Software Applications AI Search Assignment

James King

January 18, 2013

The Travelling Salesman Problem is an example of the NP-Complete class of problems; attempting to find the optimal solution by trying every possibility would require an obscene amount of time for all but the smallest of input sizes. This report describes two methods for finding decent solutions in a reasonable amount of time. The first is a type of stochastic gradient descent, and the second is a nature inspired algorithm that models a simple ant colony. A piece of analysis software was also produced which attempts to display a given graph in a form that is understandable to humans.

General Data Structures

Each city graph, after being loaded from the input file, is stored in a 2D integer array in such a way that the value in column i and row j is the distance between city i and city j. Representing the graph in this form means that checking the distance between two cities is a simple and fast operation, which was intended to make any search algorithms more time efficient.

Tours are represented as fixed length one dimensional integer arrays, with each value in the array being the index of the associated city. Each tour also has two numbers stored within it that record the number of cities in the tour and the total cost of the tour (the sum of the distances between each city in the tour). The value for the tour cost is initially set internally to -1. When the length of the tour is requested, if this value is -1 the cost is calculated and stored in the place of the -1, ready to be returned again without having to redundantly perform the costly calculation more than once. Additionally, any operations that could change the length of the tour will set this value back to -1, signalling that the value needs to be recalculated when next requested.

When an empty tour is first created, the values in its integer array are the indexes of each city in ascending order. When the ith city is added to the tour, the item in the array with value equal to the city's index is swapped with the value that was at the ith position. This means that for a tour of length j, all values in the array from $j + 1$ onwards are the indexes of cities that are not currently in the tour and therefore can be selected to be added to the tour. It is also possible to select the kth best city from this subset, using the end of the array as a buffer which is partially ordered until the city that is kth best using a given comparison criteria is found.

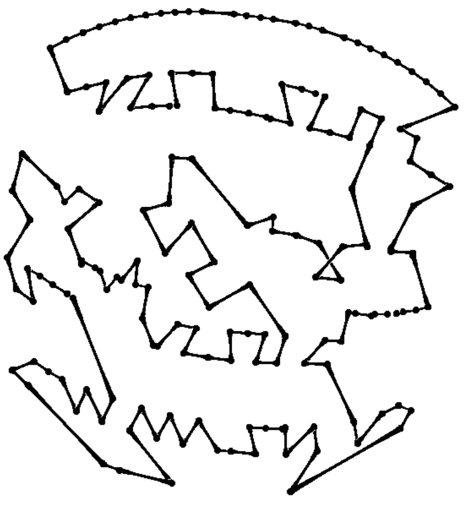
Figure 1: The 175 city graph

The graphs are given in a form that is almost impossible for a human to understand and visualise unaided, which can make it difficult to see what a search method is doing right or wrong. I wrote a simple graph visualiser that attempts to find a graphical representation of a given city file as a collection of connected points on a euclidean plane. This is achieved by attaching simulated dampened springs between each pair of cities with a strength proportional to the desired distance between the two cities. This produced far better images than I had anticipated; I was expecting all of the given graphs to have edge lengths that could in no way be realistically approximated as euclidean distances. While this is true for six of the graphs, the graphs with 12, 58, 175 and 535 cities all have fairly nice visual representations. For example, the 175 city graph appears to consist of many rows of cities with a few others dotted around them (Figure 1). The visualiser also has the ability to display a given tour, which was used extensively while improving and evaluating my algorithms.

Algorithm A

My first algorithm is a combined stochastic gradient descent and constructive search. The algorithm starts with an empty tour. For each step, a random city from the set of unvisited cities is added to the end of the tour. Next, a hill climbing algorithm operates on the tour until no new improvements are found. This process repeats, adding a new city every step, until the tour is complete. Then this tour's length is calculated and compared with the shortest found so far. If this is a new shortest tour, it is recorded and the whole algorithm starts again.

The hill climbing method used in this algorithm is based around reversing sections of the tour. To perform the reversal between two cities T_i and T_j (where T_i is the ith city in tour T), first the positions of T_i and T_j are swapped in the tour. Then i is incremented by 1, and j is decreased by 1. The swapping and incrementing / decrementing process repeats until $i \geq j$, which is when all cities between the initial two selected ones have been reversed in place.

