When reading the city files, a two-dimensional array is used as a distance matrix. Alternatives such as adjacency lists and edge lists were considered but as the city files are complete graphs, the denseness makes distance matrices the optimal choice as they are far more effective at retrieving connections between nodes and performing operations on such as Prim's algorithm.

Secondly, lists are constructed to store the order of paths and tours. Initially, strings were used to manipulate tours and add nodes to paths which was unnecessarily complex and time inefficient. By changing to a list, finding and storing new candidate tours for the simulated annealing algorithm becomes a simple process of selecting elements in the list (representing nodes in a tour) and performing operations on them. Likewise, for A*, nodes easily can be appended to the end of paths.

Next, I decided against sorting the temperature and path lists as it is far quicker to search for the maximum value than it is to order the lists by costs.

A* Search

A* is a search algorithm used for pathfinding and graph traversal. Considered as an extension of Dijkstra's algorithm, A* combines the distance from the start node with a heuristic function to calculate an estimate for the total cost a path from the start node. The heuristic function I have formulated is the following:

$$h(p_z) = MST(q) + E(p_0, q_i) + E(p_z, q_j)$$

Generally, the A* algorithm is used to find the shortest path between two different nodes, where the heuristic function is an estimate cost of the cheapest path from node z to the goal node. By calculating the length of the minimum spanning tree for the dynamic graph q (made up of all nodes excluded in the path p), the goal state is no longer a node but a tour such that all cities are visited. Added to the minimum spanning tree is $E(p_0, q_i) + E(p_z, q_j)$, which represents the two shortest connections between the current path p and the nodes in the minimum spanning tree, q. As the tour solution is a cyclic path, only the end nodes of the path p_0 and p_z will be able to accept a new connection so the heuristic function pairs up the two nodes with q_i and q_j, such that the shortest connections between the path and the minimum spanning tree are added to the total cost. In result, by adding the heuristic to Dijkstra's algorithm, an approximation for the total cost of the tour is can be made for every found path.

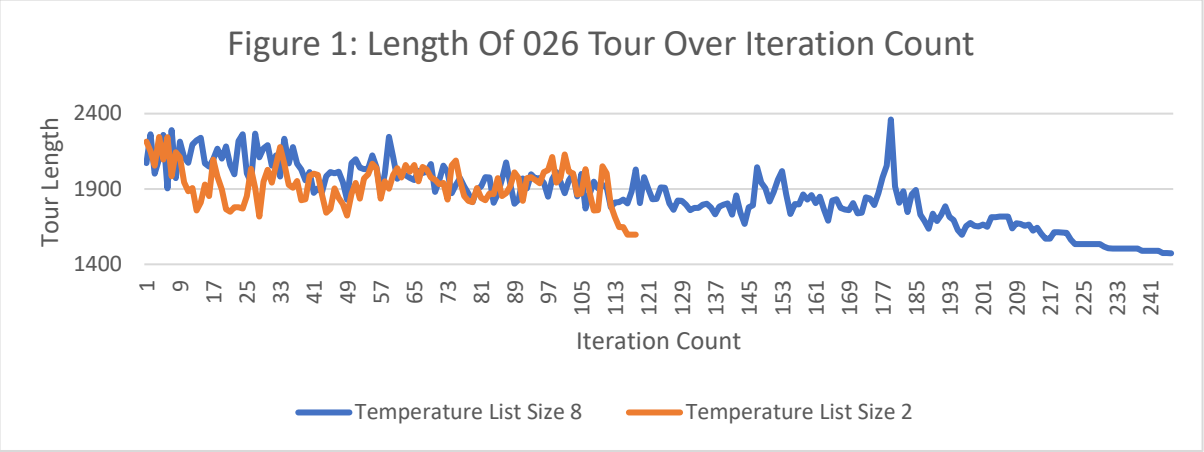
After finding it unnecessarily excessive to perform operations on strings, I decided to use lists for storing path orders, allowing nodes to be 'appended' to paths. For efficiency purposes, I store the total cost of each path with the path orders using a class. An alternative option would be to use a hash-map since each path is unique, however paths are stored as lists, which are unsuitable for hash-keys as they are mutable.

