, the 26 random locations in the 100x100 2D grid are generated making sure none of them use the same spot in the grid. Second, The euclidean distance for city n and all other cities is calculated, nevertheless only the 5 closest cities for each city are kept in a distance list. Finally, a random number between 1 and 4 is generated, this number will decide how many paths a city will include. Furthermore, the list is shuffled and then between 1 and 4, depending on the random number, cities are removed from its paths. This is done for each city, therefore it can result that city A has a connection to city B but city B did not generate a connection to A during the last processing, therefore to achieve bi-directionality any city that has a connection missing, will have one added when the problem gets turned intro a graph before processing, this achieves an undirected graph. Note that during experimentation both options are tested, that is the tests are ran without forcing bi-directionality and with an undirected graph.

2.1.1 Branches Generated

Random map generation was ran 10, 100 and 1000 times to generate that number of random maps. Each time the average number of paths (branches) is recorded. The results are summarized in Table 1.

Iterations	Average Paths
10	63.7
100	65.29
1000	65.252

Table 1: Average number of Paths (branches) generated

The code that generated this averages can be seen in Figure 1. The description of generate_map function is presented in section 4.

```
#Map branches test for Question 2
paths_10 = []
for i in range(10):
    _,_,total_paths = generate_map(True)
    paths_10.append(total_paths)
print(paths_10)

paths_100 = []
for j in range(100):
    _,_,total_paths = generate_map(True)
    paths_100.append(total_paths)
print(paths_100)

paths_1000 = []
for k in range(1000):
    _,_,total_paths = generate_map(True)
    paths_1000.append(total_paths)
print(paths_1000)

print("Average Number of paths for 10 iterations ")
print(np.mean(paths_10))
print("Average Number of paths for 100 iterations ")
print(np.mean(paths_100))
print("Average Number of paths for 1000 iterations ")
print(np.mean(paths_1000))
```

Figure 1: Code Generation for Branches Averages.

2.2 Uninformed Search Strategies

The results obtained from running Breadth-First Search, Depth-First Search, and Iterative Deepening Search on 100 different instances of a map for each algorithm are presented below. Note, that as previously the tests are ran twice for each algorithm, one with forced bi-directionality and one without.

The result of calling any of the three algorithms mentioned above will be a list of cities to follow from the start state to the goal state. If no such path exists the result will be an empty list. An example of the printed results of both cases are shown in Figure 2 and 3.

```
Attempting to solve from W to T
Solution from BFS
['W', 'Z', 'M', 'U', 'T']
Solution from DFS
['W', 'O', 'Z', 'M', 'U', 'T']
Solution from IDS
['W', 'Z', 'M', 'U', 'T']
```

Figure 2: Result from calling Uninformed Search with a path.

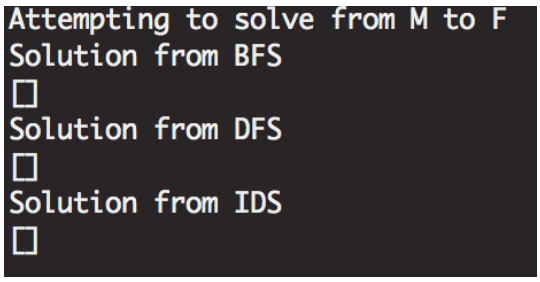

Figure 3: Result from calling Uninformed Search with no solution.

The results from from Figure 2 and 3 are generated by the code shown in Figure 4. The explanation of each function and class shown in the figure is described in Section 4.

```
# ---------- Sample result of one map for each search strategy Question 3------
#Generate cities
cities_locations, cities_distances = generate_map()
#Turn City intro Graph Object
random_map = Graph(cities_distances)
#Generate Random start and goal
start = randint(65,90)
goal = randint(65,90)
#Setup Problem
problem = MapProblem(chr(start),chr(goal),cities_distances,cities_locations)

print("Attempting to solve from " + chr(start) + " to " + chr(goal))

#Print Results for BFS
result_bfs = breadth_first_search(problem)
node, path_back = result_bfs, []
while node:
        path_back.append(node.state)
        node = node.parent
print("Solution from BFS")
print(list(reversed(path_back)))

#Print Results for DFS
result_dfs = depth_first_search(problem)
node, path_back = result_dfs, []
while node:
        path_back.append(node.state)
        node = node.parent
print("Solution from DFS")
print(list(reversed(path_back)))

#Print Results for IDS
result_ids = iterative_deepening_search(problem)
node, path_back = result_ids, []
while node:
        path_back.append(node.state)
        node = node.parent
print("Solution from IDS")
print(list(reversed(path_back)))

#-------------- End Question 3 ----------------------------------------
```

Figure 4: Code to generate the results of Uniformed Search.

The results obtained from running each search strategy including their average space complexity, average time complexity, actual running time (of the total of the 100 iterations), average path length, and numbers of problem solved are summarized in Table 2 and Table 3 below. Table 2 include the results from forced bi-directionality whereas Table 3 include the results from a non bidirectional map.

It was expected that the bidirectional will solve all problems, in some cases it will fail to solve at most 2 out of the 100 test. On the contrary, for the non-bidirectional search it was expected that not all maps will have solutions and as it can be seen from Table 3 only about 65 maps were solvable. Additionally, for the Iterative Deepening Search it took about 25 seconds to run all the 100 iterations, as it had to dig deeper each time it went through the iteration, the limit was set to 13 as anything above 15 will take a considerable amount of time to solve.

