The Traveling Salesman Problem: A Two Algorithm Solution

1 Implementation Details

1.1 Database and Data structures implemented

The main class processes the distance data for the Traveling Salesman (TSP) into a square two-dimensional array/matrix for the current problem, which is then passed into the chosen algorithm. The number of rows and columns in the array is equal to the number of cities in the current problem. The Euclidian distance between cities i and j are accessed through distanceMatrix[i][j], created from the triangular version the class accepts. This does mean that data is repeated [1] but accessing the Euclidian distance between two cities is easier in comparison to storing them as an adjacency list as the algorithms will not have to comply to an order of cities when it access the matrix, especially as memory is not a concern. In addition to this, it is important to note that cities are numbered 0 to (number of cities in the problem - 1) so the algorithms also index cities like this and only when tours are outputted is 1 added to every city. This is an inconvenience but makes generating random elements especially that for weightings in the Ant Colony easier to handle. Also, both algorithms and the encapsulating main class are written in Java, so the distance matrix is passed into implementations of each algorithm and these objects store a collection of fields detailing the current progress.

1.2 Best Tour

Each algorithm keeps a record of the shortest tour for the problem and prints it to the screen. When searching ends, the best is referenced against the current shortest tour in file and replaced if an improvement. In addition, the best tour is printed to the console so if an error is encountered or the computer running it sleeps, the best tour currently found can still be saved if desired.

2 Summary of the Best Tours achieved

Number of Cities in the problem		012	017	021	026	042	048	058	175	180	535
Length of Best Tour found	*A - Simulated Annealing*	**56**	**1444**	**2549**	**1473**	1324	12530	25970	**21550**	2050	**49898**
	B - Ant Colony	**56**	**1444**	**2549**	**1473**	**1190**	**12377**	**25905**	21653	**1950**	50414

3 Algorithm A - Simulated Annealing

3.1 Implementation

The Simulated Annealing algorithm implements the analogy used in metallurgy to the TSP. It is similar to a hill climbing search, however it has the ability to accept potentially worse short-term solutions in the hope of allowing globally optimal solutions to expose themselves. It therefore uses probabilistic methods to identify a solution as opposed to using heuristics to guide the problem. Optimising the algorithm therefore seeks to encompass as vast an array of possibilities as conceivable.
In order to create an initial tour, an array of integers with length equivalent to the number of cities in the problem is created with each element equal to its index value. This array is then passed through a shuffle method, which loops through the array and at each element swaps it with another randomly chosen index, found using the random library "nextInt" method. The initial temperature is set at an arbitrary number and cooled after each iteration. This is controlled by: $T_{New} = (1 - C)T_{Current}$, where C is the cooling factor and if the new temperature is below the end value, searching ends. The annealing process then begins, entering a while loop. Two cities in the current tour are swapped and the length of this generated tour is then calculated. The probability that the generated tour is accepted is given by the exponential function:

$$P(T_C, \Delta L) = \begin{cases} 1, & \Delta L < 0 \\ e^{\left(-\frac{\Delta L}{T_C}\right)}, & \Delta L \geq 0 \end{cases}$$

$$Where: T_C = The\ Current\ Temperature, \Delta L = the\ difference\ in\ tour\ length$$

The function is coded through a series of "if" statements. If the generated tour is shorter than the original, then it is automatically accepted; otherwise the tour may still be accepted given the above probability. This critically examines the tour depending on how relatively "worse" it is compared to the current tour so that much longer tours have a lower chance of acceptance. The current temperature is also a parameter, so acceptance is more likely at the start of the algorithm when the temperature is high. This means as it cools, it accepts fewer "worse" tours and therefore allows the algorithm to use a hill climbing technique to perfect the tour before it finishes.

3.2 Implementation Issues and Testing

Implementing the algorithms foundations was fairly straightforward with no major problems experienced. Choosing optimal start temperatures and their respective cooling factors and ending temperature however became problematic. Printing current statistics to the console after each iteration allowed for the progress of the algorithm to be recorded. This made basic tracking possible for

[1] distanceMatrix[i][j] has the same value as distanceMatrix[j][i]

what current acceptance probabilities where being generated, confirming whether it was appropriate compared to the stage in its life cycle. This seeks to avoid the hill climbing strategy that evolves at lower temperatures from taking place too early or for not long enough. Equally if the temperature is too high, the algorithm is likely to accept most tours generated to start with for most of the algorithms run time, which represents wasted computation.

Using these techniques, the optimal start temperature was identified at around 350 with an ending temperature at 0.1. The cooling factor found to be appropriate is 0.0000001, with variations made to the number of digits altered to ensure run time is suitable to the tour as the hill climbing component needs to run for longer during problems with a larger number of cities.

