Artificial Intelligence – Assignment 1

Joe Ferguson, Brett Marchese, Brad Greene

*All team members contributed in equal measure*

# Honor Statement

In completing this assignment, all team members have followed the honor pledge specified by the instructor for this course.

# Bibliography

Russell, Stuart, and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed., Pearson Education Limited, 2020.

# Execution Instructions

After downloading the code, ensure that the python files are all in the same directory.

Our program was written in python. To run it, the following command can be used

$ python runthis.py

Make sure you run this command in a terminal that is in the same directory as the code you downloaded. Once ran, a series of prompts will be asked. From there you can respond with yes(Y) or no(N) options to run our demonstration, performance test, or exit the program.

# Implementation Details:

Implementation of the Romanian Map, *State*, and program:

The Romanian Map data was implemented as a class with parallel arrays of size 20, which is the number of nodes/cities in the map. They are sorted by alphabetical, numbered 1-indexed, and the relationships/paths/edges are stored in an adjacency matrix. Heuristics such as the SLD distance (mentioned in the textbook) and nodes\_away are stored in this class as well, ordered the same way as the other arrays. Nodes\_away is another heuristic that was made according to the requirements of this assignment, that tracks how many paths or edges a node/city is from the goal node/city. In this case, that would be Bucharest.

*State* was a commonly used pseudonym in the textbook and was recognized as an important characteristic that was needed to run the search algorithm. It was implemented as a class that stores the current occupied node/city, the path traversed thus far, the cost accumulated thus far, and the *outcome* of the traversal thus far. *Outcome* is a flag that tells if the *State* has reached the goal, failed to reach the goal, or has yet to fail or succeed. An additional class was made named *INF\_State*, which inherits all attributes from *State* with an additional f\_cost attribute which tracks the heuristic of the *State* for the informed searches. This was just a more convenient way to run the informed searches.

The main program is run in python (all of the classes were made in python as well), which imports all of the classes and search algorithms and has functionality to run searches or performance tests based on what the user wants. This was developed to determine the performance of each search algorithm.

Implementation of the uninformed depth-first search algorithm:

The depth-first search algorithm was developed according to the pseudocode of the recursive depth first function in the textbook. The steps are as follows:

1. Check if the current city/node is the goal node and return *State* if true, else continue.
2. Check if we have recursed past our limit of *n* times and set our *State* as a failure and return *State* if true, else continue.
3. Iterate through all the edges/paths from our current city/node (which are in order in the adjacency matrix) , copy our *State* into a new *State* with each of these new options and then recursively call the algorithm on this new state.

With this iteration, outside of the child recursion, check if our new *States* have found a goal path, or if they got cut off. If any of them have found the goal path, return that *State* immediately. If none of them have, set our *State* as a failure and return it.

Implementation of the uninformed breadth-first search algorithm:

The breadth-first search algorithm was developed according to the pseudocode of the breadth-first search function in the textbook. The steps are as follows:

1. Instantiate a FIFO queue to store *States* to be processed.
2. Instantiate a python set (an array that does not permit duplicates) to store explored nodes/cities.
3. Loop through cities/nodes while the FIFO queue (which we call frontier) contains at least one *State* (is not empty).
   1. Pop the *State* from the front of the queue and check if it has reached the goal, return it if it has, if not continue.
   2. Add the *States’s* current node/city to the explored set.
   3. Iterate through the edges/options open to the current *State* and create new *States* for each child edge/option available. As long as these children are not already explored or already present in the frontier queue, and as long as they are not successes (in which we would return it by checking inside the loop), we then add each of these to the frontier queue.
4. If after iterating and upon having an empty frontier queue (which ends our first loop), we then return failure. This will only happen if the map itself is incomplete, which is not the case for the Romanian map. As breadth-first search will always find a path given that it is possible, this will not happen in this assignment.

Implementation of the informed greedy search algorithm:

The greedy search algorithm was developed loosely by using some of the pseudocode of the A\* algorithm in the textbook, and by natural knowledge of greedy algorithms (which are all based on a basic principle). It is a recursive function. The steps are as follows:

1. Check if our current *State* is at the goal, if it is return the *State*, if not continue.
2. Make a list of successors that contain *States* that hold the options/children to travel to. If there are no successors, return a failure, if not continue.
3. Determine the heuristic cost of each option using f(n) = h(n).
4. While we have not found a success and have not been cutoff, keep iterating. During each iteration, sort the successors by heuristic cost and recursively call our own function with the best *State* (based on heuristic cost), and store this in a variable. If the outcome of this call is a success or failure, return it accordingly.

Implementation of the informed A\* search algorithm:

The A\* search algorithm was developed according to the pseudocode of the A\* search algorithm in the textbook. The difference between this search algorithm and the greedy search algorithm is the calculation of the heuristic cost. The steps are as follows:

1. Check if our current *State* is at the goal, if it is return the *State*, if not continue.
2. Make a list of successors that contain *States* that hold the options/children to travel to. If there are no successors, return a failure, if not continue.
3. Determine the heuristic cost of each option using f(n) = g(n) + h(n).
4. While we have not found a success and have not been cutoff, keep iterating. During each iteration, sort the successors by heuristic cost and recursively call our own function with the best *State* (based on heuristic cost), and store this in a variable. If the outcome of this call is a success or failure, return it accordingly.

# Experimental Results

|  |  |
| --- | --- |
| **Search Algorithm** | **Performance (ms)**  **(20 nodes x 5 iterations = 100 total searches)** |
| Depth First | 2.523899 |
| Breadth First | 0.999928 |
| Greedy  (SLD Heuristic) | 1.00016 |
| Greedy  (Nodes Away Heuristic) | 0.500441 |
| A\*  (SLD Heuristic) | 1.999140 |
| A\*  (Nodes Away Heuristic) | 1.500845 |

*Table 1: Performance of the search algorithm*

|  |  |  |
| --- | --- | --- |
| **Search Algorithm** | **Space**  **(Theoretical)** | **Cost**  **(Actual = 20 Nodes)** |
| Depth First | O(bm) | 32774 |
| Breadth First | O(bd+1) | 5939 |
| Greedy  (SLD Heuristic) | O(bm) | 5986 |
| Greedy  (Nodes Away Heuristic) | O(bm) | 5939 |
| A\*  (SLD Heuristic) | Keeps all nodes in memory | 5779 |
| A\*  (Nodes Away Heuristic) | Keeps all nodes in memory | 5779 |

*Table 2: Time and Space Complexity*

Notes:

Each search was given the target city Bucharest. This was to use the straight-line distance heuristic given in the textbook.

Depth First Search returns the cost of 9999 if it was unable to find a path, which explains the extremely high cost. It also was given a limit of 20 recursions, the size of the map, which can explain the higher time to run as well. To limit this, one could implement loop-checking, however this would be unnecessary as depth-first search should not be used on graphs anyways.

To see the difference between breadth-first search and greedy, as well as greedy and A\*, a bigger map would be necessary. 20 nodes is not enough to clearly see the power of the informed searches over the uninformed searches.