Search	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
BFS	201	4.85	3.55s	3	100
DFS	315	12.52	3.59s	10	100
IDS	924	3.18	3.57s	3	100

Table 2: Results from running BFS, DFS and IDS on bi-directional map.

Search	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
BFS	80	6.20	3.54s	4.03	63
DFS	106	8.18	3.60s	6.2	65
IDS	5541038	2.68	24.48s	4	67

Table 3: Results from running BFS, DFS and IDS on a non bi-directional map.

The description of each statistic and how it was calculated follows. The average space complexity was calculated simply as the max number of nodes generate between all 100 iterations of the problem. Furthermore, the average time complexity was calculated as the mean of all different number of nodes generated in the 100 problems. The running time was calculated using the Python time library. Finally, the average path length is the average of all the different length of solutions generated.

An example of how the data was analyzed for the BFS case is shown in Figure 5 below. All the search strategies follow an almost identical strategy.

```
#-------------Statistic Generation for BFS ---------------------
start_time_bfs = time.time()
nodes_generated = []
nodes_visited = []
path_lengths = []
problems_solved = 0
for i in range(100):
    #Generate cities
    cities_locations, cities_distances = generate_map()
    #Turn City intro Graph Object
    random_map = Graph(cities_distances,True)
    #Generate Random start and goal
    start = randint(65,90)
    goal = randint(65,90)
    #Setup Problem
    problem = MapProblem(chr(start),chr(goal),cities_distances,cities_locations)

    print("Attempting to solve from " + chr(start) + " to " + chr(goal))

    if breadth_first_search(problem) is not None:
        solution, explored, nodes_gen = breadth_first_search(problem,stats=True)
    else:
        continue

    node = solution
    solution_length = 0
    while node:
        solution_length+=1
        node = node.parent

    if solution_length != 0:
        problems_solved+=1
        path_lengths.append(solution_length)
    nodes_generated.append(nodes_gen)
    nodes_visited.append(len(explored))


print("Average Space Complexity (max nodes generated) %f" %np.max(nodes_generated))
print("Average Time Complexity (nodes visited) %f" %np.mean(nodes_visited))
print("Total BFS Execution Time %s" % (time.time() - start_time_bfs))
print("Average path length %f" %np.mean(path_lengths))
print("Total problems solved %f" %problems_solved)

#------------ End Static Generation for BFS ---------------------------------------
```

Figure 5: Code generation for BFS stats.

2.3 Informed Search Strategies

This section presents the results obtained from running greedy best-first search as well as A* search on 100 different instances of a map for each search strategy. As before, two different instances of each 100 maps is ran. Furthermore, the three heuristics used for the informed search strategies are detailed.

2.3.1 Search Heuristics

The search heuristics used for informed searching are:

- he: straight-line Euclidean distance to goal state
- h0: just the literal value '0'
- hm: Manhattan Distance to goal state.

The heuristics functions are acceptable as they each provide an estimated cost from the current state to the goal state based on different calculations. he provides a heuristic based on the straight line Euclidean-distance from the current state to the goal state which makes it admissible as it helps choose the best strategy to get to the goal with the fewest cost possible. Similarly, the hm which uses the Manhattan Distance is also admissible for the same reason as it will just provide an informed cost based on the sum of horizontal and vertical distances to the goal state. Finally, the h0 simply states that the informed cost to reach the goal is simply 0, therefore this heuristic is acceptable as it is nonnegative, it will satisfy the constraint that h(n) if n is a goal node = 0, and it is essentially just want to find a solution without regarding cost.

In terms of dominance two different results were obtained according to the search function being used. For Greedy best-first search the dominance is shown to be he ¿ hm ¿ h0 where he is the dominating one. On the contrary, for the A* search the dominance is shown to be hm ¿ he ¿ h0 where hm is the dominating one. The results of the experiments that led to this conclusion are shown in section 2.3.2 below.

2.3.2 Statistics

The results obtained from running each search strategy include the same statistics as with uniformed case. The results are summarized in Table 4 and 5 below. Table 4 include the results from forced bi-directionality whereas Table 5 include the results from non bidirectional map.

The results found where interesting, as it was shown that the dominance of a heuristic function changed depending on the search algorithm used. As A* incorporates the added path cost to its heuristic function its dominant heuristic was the hm which calculates the Manhattan Distance, on the other hand Greedy Best first search uses only the heuristic function, for which the most dominant was the he which calculates the euclidean distance. Finally, as it was expected the worst heuristic function h0, resulted in a large space and time complexity for both cases, this was expected mainly as it simply was not informing anything and just only cared about finding a solution.

Search Type	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
GBFS he	92	2.67	3.57s	3.42	100
GBFS h0	753	11.91	3.79s	5.01	100
GBFS hm	117	2.78	3.55s	3.41	100
A* he	82	3.73	3.69s	4.0	100
A* h0	503	12.64	3.62s	6.64	100
A* hm	68	2.77	3.48s	3.36	100

Table 4: Informed Search results on bi-directional map.

Search Type	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
GBFS he	46	3.93	3.66s	4.30	59
GBFS h0	106	8.68	3.49s	4.95	70
GBFS hm	71	3.68	3.63s	3.94	57
A* he	67	5.11	3.62s	4.73	63
A* h0	95	9.14	3.43s	5.10	64
A* hm	61	4.22	3.56s	4.21	75

Table 5: Informed Search results on non bi-directional map.

The statistics in the tables above were calculated in the same way as in the uniformed case.

3 Implementation Overview

Implementation Overview to go here.

4 Code Listing

Code explanation to go here.