Name: Alexander Curwood User-name: wxnq87

Algorithm A: Basic Greedy Search

Algorithm B: Greedy Best-First Search

Description of enhancement of Algorithm A:

The basic implementation of greedy search selected the next node to be included in the path based on its distance from the current end node. The enhanced the algorithm instead selects the next node based on the greedy completion cost through the remaining unvisited cities, starting from the neighbours of the current node. Each neighbour of the current end node is added to a copy of the current path, which is then passed through a function calculating the cost of a greedy completion. The neighbour with the lowest cost is chosen, and this process is repeated a complete tour is found. Note that no biasing factor is needed as the algorithm is selecting between paths of equal length each time.

Description of enhancement of Algorithm B:

The basic implementation of greedy best-first search used the estimated cost of a greedy completion through the remaining unvisited cities, from the current end city, as a heuristic function. However, I noticed that the lowest cost path which was pulled from the queue would almost always be the path which is furthest from the start node, which is consistent with the heuristic function used (the estimated cost of a path through the remaining unvisited cities will most often be lower for a path which has visited more cities already). This means that the algorithm was often not backtracking, causing it to potentially miss promising paths due to the heuristic’s bias towards longer paths within the queue. Whilst this significantly decreases the algorithm’s run time, I wanted to experiment with removing this bias by introducing extra parameters within the output of the heuristic function. This would allow me to manipulate the trade-off between algorithm performance, which would be improved by expanding potentially missed paths, and runtime.

During experimentation, I had the algorithm print each lowest heuristic cost path which was being pulled from the search tree, to analyse the algorithm's backtracking. The first method attempted was simply multiplying the output of the heuristic function by the length of the path considered (therefore increasing the heuristic cost of longer paths to bias the algorithm against them). However, this led to the algorithm essentially expanding every potential path within the search tree, compromising the algorithm's runtime significantly and indicating that the factor should be a reciprocal power of the considered path length. Through experimenting, I found that the closer the exponent of the path length became to 1, the more backtracking the algorithm conducted, increasing the quality of the tours but drastically increasing the runtime also. Whereas the closer the exponent became to 0, the less backtracking the algorithm conducted, having the inverse effect.

Ultimately, I found that a factor of the path length to the power of 1/5 yielded the best results for distance matrixes with sizes less than 60, where the increased runtime is offset by the smaller tours being considered. For distances with sizes between 60 and 180, I found that a power of 1/50 yielded the best results, and for distance matrixes of size greater than 180, I found that a power of 1/5000 was around the maximum possible without massively compromising performance.