College of Engineering and Applied Science
Department of Electrical Engineering and Computer Science

20CS 4033 & 6033 AI – I
Fall 2023
Instructor: Anca Ralescu

Homework Assignment #2
Assigned on September 25 2023
Due on October 5, 2023
11:59PM on Canvas
50 points

1.1 Correctness

The following sections discuss the correctness of each algorithm implementation, specifically, the ability for each algorithm to find a path or return empty if no path exists.

1.1.1 Breadth Search

In breadth-first search (BFS), all nodes at each depth level are explored prior to the algorithm moving deeper into the search space. This is comparable to a table where a full row of each column is searched at a time. Being a first-in-first-out algorithm, the children of the current node are always placed at the back of the queue. By doing this, BFS can search level-by-level, fully exploring each level as stated earlier. If there is a valid path from the start city to the goal city, BFS will find it due to its thorough search, making it correct in finite cases. If there is no valid path, BFS will be able to return empty since all possibilities are explored.

Our specific implementation of breadth-search was able to find a path from Oradea, Timisoara, and Neamt to Bucharest (weight included), as expected and shown below:

```
# Path from Oradea to Bucharest via BFS
path_oradea_to_bucharest = ['Oradea', 'Sibiu', 'Fagaras', 'Bucharest']
weight_oradea_to_bucharest = 461

# Path from Timisoara to Bucharest via BFS
path_timisoara_to_bucharest = ['Timisoara', 'Arad', 'Sibiu',
'Fagaras', 'Bucharest']
weight_timisoara_to_bucharest = 568

# Path from Neamt to Bucharest via BFS
path_neamt_to_bucharest = ['Neamt', 'Iasi', 'Vaslui', 'Urziceni',
'Bucharest']
weight_neamt_to_bucharest = 406
```

1.1.2 Depth Search

In depth-first search (DFS), all children nodes are explored prior to the algorithm backtracking and searching any sibling nodes. This is comparable to a table where every row of a single column is searched at a time. Being a last-in-first-out algorithm, the children of the current node are always placed at the front of the queue. By doing this, DFS can search to the greatest depth, fully exploring all children as stated earlier. A downside of DFS is that if the search space contains any cycles, the algorithm may revisit already explored nodes and loop infinitely without making any search progress. If there is a valid path from the start city to the goal city, DFS may find it, however, the algorithm has the disadvantage of looping in certain finite cases. Since DFS explores the entire search space, if there is no path to the goal node, it will eventually return empty (in the case of no loops).

Our specific implementation of depth-search was able to find a path from Oradea, Timisoara, and Neamt to Bucharest (weight included), as shown below:

```
# Path from Oradea to Bucharest via DFS
path_oradea_to_bucharest_dfs = ['Oradea', 'Sibiu', 'Fagaras',
'Bucharest']
weight_oradea_to_bucharest_dfs = 461

# Path from Timisoara to Bucharest via DFS
path_timisoara_to_bucharest_dfs = ['Timisoara', 'Lugoj', 'Mehadia',
'Drobeta', 'Craiovata', 'Craiova', 'Pitesti', 'Bucharest']
weight_timisoara_to_bucharest_dfs = 615

# Path from Neamt to Bucharest via DFS
path_neamt_to_bucharest_dfs = ['Neamt', 'Iasi', 'Vaslui', 'Urziceni',
'Bucharest']
weight_neamt_to_bucharest_dfs = 406
```

1.1.3 A* Search

In A*-search, a priority cue is maintained in nondecreasing order of the straight-line distance from each city to the goal city (each node to goal node), by the given heuristic equation:

$$f(n) = g(n) + h(n)$$

Here, f(n) represents the total estimated cost from the start node to the goal node, g(n) represents the cost of the path from the start node to the current node, and h(n) represents a heuristic estimate of the cost from the current node to the goal node.

The algorithm will place nodes with a lower f(n) value closer to the front of the priority queue, which will balance the cost to reach the goal node. This allows A* to explore multiple paths to the goal node, making it a generally correct algorithm. There are cases where A*-search is not correct or is unable to be correct. Since the algorithm relies on the heuristic estimate and the cost function, specific conditions not being met may lead the algorithm to return empty. For example, if the heuristic function is inadmissible, the algorithm will overestimate the cost for a node to reach the goal node and may terminate its path. Going further, if the cost function decreases along the nodal path, the cost functions condition will be violated and the A* algorithm may backtrack to nodes already visited that have higher costs. Both possible issues may cause A* to return with an incorrect solution. If all goes right in the algorithm, A* can find an optimal path to the goal node or return empty after it has explored all areas and concluded that no path exists.