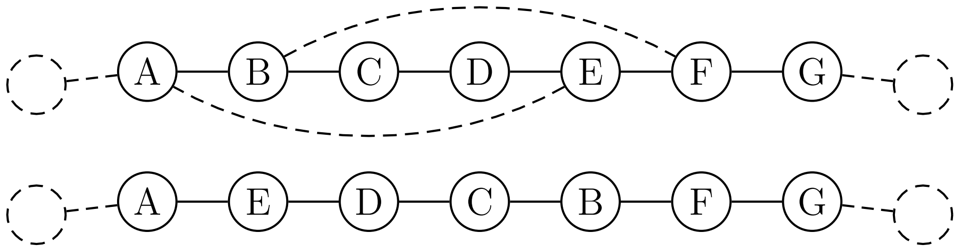

Figure 2: Segment of a tour before and after a reversal is applied between B and E, with the two new connections to be added represented by dashed arcs.

The two cities to reverse between are chosen by checking every pair of cities for the pair that would give the largest improvement to tour length if reversed. The operation to find the change in tour length is made computationally simple by the fact that only two connections will be changed in a way that could affect the tour length - the connections between the cities inside the swapped section are reversed, but their length stays the same. The change in tour length is therefore found with this simple formula with constant time complexity ($|\overrightarrow{T_iT_j}|$ is the distance from city T_i to T_j as described by the city graph given as input).

$$(|\overrightarrow{T_{i-1}T_i}| + |\overrightarrow{T_jT_{j+1}}|) - (|\overrightarrow{T_{i-1}T_j}| + |\overrightarrow{T_iT_{j+1}}|)$$

Because of the nature of the reversal, there exist sets of pairs of cities which produce the same change in tour length when reversed. Pairs that would have the same effect on tour length as others can be safely excluded to improve performance. I achieve this by picking the index i of the first city, and the number of cities to reverse c, where $c \geq 2$ and $c \leq \frac{toursize}{2}$. This excludes swapping a city with itself (which will never change the tour length) and any reversals that would return the same tour as a previous one but entirely in reverse.

The tour construction and reversal process is very fast, and because the process is independent of any other systems related to the generation process it may be easily threaded. This algorithm relies on being executed many, many times to increase the probability of finding the optimal tour, and through multithreading and other more general optimising techniques I improved the speed of the algorithm to run thousands of tour tests a second. While the algorithm may not be the most complex, by running it long enough it produced some pretty small tour sizes.

Initially the algorithm would only run the reversal improvement method after the entire random tour was generated. This produced better results less frequently and was actually slower despite running the improvement algorithm only once per tour. The speed increase when improving after every new city added is probably because the tour would require less reversal operations to find a local minimum when the tour is larger (and the reversal algorithm is more expensive) if the tour is constantly being maintained as it is built. However, I am not certain that it is possible for the absolute optimal tour to be found in every case when the tour is being improved as it is built whereas it is trivial to see that finding the optimal tour is possible when only improving after a random tour is built. I decided to trade the possibility of finding the optimal tour for finding sub optimal but decent tours more frequently.

Algorithm B

My second algorithm is a constructive search using an Ant Colony heuristic. The method places thousands of virtual ants at different starting positions on the graph, which then wander between cities they have yet to visit until they complete a tour. When a tour is complete, virtual pheromones are placed along the route that the ant took, which influence other ants to take the paths along that tour more often. The ant then forgets which cities it has visited and starts another tour.

The ants choose their next city to visit by first scoring each unvisited city C with the given criteria:

$$score = PW \times PS(C) + (1.0 - PW) \times \frac{CT + 1.0}{PL(C) + 1.0}$$

Here PW is a random number between 0.0 and 1.0, $PS(C)$ is the potency of the pheromones left on the path between the ant's current city and city C, CT is the number of tours completed by any ant which was a new shortest tour, and $PL(C)$ is the length between the ant's current city and city C.

The equation is split into two parts, the first to score based on pheromone level, and the second on path cost. The value of PW determines how much the score is weighted by pheromones, which was introduced so that some ants would select cities by looking at path cost while others would use pheromone potency.

After ranking each city in order by score, there is a random chance that the ant will skip the best city and use the second best, and an even smaller chance it will use the third best, and so on.

The amount of pheromone left when an ant completes a tour is given by:

$$pheromone = \frac{GraphCount}{TourCost - Minimum}$$

Here $GraphCount$ is the number of cities in the graph, $TourCost$ is the total length of the tour, and $Minimum$ is a constant which is the sum of the smallest cost from each city in the graph to any neighbour. The total amount of pheromone between two cities is also limited to being at most the value of CT. The subtraction of $Minimum$ was added later to make improved tour lengths have a bigger difference in pheromone potency. Since $TourCost$ can never be lower than $Minimum$, not subtracting this value would mean that an improvement in tour length would lead to only an insignificant difference in the amount of pheromone placed for larger graphs.