3.3 Optimization Strategies

3.3.1 Recalculation after City Swapping

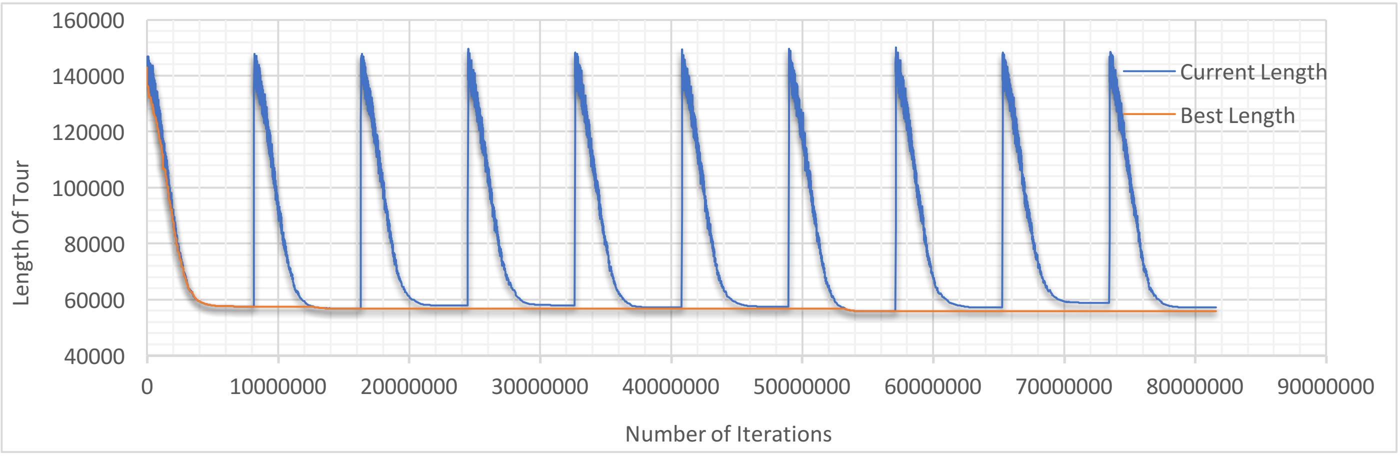
In order to avoid complete recalculation of the tour length after two cities are swapped, a new method was used which finds the length of only the components of the tour that will change instead of complete recalculation.[2] This significantly improved computation time especially for larger problems where an entire array is no longer looped through to recalculate length, therefore allowing more iterations in the same period of time and a better search. This improved the annealing run time on average by 48% for consistent temperatures and cooling factor.[3]

3.4 Experimentation

3.4.1 Restarts

The most effective solution found for effective generating of optimal tours was to recursively call annealing. By placing a "for" loop after the foundation parameters are set, the process can essentially be reset with a fresh temperature and start tour, while keeping track of the best tour that has been produced. The major drawback of the annealing algorithm is that in its first stages, the algorithms randomly produces tours and only begins proper improvements as it cools down, by which time some cities in the tour are already fixed. A solution to this is to replicate the process on a loop in order to experiment and hopefully find an optimal tour. A stopping condition was also introduced which accomplished similar success. If the length is not changed after a set number of iterations, the tour is reset and the algorithm repeats. For resetting two different techniques were tested, first the tour was completely shuffled and the temperature reset but also setting the tour equal to the best currently found and letting the algorithm optimize it was tried. The stopping value applicable depends on many factors but 500,000 was used as the default value for all problems and altered if never reached.

Figure 1: A Graph to show how the tour lengths vary over recursive recalls of the algorithm. Start Temperature: 350, End Temperature: 0.1, Cooling Factor: 0.999999. As can be seen, incremental improvements to the best tour found are made with each recursive call.



Note that significantly better results can be achieved with this technique if computation time and accuracy are not of importance.

3.4.2 Initial generated tour

Instead of just shuffling the initial tour, a "fitter" tour is established for the algorithm to start on, providing a solid foundation. The issue found with this method is even with annealing's random nature, similar start tours are established in both implementations so improvements are found during the first implementation but the algorithm has a smaller scope to experiment and so therefore produces similar results each time it runs.

3.4.2.1 Greedy implementation

A greedy first search tour is generated. A start city of 0 is set for the tour and in an array list, all the cities not yet visited. Looping through the potential cities list, the shortest distance from the current city to another is identified and the resulting city is added to the tour. This produced a solution for the annealing algorithm to run from and the time taken (or number of iterations) to reach 95% of the best tours length for the smallest 6 problems reduced on average by 18% when run three times for each. However, the

[2] For each of the cities swapped, finds the net distance between one city and the cities on each of its side. Doing this for the generated tour and the original and comparing them will give the change in tour length.
[3] A 73.51% reduction was the best found when solving the 535-city problem.

greedy search always produces the same tour so this reduces the scope of the algorithm to be run recursively, especially for larger problems. For smaller problems, too many changes are made at higher temperatures altering the tour significantly and reducing the usefulness of such an implementation.