List-Based Simulated Annealing Algorithm

Simulated annealing is a probabilistic technique that can be used as a metaheuristic to find a solution for the travelling salesman problem. The implemented solution for the travelling salesman problem begins by defining an initial tour solution [1, 2, 3, …, n]. I deemed the tour unnecessary to randomise as it subsequently will be rearranged enough times after such that the initial tour will be ineffective towards the final solution.

An essential property of simulated annealing for the problem is the ability to replace the current tour with a worse tour, as it allows the search to explore more paths in aim of finding a global minimum. By implementing a list-based cooling schedule of size 8, the likelihood of the tour being

trapped in a local minimum is significantly reduced, as demonstrated in figure 1. This is due to the adaptiveness of the temperature values. Usually, when a significant tour is found for a linear system, the temperature tends to drop rapidly, making it difficult to explore new tours. On the other hand, by making note of the temperature in the list and switching to the new maximum, the implemented list-based algorithm has a more controlled cooling schedule.



When searching for new tours, candidate solutions are generated and compared to the current tour. Whenever a candidate tour is greater than the current tour, an acceptance probability is calculated:

$p = e^{\frac{c_{curr} - c_{new}}{t}}$, where t is the maximum temperature and c is the tour length.

By using the standard acceptance probability, temperature has a considerable impact on the final value, allowing more tours to be accepted in the earlier iterations. If the (worse) candidate tour is accepted (through the Metropolis acceptance criterion), a new temperature is calculated from the following formula:

$t = \frac{c_{curr} - c_{new}}{\ln(r)}$, where r is the random value such that $0 < r < 1$ and c is the tour length.

Dividing the difference in tour lengths by $\ln(r)$ ensures the function is (linearly) monotonically decreasing whilst allowing the randomised value to have a notable effect on the new temperature. When initialising the temperature list, r is set to 0.9, to establish a high variance of temperatures. The number of trials for candidate solutions before the temperature list gets updated is 'n' (the number of nodes in the tour). This allows for multiple candidate solutions to be accepted, increasing the ability to explore the solution space once more.

Tour Results

File Number	A* Search	List-Based Simulated Annealing (best result after 5 runs)
012	56	56
017	1514	1444
021	3089	2549
026	1762	1473
042	1340	1337
048	15264	13172
058	27978	25395
175	22475	22461
180	Stack Overflow	3720
535	50683	55679

A* Search

As previously mentioned, the A* algorithm is generally used to find the shortest path between two different nodes, where the heuristic function is an estimate cost of the cheapest path from node z to the goal node. For A* to be applied to the travelling salesman problem, the heuristic function needs to reach zero once all nodes are traversed. Through research, I found minimum spanning trees to be prevalent in heuristic functions for the travelling salesman problem. Using Prim's algorithm to calculate the minimum spanning tree for non-connected nodes in a graph and by adding together the found path and the minimum spanning tree, a good approximation for a tour's total length given a path can be made after the first iteration of the algorithm.

With a running time of $O(|V|^2)$, executing Prim's algorithm for each individual new path is heavily time consuming. To significantly improve performance, the implemented solution avoids running Prim's algorithm for every node when branching the path with the lowest cost. Instead, the algorithm pre-computes the minimum spanning tree of the non-connecting nodes including the branch node, reducing the number of runs from the total number of nodes minus the number of nodes in the path to once every iteration. The trade off is a slight drop in accuracy of the approximation, but running time considerably improves.

To enhance the approximation of the total length of the tour, the two edges are added to the heuristic representing the shortest connections between the path and tree. As the final solution is a tour in which each node has two connections, the nodes in the found path connecting to the minimum spanning tree are set to be the first and last, as the other nodes have two connections already and are no longer looking to branch. Likewise, the connection nodes in the tree are forced to be unique else there is the possibility that the approximation includes a node branching to three other nodes.

For time purposes, the algorithm finds the two shortest connections in succession, meaning only the first connection influences the second and not vice-versa (i.e. node 'x' used in the first connection can no longer be used for the second connection, consequently causing the program to ignore a small portion of potential solutions). To improve the approximation, the algorithm should compare the combined minimum of the two connections with the current lowest solution, ensuring all possible connections are found. With a negligible effect on the final tours produced by the search, I decided the extra computational time the algorithm required was unnecessary and hence reverted to calling the two connections in succession.

