Spenser Mason, U80942167

Intro to AI, Project 1

A* Search

Algorithms

The included Python program contained two variants of the A* search algorithm. The first used a heuristic based on straight-line distance to the endpoint. The second used a heuristic based on the number of links followed from the start point. Both algorithms followed the same procedure, differing only in how they calculated the "score" for certain possible moves at each step in the search. The algorithm starts at a given point, then iterates until it finds itself at the endpoint. Each iteration, the algorithm builds a list of its potential moves given its current position, as well as past positions that were skipped because their "score" was not the highest. The algorithm then determines the "score" for each potential move through the use of a heuristic function. The move with the best "score" is then taken and iteration continues. This is very similar to the Breadth First Search algorithm, except that it evaluates potential moves in an order determined by a heuristic, rather than in the order of discovery. This allows the algorithm to avoid unnecessary moves that would be inefficient, and thus not part of the optimal solution. The key point of the algorithm is the heuristic function that is applied to every possible move to determine which is likely the best one to make. For the distance heuristic, the cartesian distance between the current position and the endpoint added to the total distance traveled is used as the heuristic "score." For the fewest links heuristic, the number of links followed on a given path from the start is used to determine the "score."

Program Design

I designed the program as a Python script for simplicity, portability, and ease of use. The program first parses the input files and builds the map, then executes an A* function to build the path. This function accepts an argument that determines which heuristic to use. Calling the function with the use_distance parameter set to True will use the cartesian distance for the heuristic. Calling the function with this parameter set to False will use the fewest links heuristic. The output of the function is a list of Cities. A City is a simple data structure that holds a string name, two integer coordinates for its x and y position, and a list of references to other Cities which can be reached from the City. Using such a data structure simplified the design of the algorithm, as the connections list made it possible to explore the map similarly to a linked list, which is analogous to how one would travel between cities in real life. This was made possible by Python's memory management, as a language with no garbage collector would make it difficult to maintain such a complex web of references without causing memory leaks.

Results

The results were as expected. However, the path taken by this program was different than the provided sample output when using the fewest links heuristic. This is due to the method used for sorting the list of possible moves by score. Since the fewest links heuristic causes many paths to have the same score, different sorting algorithms would cause different ordering of these equivalent paths. In this case, the default Python list sorting algorithm was used.

Performance

The performance is extremely fast, even on large data sets. For instance, the program was tested on a randomly generated map of 1,000 cities with a total of 98,775 roads. The algorithm took 0.701 seconds on the fewest links heuristic and 0.031 seconds on the distance heuristic to find the path for this large map. The system used to achieve these times was a Windows 10 laptop with an Intel i5@2.7GHz processor.

Task Delegation

This was accomplished alone by Spenser Mason