# A*

## 1 Implementation

To formulate the Travelling Salesperson Problem as a search problem, a *state* needs to be represented as a partial tour of length less than or equal to **n**, where **n** is the number of cities. As I needed to use certain 'properties' of a partial tour frequently, such as its *f*, *g* and *h* values, I decided to implement a PartialTour class. Instead of representing a partial tour as an array and having to calculate the length of the partial tour manually when required, my PartialTour class has an instance variable called *g*, that is set upon creation of the partial tour and can be accessed quickly thereafter. The other instance variables I used were *g*, *h*, *path* (an array holding the actual partial tour) and *endNode* (a string holding the final node of the partial tour). One other benefit of using object-orientated programming is Python's built-in *__cmp__( )* function. I defined this function to compare the *f* values of the two objects being compared, allowing me to use Python's *heapq* module, which I will discuss in detail shortly.

I decided to represent the connected graph of cities in two different formats, as I believed at the time there would be a significant performance increase when using the different implementations in specific cases. In fact, I have now come to realise that I could have achieved the same functionality using only the first representation, instead of using both. However, it would now be a significant task to change, and so I will discuss both implementations as both are still used in my code. In my first implementation of the graph, I used a simple two-dimensional array, named *distances*. To access the distance between city *a* and city *b*, I can simply call *distances[a][b]* or *distances[b][a]* (I ensured the matrix was symmetrical upon creation). This implementation allows for fast calculation of the distances between two cities. In my second implementation of the graph, I used Python's built-in dictionary mapping type, which I called *graph*. This dictionary maps each city to a list of its neighbouring cities, along with the distance to each city. I chose to also implement the graph this way as I believed dictionary lookups were quicker than array indexing, but I came to realise that simply calling *distances[a]* would return a list of city *a*'s neighbours, and on average dictionary lookup and list indexing both have O(1) time complexity in Python. However, I will explain the implementation nonetheless. The key-value structure of the dictionary is as follows: *graph[city] = [Node]*, where *[Node]* is an array of *Node* objects, and *Node* is another class I implemented. Each *Node* object has two instance variables: a string *name* and an int *dist*. The *name* variable represents the name of the city, and *dist* holds the distance from this city to the city key.

To run efficiently, the A* algorithm requires a priority queue implementation of the fringe. This ensures that when a partial tour is popped from the fringe, the partial tour with minimal *f* value is the one removed. To implement a priority queue in Python, I used the *heapq* module, which is a binary-search tree implementation of a priority queue. I used the *heappush( )* and *heappop( )* functions to push and pop my PartialTour objects on and off the fringe, which execute with O(log(n)) time complexity. This is much faster than having to search through an array of PartialTours to find the shortest one (O(n) time complexity). Furthermore, this is where the aforementioned *__cmp__( )* function is used, as it instructs the *heapq* module on how to compare two PartialTour objects.

One final implementation detail worth noting is that I also created Writer and Parser classes to handle the input and output of the tour data. In doing so, I ensured my code was kept clean and organised.

## 2 Experimentation

As the A* algorithm (with an admissible heuristic) is complete, I didn't expect the algorithm to terminate in a feasible time for the larger cityfiles. I found that my algorithm only found optimal solutions without running out of memory for the cityfiles containing 12, 17 and 180 (due to the abundance of 0s) cities. For the cityfiles containing 21 and 42 cities, my non-admissible heuristic managed to find sub-optimal tours. I experimented with several heuristics, and compared the average execution time for the 12 and 17 cityfiles.

## 2.1 Nearest Neighbour Heuristic

My initial, very simple heuristic worked by calculating the distance to the nearest neighbour from the final node in the partial tour, reflecting that we always want to add a city that is close to our current tour. Although admissible, this heuristic is extremely inaccurate in its estimation of the length of tour connecting the unvisited cities, and so fails to 'guide' the algortithm towards an optimal solution quickly.
**Average Execution Time:** [N/A - Did not complete in feasible time]

## 2.2 Nearest Neighbour and Distance to Start Heuristic

The next heuristic I tried is an improvement of the previous one, and works by calculating the distance to the nearest neighbour from the final node in the partial tour as before, but then adding to this the shortest distance from the first node in the partial tour to any other node not in the partial tour. This heuristic reflects that we always want to add a heuristic that is close to the end of our current partial tour, but not too far from the start node, to which we eventually have to return. Again, the algorithm did not terminate.
**Average Execution Time:** [N/A - Did not complete in feasible time]