3.4.2.2 Pass in the results from Ant Colony Algorithm for the initial tour [4]

Too much surprise such a technique is not very effective for optimising tours. The annealing algorithm was copied with small alterations made so the initial tour is accepted as a parameter and not made during the algorithm. Similar to that of the greedy implementation, so much of the initial tour is changed and optimising the start tour is not overly useful. In addition, the ants themselves (as is discussed) are very capable of identifying the best lengths for themselves. All tours created by the Colony were run through the annealing algorithm and only for problem 175 was an improvement noticed[5].

4 Algorithm B - Ant Colony

The Ant Colony was much more complex to implement due to the number of changing and interconnected components needed for it to operate. In nature ants from a colony identify food surrounding the nest by moving in random directions until they find food at which point they return to the nest with the food. While returning to the nest they leave a chemical pheromone to indicate the path to the food. Other ants will pick up this pheromone and follow it as long as the trail is strong enough to the food source to collect more food and reinforce the trail. Executing this for the TSP uses an identical concept where food is instead favourable tours and pheromones are left on edges between cities along the trails ants have taken. They use this to find the fastest route to a food source or any other landmark since as more ants use a trail the pheromones become more concentrated so future ants are that much more likely to favour that route. The algorithm seeks to gently persuade ants to experiment slowly with other routes while remembering what has worked in the past.

4.1 Implementation

The central component of this execution is the layout of pheromones. Pheromones are meant to indicate a potential desirability of certain paths over others and so are applied to each edge[6]. In the same way that the adjacency of cities is stored, a Pheromone Two-dimensional Matrix is created, with pheromoneMatrix[i][j] representing the Pheromone between cities i and j. An initial pheromone is set for all values in the matrix at conception.

A list of "Ant" objects is created and stored as an array of ants. Each Ant has a memory stored as an array of cities it has been to and is relatively intelligent in that it has knowledge of where it has not been yet [7]. This means that each Ant keeps track of where it can go next and the tour it has taken thus far, optimising the solution by avoiding recalculation. Ants are placed individually into the network with a random starting city. An ant then chooses and moves to a new city that it knows is has not visited. The new current city is then set, which then removes the city from unvisited cities and adds it to the tour. This process repeats until the ant has a complete tour. This is repeated for every ant in the array. [8]

This matrix is updated after all ants have traversed between the cites. For each ant, the length of its tour is calculated and checked against the best tour currently found. A pheromone depository is then made on each edge that the ant traversed in its tour. This depository is calculated by dividing an arbitrary deposit specified at that start by the length of the tour. Favouring of shorter tours results from this since a smaller length will mean more pheromone is deposited per edge so the tour will have a larger weighting in the future. This happens concurrently, therefore pheromones are not updated until all ants have completed a tour.

Weightings are determined to allow ants to decide the city to visit next. The probability of ant k in the list of ants moving from city i to city j is determined by the function:

$$P_k(i,j) = \frac{\rho_{ij}^{\alpha} \cdot d_{ij}^{\beta}}{\sum_{t_{ij} \epsilon N_K(i)} \rho_{ij}^{\alpha} \cdot d_{ij}^{\beta}}$$

$$\textit{where}: i = \textit{the current city of ant } k,\ j = \textit{the city to move to},\ \alpha \,\&\, \beta = \textit{arbitary parameters specifing distance and pheromone relative significance on edge weight},\ \rho = \textit{the pheromone amount on an edge},\ d = \frac{c}{\textit{relative adjacency}},\ c = \textit{arbitary parameter, set to } 1 \textit{ for most instances},\ t = \textit{trail, moving from city } i \textit{ to } j,\ N_K(i) = \textit{the set of all possible moves from city } i \textit{ to all adjacent cities not visted by ant } k$$ [9]

This is a weighted random distribution. The function incorporates the distance between cities and the amount of pheromone that exists on the edge, favouring trails with high pheromone amounts and short edges. To select a new city, the sum of all the

[4] Since the algorithm was altered for this, an additional class called AntAnnealing was created and is part of the submission.
[5] The improvement was from 21653 to 21550, a significant change but perhaps only by chance.
[6] The edge is a representation of the adjacency between two nodes, in the city analogy a road connecting them.
[7] This is rather unlike natures ants who are blind and have the only 250,000 neurons average in their brains.
[8] This was originally stored as an updated array list which was updated with each iteration and generalized for all ants. After the change was implemented to make the ants intelligent, for the 535-problem with 250 ants, computation time for a single iteration (sending ants out and then updating pheromones) was cut from 10.56 seconds to 4.87 seconds.
[9] Function adapted from Dorigo M., Gambardella L.M. (1997)

weightings is calculated and a random number smaller than this sum is generated. If the weight of a particular trail is larger than this value then that edge is accepted otherwise the weight is taken away from the random number. This is repeated until a new city edge is accepted. Using such a decision process ensures that the trail with just the largest pheromone value is not picked and the random element helps reduce the likelihood of entering a local loop. This turns out to be very difficult to avoid when using the algorithm to compute especially large tours where local maximums are reached and cannot be escaped without a complete restart.