Our specific implementation of A*-search was able to find a path from Oradea, Timisoara, and Neamt to Bucharest (weight included), as shown below:

```
# Path from Oradea to Bucharest via A*
path_oradea_to_bucharest_a_star = ['Oradea', 'Sibiu', 'Rimnicu',
'Vilcea', 'Pitesti', 'Bucharest']
weight_oradea_to_bucharest_a_star = 429

# Path from Timisoara to Bucharest via A*
path_timisoara_to_bucharest_a_star = ['Timisoara', 'Arad', 'Sibiu',
'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
weight_timisoara_to_bucharest_a_star = 536

# Path from Neamt to Bucharest via A*
path_neamt_to_bucharest_a_star = ['Neamt', 'Iasi', 'Vaslui',
'Urziceni', 'Bucharest']
weight_neamt_to_bucharest_a_star = 406
```

1.2 Efficiency

The following sections discuss the efficiency of each algorithm implementation, specifically, the number of visits needed and total cost for each algorithm to find a path or return empty.

1.2.1 Breadth Search

As described in the 'Correctness' section above, breadth-first search (BFS) works by exploring all nodes at a level prior to searching any child nodes. Since BFS searches level-by-level, it ensures that the first time that the goal node is reached, it is reached via a short path. In terms of optimization, the shortest possible path isn't always guaranteed since BFS does not consider the weight of the path leading to the goal node. When it comes to timing, if the goal city is located deep in the search space there may be a significant number of cities to visit, which could increase the time needed for the algorithm to search for the goal city.

BFS keeps track of all visited nodes, therefore there is a possibility for a great deal of memory to be consumed as the algorithm stores node information. BFS is generally efficient at indicating that no path exists, especially since no nodes are revisited and all possibilities are exhaustively explored. In short, BFS is generally efficient for search spaces, however, time and memory usage skyrocket as that search space expands and the cost is not guaranteed to be as low as possible.

1.2.2 Depth Search

As stated in the prior section, depth-first search (DFS) explores all children of a node prior to exploring any sibling nodes and their associated child nodes. Since DFS explores a single branch to its complete depth before backtracking, the path found is not always the shortest or most efficient. In DFS, only a single path of memory is consumed at a time (depth of branch), making it generally more optimal than BFS in that regard. Like breadth-search though, DFS can also have an increased amount of time needed to search for the goal node. Algorithm time is most likely to increase if the goal node is extremely deep in a branch, or if there are many nodes on one branch to search through in general. This can have the algorithm exploring a very large number of nodes before determining if a path does or does not exist.

To generalize the past statements, the structure of the search space can be the difference between DFS visiting only a few cities (efficient, less time) or many cities (inefficient, substantial time). Since DFS does not take the cost of cities into account as it finds a path, the most optimal path is not guaranteed, and the associated cost may be quite high. Due to backtracking being a characteristic of DFS, there is always the possibility for explored nodes to be revisited- it is important to note that the efficiency of this algorithm can be drastically reduced if there is a cycle present in the search space (infinite loop).

1.2.3 A* Search

As described in the ‘Correctness’ section above, A*-search works by creating a priority queue based off the straight-line distance from a node to the goal node; using a function that sums the actual cost to reach the node from the start node (g(n)) and the estimated cost to reach the goal node (h(n)). This function provides the ‘priority’ value that may put a node at higher precedence in the queue (if the SLD is low). This prioritization allows A*-search to explore nodes in order of non-decreasing SLD and optimize the total cost to be as low as possible. If an admissible heuristic is used, A*-search will terminate with no goal-node or supply the most cost-effective path to the ending node given. With its informed-decision design, A*-search should visit fewer nodes than its BFS and DFS counterparts- the algorithm simply does not need to visit all nodes since it utilizes a priority queue. Of course, like the other algorithms, as the search space becomes larger, the number of visited cities and memory usage may rise. Memory usage becomes especially intensive when there are more nodes in the search place, as A*-search adds them to their proper place in the queue.

Again, as stated earlier, there are conditions that may cause the A* search algorithm to be inefficient. The algorithm itself relies on an admissible heuristic, meaning that the true cost to reach the goal node is not overestimated. If the heuristic is inadmissible (e.g. negative weights), A*-search may not be able to find an optimal solution, or a solution at all. In short, A*-search places priority on finding the lowest cost path to the goal city, which is generally the most optimal path, and is relatively time efficient due to node prioritization.

1.3 Weight Results

The table below shows the difference in weight between all three algorithm's paths from each city (Oradea, Timisoara, and Neamt) to Bucharest.

City	**Breadth-first**	**Depth-first**	**A***
Oradea to Bucharest	461	461	429
Timisoara to Bucharest	568	615	536
Neamt to Bucharest	406	406	406