## 2.3 Minimum Spanning Tree Heuristic (Non-admissible)

I realised that I needed a more accurate estimation of the cost of completing the tour from the current partial tour, and realised that calculating a minimum spanning tree of the remaining nodes would provide me with such an estimate. My first attempt at a minimum spanning tree heuristic improved the performance of my A* algorithm by quite some margin, but unfortunately yielded sub-optimal tours. I initially thought this was due to incorrect implementation, but after some time I realised this is in fact due to the incompleteness of this heuristic. This heuristic worked by using *Prim's algorithm* to construct a minimum spanning tree of all of the unvisited nodes, which occasionally may overestimate the true cost of completing the tour, as it effectively 'double counted' the final edge of the new partial tour being created, once in the *g-cost* and once in the *h-cost* calculations. This overestimate causes the algorithm to become incomplete, and thus is not guaranteed to return an optimal solution. I would have liked to have tested this heuristic on the larger cityfiles, however, as suboptimal heuristics decrease the execution time. They do so by forcing the A* algorithm to run more like a greedy-algorithm, favouring partial tours with a lower *g cost.* Unfortunately, I did not have time to do so.
**12:** *Av. Time:* [0.07298 seconds] *Length:* 56, **17:** *Av. Time:* [122.7972 seconds] *Length:* 1444

## 2.4 Minimum Spanning Tree Heuristic (Admissible)

The admissible version of this heuristic, which performed the most successfully out of all of the heuristics I trialled, works as follows: it sums the shortest distance from the final node in the partial tour to any other unvisited node, with the minimum spanning tree of the remaining unvisited nodes (not including the node closest to the end of the partial tour), and the shortest distance from any of the unvisited nodes (again not including the node closest to the end of the partial tour) back to the start. This heuristic never overestimates the cost of completing the tour, and so is admissible and thus complete. However, calculating the MST every time the fringe is expanded is very computationally expensive, and so to improve the performance of the A* algorithm further I decided to experiment with different implementations of the heuristic, before experimenting with any further heuristics.

My initial, naive, implementation of the MST ran with $O(n^2)$ complexity, as it looped through every child of every node currently in the MST, adding the closest, and then restarting. My second experimental implementation ran as follows: all the vertexes are pushed on to a priority queue, with a 'distance to MST' property that is used as the priority, initially set to the maximum system size (except one). The vertex with the shortest distance is popped off the queue, added to the MST, and its distance added to the total. Then, the algorithm loops through every remaining vertex on the priority queue, comparing the distance between the node recently added to the MST and the remaining vertex, with the current 'distance to MST' value of that vertex. If the distance between this vertex and the node recently added to the MST is smaller than the current 'distance to MST' value of the vertext, its 'distance to MST' value on the priority queue is updated. However, in changing this value we break the heap invariant, and so must call the *heapq.heapify( )* function, re-establishing the list as a priority queue. Pushing on to and popping off the priority both have O(log(n)) time complexity, but the *heapify( )* function runs with O(n) time complexity. The overall time complexity for this experiment is, therefore, O(n + Elog(n)) (E being the number of edges in the graph) - an improvement on O($n^2$), but still with room for improvement.

The implementation issue with using *heapq* for our priority queue is that it does not facilitate the modification of a priority after it has been pushed onto the queue. To overcome this, I implemented my own push( ) and pop( ) functions to fascilitate updating values on the priority queue. Instead of the vertexes themselves being pushed onto the queue, an *entry* tuple of [shortestDistance, node] is used, along with a map of nodes to entries, called *entryMap*. When a node/vertex is to be pushed onto the stack, we first check if it is in the *entryMap*. If so, then it already has a value on the priority queue, so we find this entry in the *entryMap*, set its node value to 'UPDATED', create a new entry with the new shortest distance, push this entry on to the heap, and map the node to this entry in our *entryMap*. When we pop the top entry off the heap, we check to see if its node value is 'UPDATED' - if so, we continue popping values off the priority queue until a non-update entry is found. This implementation of the MST heuristic has O(Elog(n)) time complexity, the fastest possible for *Prim's Algorithm* without using a *Fibonnaci Heap.*
**12:** *Av. Time:* [0.09207 seconds] *Length:* 56, **17:** *Av. Time:* [122.7972 seconds] *Length:* 1563

### 2.5 Greedy Completion Heuristic and IDA*

Had I more time, I would have continued my experimentation by first implementing a greedy completion heuristic, which would work by simply applying the nearest neighbour algorithm to the unvisited nodes. It is unfortunate that I did not have time to do this, as I believe this heuristic may well have outperformed my admissible MST heuristic. I then would have gone on to implement the IDA* algorithm, which is very similar to normal Iterative Deepening, except that at each iteration, the depth-first search is cut off when a branch's total cost (g + h) exceeds a given threshold, rather than at a specific graph depth. This threshold starts at the estimate of the cost of the initial state, and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