As each city file stores complete graphs, the number of paths found after each iteration of the algorithm increases at an inadequate rate. To ensure the algorithm terminates in polynomial time, the list of paths found needs to be culled. Using the following formula, paths greater than the filter value are removed from the list.

$filter\ value = p_l + \frac{\mu - p_l}{2n}$, where p_l is the lowest path value and where μ is the mean cost of paths in the list. Dividing the difference between the mean and lowest costs by the number of paths in the list improves consistency in the number of results accepted. Reducing the probability of accepting new solutions based on list size makes it difficult for to the list grow. Unfortunately, city file 180 is an anomaly, having multiple paths having the same lengths. This causes a stack overflow to occur during run-time as too many paths are stored, filling up memory.

Lastly, the results for the implemented algorithm varied drastically based on the initial starting node of the paths. To accommodate this, a good spread of results is achieved by repeating the A* algorithm a maximum of 25 times, starting at a unique node each time. I chose against running the

algorithm for each individual node due to diminishing returns and time dependencies. As the number of found tours increases, the likelihood of finding a shorter tour than all the previous decreases.

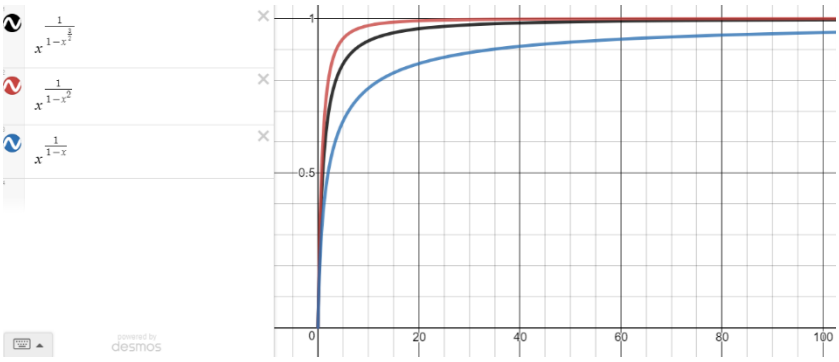
List-Based Simulated Annealing

When searching for new tours, three candidate solutions are generated by a. swapping two nodes, b. inserting a node at a position or c. inverting all nodes between two positions. By producing three potential solutions and selecting the candidate tour with the lowest length, the probability of finding more optimal routes increases. This escalates the rate of cooling, reducing the number of iterations required to reach the terminating condition.

Moreover, the effectiveness of a simulated annealing algorithm is greatly defined by the quality of the cooling system. Originally, when updating the temperature list, only the mean value of the new temperatures was used when replacing the previous maximum temperature. If no worse candidate solutions were found, the maximum temperature would remain the same. Unfortunately, this led to issues terminating the algorithm. Once a near-optimal solution was found, the maximum temperature struggled to decrease and fixed at varying temperatures. I therefore implemented a secondary cooling schedule. Having experimented with multiple conditions, I have finalised with the following formula:

$$t_{new} = \frac{t_c}{c} \times Var(T)^{\frac{1}{1-Var(T)^{\frac{3}{2}}}} \times 0.999^{count}$$

$Var(T) = E[T^2] - E[T]^2$ is the variance of the temperatures in the list, t_c is the cumulative total of all the temperatures created by worse solutions and c is the number of worse solutions that were accepted. As tour solutions improve, temperatures in the list drop and in turn decrease the variance of the list. As demonstrated in figure 2, $Var(T)^{\frac{1}{1-Var(T)^{\frac{3}{2}}}}$ clearly begins to take effect for $T < 50$, compared with $T < 20$ for $Var(T)^{\frac{1}{1-Var(T)^2}}$, which I found to be the ideal value to begin lowering temperatures in list in aim of terminating the algorithm with a near-optimal solution. 0.999^{count} is included in the cooling value to ensure the program terminates in polynomial time. The terminating condition is set as max(t) < 0.025. As the maximum temperature decreases rapidly once smaller than 50, the minimum acceptable solution can be set to a very small value.



In conclusion, the results for the list-based simulated annealing tends to be somewhat shorter than the results for A* search. However, simulated annealing has an element of randomness, meaning occasionally very poor results are output as solutions. Comparatively, given a specific graph, the A* search outputs the same result every time. A reliable, near optimal solution can therefore be found using the A* search, giving both algorithms and advantage over the other.