4.2 Experimentation

4.2.1 Rate of evaporation

A common addition to an ant colony algorithm. Reducing all values in the pheromone matrix by the same factor, allows the algorithm to escape loops where only 1 or 2 edges between a city and the other network are used as a result of collecting pheromones. This is possible since the pheromones in the matrix are added to arithmetically while reduction is geometric. As a result, the algorithm is not reset but the opportunity for other tours to be generated becomes more likely since values in the pheromone matrix are brought closer together. Evaporation either takes place after every iteration by a small factor or is more aggressive but happens significantly less frequently. Random evaporation seemed to be most effective at opening opportunities for new ants and was implemented by the modulus of the current iteration count by the maximum number of iterations divided by the frequency of evaporation in this run. This makes it easy for evaporation to adapted to altered running times.

4.2.2 The Number of Ants

Does not seem to affect the quality of results in the long run if the number is balanced relatively to the other parameters, for example the frequency and weight of evaporation. The number of ants by default was set to the number of cities in the problem for ease of running, however it became easier that for medium tours, only using 10 ants at a time. Doing so made the effect of changing other parameters more apparent so experimentation became significantly easier. For the 535-city problem, using too many ants slows down computation time dramatically as when the number of cities increase by 1, the computation increases exponentially due to the extension of loops now required o run for longer. This has to therefore be adapted to the problem,

4.2.3 Relative Parameter values

4.2.3.1 Initial choices

Eliminating the pheromones completely still produces results very quickly in the small city problem, usually only sending ants out on 3 occasions in the best cases. Parameters passed in where as follows: $\alpha = 1$ and $\beta = 2$, evaporation = 0.5 with conduct approximately 1000 times in the algorithmic cycle, initial pheromone, the pheromone deposit = 1/number of cities and the ants were sent out usually 100,000 times. Adjustment took place over time but these were considered the default values. It is interesting to note that undervaluing the pheromones in comparison to the distance with the exponentials yields a more effective solution, possibly by placing less emphasis on extreme values in the pheromone matrix and allowing for greater exploration.

4.2.3.2 Pheromone Depositing

Pheromone deposit is constantly updated so the pheromones on the paths via the deposits is roughly the same as the lengths that are being dealt with as the slow paths are slowly filtered out. Trying to get weightings of distance and pheromone matrices to be similar. The update was to a percentage of the best length found, so pheromones hopefully reflect the distance ants are traveling. In theory, such a technique may improve the accuracy representation of pheromones, however when used practically, deposits became too insignificant and ants never entered local loops because minimal progress towards a solution was ever accomplished. Parameters were updated to reflect this and while the speed at which local loops were reached was reduced, the end results were indifferent. Putting an upper limit on the pheromone values in the matrix was attempted to fix this issue by allowing parameters to increase to counter the weak depositing but prevent exponential inflation of certain edges. The issue with this was finding the appropriate max pheromone was difficult as it varied by problem but doing so allowed for much experimentation

4.2.4 Random Mutation

The ants where incredibly effective at identifying optimal tours in smaller problems but as the number of cities grew, the ants became less successful. This is because the ant's pheromone trails cause the ants to be susceptible to local maximums that they are unable to escape from, arising from choosing sub-optimal paths accumulating pheromones by chance keeping the ants focused on a specific set of cities. In order to counter this, the pheromones currently in the matrix are "mutated" in order to alter the edges ants choose. The idea is similar to that of evaporation but more brutal in its approach.

Three methods of doing this were attempted. The first multiplies each value in the matrix by a random number between 0 and 1. The other calculates the mean of all values in the triangular equivalent pheromone matrix, then multiplies all values above this average by a C_1 and all those below by C_2. The mean was also replaced by the median and quartiles to see which implementation was better. The constants were set to 1.2 and 0.8 respectively in hopes of leafing out the cities that are "geographically" further away for a given city. Once pheromone concentration became a problem this was altered to 0.1, 1.5. This is in order to reduce the size of absolute extremes in the system. Using mutation was incredibly useful and is directly responsible for all the best tour through the Ant Colony algorithm, which were only found after its formation. It is not considerably better than recursive implementation of the algorithm but it achieves improvements in tour lengths at a fitter rate.

5 References

Dorigo M., Gambardella. L.M. (1997). Ant colonies for the traveling salesman problem. *Université Libre de Bruxelles*. 1 (2: Artificial Ants), p2-5.