The equations used to determine which cities to visit were formulated using the graph visualiser I had previously written; initially the ants would tend to stop improving the route found after only a few iterations, so I modified the graph visualiser to show the paths each ant took at each step. This allowed me to repeatedly modify my path selection and pheromone placement algorithms to try and force the ants to experience more routes, as tested by viewing them with the visualiser. One such improvement was that I added a random probability for the pheromones to be cleared if no better result was found in a while. This would give the simulation the opportunity to find other tours.

Like my first algorithm, this method was easily threaded to improve speed. I also improved execution speed by recording which cities each ant had visited in its current tour in an array of boolean values so that looking up if a tour was unvisited would have a constant time cost instead of a linear one. I chose to use 24 ants per city because having much more tended to saturate the graph with pheromones too quickly.

Admittedly I didn't spend as much time adjusting this algorithm as I did with the first, mainly because the first algorithm produced very good results from the start with relatively little effort. However, I expect that the ant colony, given enough tweaking and thought, will be more effective at working towards smaller tours in an intelligent manner.

Results Analysis

The results displayed in Table 1 are the all-time lowest tour lengths achieved by each algorithm. The table also shows the difference between the results of the two algorithms, and the difference as a percentage of the shortest tour's length when the difference is greater than zero.

City File	Result A	Result B	Difference
SAfile012	56	56	0
SAfile017	1444	1444	0
SAfile021	2549	2549	0
SAfile026	1473	1473	0
SAfile042	1187	1188	1 (0.08%)
SAfile048	12166	12177	11 (0.09%)
SAfile058	25395	25395	0
SAfile175	21408	21605	197 (0.92%)
SAfile180	1950	1950	0
SAfile535	48533	49061	528 (1.09%)

Table 1: Final results of both algorithms for each given city, and the difference between them.

The first obvious thing to note is that both algorithms achieved the exact same tour length for the first four tours. Given the small size of the graphs, I would be tempted to claim that this may be because the length found is the absolute minimum. Both algorithms find their respective minimum tours in very little time, and it is hard to compare the two in that sense.

Tour lengths for the graphs of size 42 and 48 differ very little between the two methods, and the 58 city graph again has equal tour costs for both. This may have something to do with the nature of that graph, which when visualised shows that the paths have costs similar to euclidean distances, and a clean tour can be traced around the outside with only a few high density areas where the best route is ambiguous.

The next graph, with 175 cities, has a sudden jump in difference between the two methods. When visualising the best tour produced by each algorithm it is clear that the first method traces a neat route around the graph with no unnecessary paths between distant cities. The second method's visualisation, however, shows several wasteful jumps between the rows of cities.

The 180 node graph is an exceptional one. It has many paths between cities with zero cost, which are probably designed to act like "dead ends" in that a Best First Search will attempt to follow these and end up at a city with the only paths available having extremely high cost. Both of my methods seemed to avoid this issue. An interesting difference between the two algorithms is that the first took a few hours of execution over the course of many runs to arrive at its minimum, and yet the second tends to find the same result almost instantly. This could be because the ant colony searcher is able to find improved tours that require more than one mutation of the current best, whereas the gradient descent can only move to immediately beneficial tours. This means it will always choose to use the paths of zero length if possible, where a slightly more costly path would open up the opportunity to choose many shorter ones later.

The last (and largest) graph causes both algorithms to struggle. The first takes noticeably longer to test each tour, and so will get through a smaller sample in a given amount of time than with a smaller graph. However, it will still generate a better result than a direct Best First Search or the second algorithm with its first few tests. The second algorithm produces its best result in a much longer time than other graphs took. The difference in speed of finding minimal results is due in part to the increased amount of time needed to perform operations on the larger tours, and also because there are simply many more possible tours so the probability of an arbitrary tour being minimal is substantially lower. Looking at the visualisations, the first method navigates its way cleanly through the regular grid-like clusters on one side of the graph, whereas the second mostly traverses them in no real pattern.

When comparing simply by result score it is apparent that Algorithm A currently can produce better results than Algorithm B. However, there isn't really much scope for improvement with the first algorithm apart from speed optimisation and maybe eliminating redundant tour checks. With Algorithm B, however, there is plenty of opportunity to fine tune the method to attempt to get better results, including the ability to alter it with optimisations for individual graphs. Given more time, I am sure that it would be the better algorithm of the two.