# Simulated Annealing

## 3 Implementation

The simulated annealing algorithm is comparatively easy to implement, and did not require any interesting data structures that I have not already discussed. Similarly to A*, I opted to use both the *distances* two-dimensional matrix and *graph* dictionary representations of the matrix, incorrectly believing this would increase performance. The implementation details worth noting for Simulated Annealing are all a consequence of the extensive experimentation process. As there are many 'variables' in this algorithm, I parameterised the algorithm using global variables, which could then be changed programmatically. I defined INIT_NODE, BETA, INIT_TEMP, OPT, RANDOM, and SUCCESSOR global variables, all of which affect the running of the program. If RANDOM is set to *true* then INIT_NODE is ignored, and the algorithm generates a random initial tour. If RANDOM is set to *false* then a greedy algorithm is used to create a 'good' initial tour, starting at the node defined by INIT_NODE. BETA defines the constant used in the schedule algorithms, OPT defines whether to use a 2- or 3-opt successor function, INIT_TEMP defines the initial temperature that should be used, and SUCCESSOR can be one of *exp, log, lin* or *quad* - specifying if exponential, logarithmic, linear or quadratic cooling should be used.

## 4 Experimentation

Initially, I began my experimentation by manually changing the variables explained above. I quickly realised, however, that this method was both illogical and inefficient, as I simply did not know which combination of variables would yield optimal outputs. Instead, I wrote a program to automate this process. Through research, I found that E. Aarts and J. Korst had proven that for the simulated annealing algorithm to converge asymptotically towards the set of global optimal solutions, the temperature of the system must decrease logarithmically with a 'BETA' value of 1. In practice, however, this cooling schedule requires an infeasible amount of time to execute, and so I decided not to test my program using the logarithmic cooling schedule. Furthermore, after my initial basic experimentation, I found that the linear cooling schedule never produced good quality tours either. Therefore in my first automated experimentation cycle, I focused on using the exponential cooling schedule. For every cityfile, the program ran once from every INIT_NODE (using a greedy algorithm to generate a reasonable initial tour), varying BETA twenty times between 0.99 and 0.1, and testing both the 2- and 3-opt successor functions for each value of BETA. I used an INIT_TEMP of 10 and a 'cut-off' temperature of 0.000001. Then, for every BETA/OPT combination, I ran the program a further 5 times starting from a randomly generated initial tour, rather than using a greedy algorithm. For each cityfile I recorded the variable values that yielded the shortest tour, and found that using a 2-opt successor function when starting from a greedy-generated initial tour gave the best results unanimously. On average, the BETA value that yielded the best results was 0.9958, and as the number of cities increased larger BETA values and INIT_TEMPs were favoured.
The tours generated using exponential decay were reasonable, but I felt that through further experimentation I could improve the results. In my second automated experimentation cycle, I focused on using the quadratic cooling schedule. I also modified my 2-opt algorithm, from simply swapping two random cities in the list to reversing the order of all the cities between these two cities. For every cityfile, the program ran once from every INIT_NODE (using my greedy algorithm to generate a reasonable initial tour), varying BETA twenty times between 0.35 and 0.55, using my newly modified 2-opt successor function. I used an INIT_TEMP of 100 and a 'cut-off' temperature of 0.000001. Then, similarly to the first round of experimentation, I ran the program a further 5 times for each BETA value, generating a random initial tour each time. The quadratic cooling schedule, along with the improved randomisation of the 2-opt function, generated even better tours. Once again, starting from an already 'okay' greedy-generated tour yielded the best results, with increasingly large BETA values favoured for the larger city files. On average, a BETA value of 0.4457 was favoured.

# Best Tours

| **No. of Cities** | **Tour Length** | **Algorithm** | **Variables** |
|---|---|---|---|
| 12 | 56 | A* | *Heuristic:* Minimum Spanning Tree |
| 17 | 1444 | A* | *Heuristic:* Minimum Spanning Tree |
| 21 | 2549 | Simulated Annealling | *Beta:* 0.39, *Schedule:* Quadratic, *Successor:* 2-Opt |
| 26 | 1473 | Simulated Annealling | *Beta:* 0.45, *Schedule:* Quadratic, *Successor:* 2-Opt |
| 42 | 1189 | Simulated Annealling | *Beta:* 0.39, *Schedule:* Quadratic, *Successor:* 2-Opt |
| 48 | 12274 | Simulated Annealling | *Beta:* 0.46, *Schedule:* Quadratic, *Successor:* 2-Opt |
| 58 | 25406 | Simulated Annealling | *Beta:* 0.5, *Schedule:* Quadratic, *Successor:* 2-Opt |
| 175 | 21594 | Simulated Annealling | *Beta:* 0.48, *Schedule:* Quadratic, *Successor:* 2-Opt |
| 180 | 1950 | A* | *Heuristic:* Minimum Spanning Tree |
| 535 | 49989 | Simulated Annealling | *Beta:* 0.45, *Schedule:* Quadratic, *Successor:* 2-Opt |