The Traveling Salesman

Ludvig Jonsson
ludjon@kth.se

Jacob Håkansson
jacobhak@kth.se

2012-12-06

Abstract

We have implemented the Nearest Neighbour path generation algorithm for the traveling salesman problem. We have also implemented three different versions of the local search optimization algorithm 2-opt. We found that using sorted neighbour lists in 2-opt dramatically cut our solution's running time. This was the most important algorithm of those we implemented in order for our solution to be time-efficient.

The ID for our best submission to Kattis was: 330741

Contents

1 Background

The traveling salesman problem is the problem of finding the shortest path between a set of cities, were the salesman may only visit each city once. The traveling salesman problem is a classic optimization problem. This is because it has been proven to be NP-hard and has a lot of practical applications. In this particular instance of the problem we are dealing with cities in a euclidean plane, this is also know as the 2-dimensional traveling salesman problem. The size of the problem sets is up to 1000 cities each. We had a time constraint of 2 seconds per instance of the problem.

A standard approach to get a short path in the traveling salesman problem is to first generate a path with a naive algorithm and then optimize that path with local search optimization algorithms.

1.1 Path Generation Algorithms

1.1.1 Nearest Neighbour

The nearest neighbour algorithm generates a starting path by choosing an arbitrary starting city and then repeatedly visit the nearest city (which hasn't yet been visited) until all cities have been visited. This is generally not a bad approach, but it's obviously not optimal since the closest city isn't always the best choice in the long run.

1.1.2 Clarke-Wright

Clarke-Wrights algorithm generates a starting path by choosing an arbitrary city, which it uses as a hub. The algorithm then proceeds to visit each city, while always returning to the hub before visiting the next one. When this is finished the algorithm checks each pair of cities which doesn't include the hub for how much shorter the path would be if it bypassed the hub and went directly between the city-pair, this distance is called savings. The algorithm then goes through these savings and perform the best path switch available while maintaining the rules for the path (each city can only be connected to two other city). When the hub is connected to only two other cities the algorithm is done and we have a starting tour.

1.2 Optimization algorithms

1.2.1 2-opt

2-opt is a local search algorithm which tries to shorten the overall path by switching the paths between 2 (hence the name) city-pairs. There are a number of strategies of choosing which city-pairs to try. The simplest one is to look for paths that intersects, as eliminating these are guaranteed to shorten the overall path. There are, of course, a possibility of a lot more moves that would shorten the path and these switches are missed with this simple approach.

A more general strategy is to for each city simply follow the path and look for switches that result in a shorter path. This however is not very time efficient, as it will check all possible switches for each city. A more efficient version of this strategy is to create sorted neighbour lists for each city and only look for switches for the between cities that are close to each other. This works especially well to counteract the greediness of path generation algorithms such as nearest neighbour.

1.2.2 3-opt and k-opt

The 3-opt algorithm is similar to 2-opt, but instead of switching two edges it switches three and chooses the best combination of them (the edges resulting in the shortest total path). One 3-opt move can be seen as two or three 2-opt moves. When a tour can't be optimized any further with 3-opt ("3-optimal"), we also know that it can't be more optimized any further with 2-opt ("2-optimal").

k-opt is just a more general version of these algorithms, meaning that you switch k edges and choose the edges resulting in the shortest path.

2 Method

We used the same representation for the path and cities throughout our experiments. The cities were given an integer ID and the path were simply represented by an array of the cities' ID:s. If two cities were next to each other in the path array they had a path between them. One disadvantage of our approach to path representation is that we needed to constantly change the ordering of all the cities between two cities that we wanted to switch.

We pre-calculated the distances between all cities and stored them in a distance matrix. We traded the calculation of a euclidean distance to an array lookup, which takes less time. In return we took a memory hit and

longer set up time. The set up time wasn't a real trade-off as we would need to calculate the distances between all cities in our final 2-opt approach anyway.

We started out by implementing nearest neighbour and 2-opt with only crossing detection. For the intersection detection we used Java's Line2D class, which has built-in support for intersection detection. We then simply converted the paths between cities to lines on the fly. We then proceeded to improve our 2-opt implementation by generalizing it to compare all possible switches and choosing the switch which resulted in the shortest total path. Finally we implemented a 2-opt which used neighbour lists instead of going through all possible moves. We did this because we breached the time limit of 2 seconds.

We decided to not implement 3-opt because [1] indicated that it would need more time to be effective, and time was something we could not sacrifice. We did not experiment much with different path generating algorithms, and chose to focus on 2-opt as we expected better gains from it. We generated random test cases and visualized them to evaluate the algorithms.

3 Results

The benchmarking done was mostly based on the Kattis results, but we also measured the preprocessing time (time to read and set up the TSP instance) and running time of the different optimizations.

On our generated test cases (1000 vertices, x and y coordinates between 0 and 10^6), the average time for the initialization of the TSP instance (read input, build graph, build distance matrix and build closest neighbours-lists) was 300 ms. This is 15% of the maximum running time in the Kattis evaluation.

Our implementation of the Nearest Neighbour algorithm, our starting point, gave us a Kattis score of 3.01. The average time for the algorithm to find a path on a test case with 1000 vertices was 20 ms. This is mostly because all of the distance calculations are done in the initialization of the TSP instance.

With the intersection-based 2-opt algorithm, we got a Kattis score of 8.81 and the average running time on our test cases was 1140 ms.

The general 2-opt that was not using the closest neighbours-lists got a Kattis score of 14.91 and the average running time was 954 ms.

Our final version of 2-opt, with the closest neighbours-lists, got a Kattis score of 26.62 and the average running time on our test cases was 440 ms.

Each algorithm had a maximum running time set to 2000 ms.

Algorithm	Kattis score	Improvement over Nearest Neighbour	Time to reach shortest path possible
Nearest Neighbour	3.01	0%	20ms
Intersection-based 2-opt	8.81	292%	1140ms
General 2-opt	14.91	495%	954ms
2-opt using sorted neighbour lists	26.52	884%	440ms

The timings are based on solving our generated instances on an "ultrabook".

4 Discussion

The intersection based 2-opt gave a worse result than the others, but better than only running nearest neighbour, as expected. This was because it misses some opportunities of optimization. The running time was relatively long, this was probably due to the use of Line2D, which is normally used for graphical applications.

The general 2-opt and neighbour list 2-opt had similar results in terms of path length, but the version with neighbour list were faster, due to the fact that it had fewer choices of optimization that were ultimately the right ones.

5 Conclusions

The biggest conclusions that we could draw from implementing and testing the different algorithms was that the most important part of the local optimization algorithms was to choose carefully which and limit the number of vertices to use when trying to switch edges. Having a precomputed distance matrix and precomputed closest neighbours lists for each vertice was a big part of the final algorithm used.

If we had more time, there are things that could be improved further. The next step would obviously be to implement 3-opt and maybe even a smarter algorithm for the starting path generation. Trying another data structure for the path would probably also be a good idea, the integer array is a slow

alternative especially when the path changes a lot during the optimization of it.

Another thing that we thought about optimizing is the initiation of the TSP instance. The current version uses about 15% of the total run time for the building of the graph, distance matrix and the neighbour lists. This is something that we would look into more if we had more time.

6 Reference

1. David S. Johnson, Lyle A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization.
http://www2.research.att.com/ dsj/papers/TSPchapter.pdf