Route searching – report

Made by Jakub Wawrzyniak

# Task 1 – Creating the set of cities

This task is handled by the World() class, which in its constructor creates a list of City() objects and then connects them together. The distances between the cities are calculated only once, during initialization of the World() object; they are then stored in the City.distTo object of each City instance.

The city connections are performed as follows: first we compute all unique pairs of cities (this is done by the newton() function), then we randomly choose specified percentage of the pairs, obtaining the pairs that should be connected. Then we iterate over those pairs, calculating distances between them and connecting them together. It’s worth pointing out that this method does not guarantee that every city will be reachable from every other one.

# Task 2 – Graph representation

The graph is represented as a list of City objects, where each instance of City contains references to its neighbours. As previously mentioned, every City also has a distTo object, which contains information about distances to all of its neighbours.

# Task 3 – Travelling salesman

Each algorithm uses the Path object to represent the current state. The Path object contains the list of City objects, which are the nodes of the path as well as the current distance, which is the sum of distances between the cities stored in this.nodes[] list.

## Breath- and depth first searching

The BFS algorithm is probably the most straightforward one. For each element taken form the paths list it creates all new possible paths, which are obtained by extending the taken path by each neighbour of the last path.node element (this is exactly path.end), provided that the neighbour does not appear in the path previously. This step is repeated until the path list has the length of world.cities, meaning it already reached all cities exactly once. Then the algorithm chooses the best path using the chooseBetterPath() function. If no path can be found, it returns undefined.

It is woth pointing out, that this algorythym uses so much memory, that node.js refused to compute the path for no of cities higher than 8. When the app is run in Chrome, the website crashes for no of cities greater than 11. This algorithm is the slowest of all of them, which is slightly surprising, as I would expect the DFS algorithym to perform slightly worse than BFS due to it’s recursive nature. I suppose the reason might be connected to the constant memory allocation that the BFS requires, which might need more and more time as we approach the limits of the available memory in the V8 engine.

The DFS algorithm is about as simple as the BFS one. It traverses all cities that are available from the last visited city but weren’t visited before. Then when the path length reaches no of all cities, is tries to add the first node to the end, if that’s possible it compares the last best path with the current one, and returns the better value. It is a recursive algorithm. Works reasonably fast when the no of cities is below 11. Returns undefined, if no path can be found

## Greedy algorithm

The greedy algorithm is also recursive, but it differs in a few ways from the DFS approach. First, it returns the first valid path it encounters, so no need to use chooseBetterPath() here. Second, before searching through all of the neighbours of the last visited city is sorts them by distance. Therefore, this algorithm tries to get to the closest cities first. It returns -1, if no path can be found.

## Minimum spanning tree

This solver is by far the most complicated one. First think it does is it computes the minimum spanning tree using Prim’s algorithm: it lists all nodes that are reachable from the current state of the tree, chooses the closest one and adds it to the tree. It does it until the tree contains all available nodes. Then, it recursively traverses the tree and terminates when all nodes has been visited.

This solution differs from the previous ones as it can’t visit every city exactly once .Some backtracking is necessary as the minimum spanning tree cannot contain loops (otherwise it wouldn’t be a tree). Therefore, the length of the path is always twice the sum of all connection lengths in the tree.

It’s worth pointing out, that the created minimum spanning tree is a list of City objects, that are not connected with the cities from world.cities, though they have same ids and (x,y) coordinates. It was made that way, so that I could freely reduce the number of neighbours of each city without destroying the original world.

## Task 4 – Bidirectional search

This solver is contained is