Travelling Salesperson 2D
Project 1 - D2440

Jacob Hallman, jacobhal@kth.se
Nedo Skobalj, nedo@kth.se

November 2016
Best Kattis Score: 23,266534
Kattis ID: 1478068

1 Introduction

1.1 Problem

The *Travelling Salesman Problem* (TSP) is an optimization problem for graphs. The problem involves finding the shortest path when visiting all vertices or cities in a graph exactly once, and then returning to the origin. Research regarding this is commonly used for real world problems where transportation is used. To calculate the distance between two vertices we use the *Euclidean distance* which is the length of the line segment between them. This value is rounded to the nearest integer in this problem definition. If we have two vertices $p(p_1, p_2)$ and $q(q_1, q_2)$ the two-dimensional euclidean distance can be calculated as follows:

$$Distance(p, q) = \sqrt{p(p_1, p_2)^2 + q(q_1, q_2)^2}$$

A formal description of the TSP problem can be defined like this: Find the cheapest *Hamiltonian cycle* in a given graph with weighted edges. A Hamiltonian cycle is a graph cycle through a graph that visits each vertex exactly once.

2 Algorithms

2.1 Construction algorithm

2.1.1 Nearest neighbor

The greedy algorithm that was implemented was the Nearest Neighbor algorithm. It works by comparing the distances from the last vertex in the path to all other vertices that are not part of the current path, and then adding the closest vertex to the path. Once a vertex has been added to the path it is marked as visited which means it will not be used in further distance comparisons from other vertices. The algorithm then proceeds by repeating the process for the added vertex, and then the next added vertex, and so on until all vertices are part of the path. The path is returned as the solution to the TSP where the first vertex in the path is the starting vertex. Vertices are visited in order from first to last.

```
   Data: Set of vertices $V$
   Result: List of vertices as solution to TSP
1  Initialize vector P as path and vector Vis as visited vertices, both of size
     $|V|$
2  P[0] = 0, Vis[0] = true
3  for i=1 to $|V|$ do
4      best = -1;
5      for j=0 to $|V|$ do
6          if (!Vis[j] and best = -1) or (!Vis[j] and Distance(i-1,j)
            <Distance(i-1, best) then
7              best = j;
8          end
9          P[i] = best;
10         Vis[best] = true;
11     end
12 end
```

Algorithm 1: Nearest Neighbor

The data structures used in this algorithm are just the two C++ vectors for the path and visited vertices. The reason that these were used is that they provide easy access to elements at specific indexes, their size will not need to be changed and accessing an element is done in $O(1)$ time complexity. Since we never search for an element in any vector the time complexity of search does not matter. One could use a different data structure to achieve the same result, but we do not believe there are other ones that will perform considerably better.

Nearest Neighbor was chosen as the construction algorithm because of its simple nature. It gave us a good construction algorithm to build our optimization algorithms on. We later tried replacing Nearest Neighbor with a Greedy heuristic/Shortest Edge heuristic. This would simply pick the shortest edge until all edges form a TSP route. Naturally we would have to check that the added edge does not cause a cycle and does not cause the degree of any node to go above 2. Unfortunately we ran into some problems trying to implement the algorithm which lead us to abandoning this approach because of time constraints. This heuristic should in theory have performed a bit better than the Nearest Neighbor algorithm without being much more complex. This is the reason it was our other candidate for a construction algorithm. Other construction algorithms such as Insertion Heuristics were considered but once the Shortest Edge heuristics failed we did not have time to try another third option as our construction algorithm.

2.2 Optimization algorithm

The optimization algorithm uses three algorithms to approximate a good solution to the TSP, Nearest Neighbor, 2-opt and a randomization algorithm. The optimization algorithm starts of by using the nearest neighbor algorithm to find an initial solution. 2-opt is then performed on this initial solution to find a better solution. This solution is then stored as the first best solution.

The optimization algorithm then makes uses of randomization to search for an even better solution. It randomizes a new path as the solution to the TSP problem and then applies the 2-opt algorithm to it. There is a chance that this solution is better than the Nearest Neighbor with 2-opt solution previously found, and if that is the case it will be saved as the new best solution. This process is repeated until the running time reaches around 1.75 seconds. The reason behind the time limit is to make sure the algorithm tries as many random solutions as possible in the given time limit. Doing this increases the chances of finding a better solution.

2.2.1 2-opt

The 2-opt algorithm uses local search to solve the TSP problem. This means that it goes through solutions one by one by performing local changes and picks the best one in the end. The algorithm removes two edges from the path and reconnects the paths created which can only be done in one way for the tour to be valid. This is done iteratively until no more improvements can be made.

We used mostly vectors for the 2-opt implementation. However, we used a distance matrix for all edge combinations so we did not have to recalculate them when calculating the distance of new tours found by 2-opt. Also, the distance matrix was an effective way to look up distance values when deciding whether to perform a swap or not.

The reason for picking the 2-opt algorithm as our main optimization algorithm was because of its simplicity and effectiveness. We tried using a different algorithm at first but that did not work out too well. So when we switched algorithm 2-opt was the best option considering the limited time we had to implement it and the grade we were aiming for. It is also a proven algorithm for the TSP, meaning that it has been demonstrated to work well enough as an approximation of the optimal solution.

2.2.2 Randomization

The randomization algorithm simply uses built in functions in C++ to shuffle the vector that contains the path. A new random seed is generated every time the function is called. The reason behind adding randomization is that a random path can generate a better solution than the Nearest Neighbor algorithm, when 2-opt is applied to both. This is because the Nearest Neighbor algorithm can produce a solution that is a local minimum, which means that the 2-opt algorithm can only improve the solution so much. By using randomization we can escape the potential local minimum that Nearest Neighbor produces and find a better solution.

3 Results

3.1 Greedy algorithm

Kattis ID: 1443655, Score: 3.036214 (Greedy nearest neighbor)

The greedy algorithm provided in the assignment description achieved a score of approximately 3. The reason this algorithm performs the worst of the three in terms of Kattis score is simply by design. Since it is a greedy algorithm it only looks at the best current option of vertices to add to a path. It does not consider how its choice of vertex will affect the entire solution and its future options of vertices. This can for example lead to having edges that cross each other which is not the optimal solution, and which the 2-opt algorithm tries to solve.

3.2 Optimization algorithm

Kattis ID; 1472510, Score: 12.542937 (Greedy algorithm with 2-opt)
Kattis ID: 1472869, Score: 17.388761 (Random initialization and 2-opt with limited number of iterations)
Kattis ID: 1478068, Score: 23,266534 (Greedy algorithm with 2-opt once + Random initialization and 2-opt with time limit)

The fact that the greedy algorithm with 2-opt performs better than just the greedy algorithm alone should be quite obvious. Since the 2-opt algorithm's purpose is to improve on the solution produced by the greedy algorithm it make sense that it will score higher. Especially if you consider that the greedy nature of Nearest Neighbor will most likely produce a solution with plenty of improvement possibilities(see previous section).

The random initialization and 2-opt performed a bit better than the greedy algorithm which can seem a bit unusual, but the answer lies in the limited iterations and the speed of the algorithm. The number of iterations that were performed were the maximum amount that would pass the time limit on the largest Kattis test. This means that a faster algorithm will allow for more iterations. We were able to develop an algorithm that was fast enough to allow us to get a better score than the greedy algorithm with 2-opt. Of course the score also ties into what was mentioned in Section Randomization about random solutions being able to escape the local minimum of the greedy algorithm.

The last algorithm performs the best for two reasons. Firstly, by running the greedy algorithm with 2-opt once allows us to set a kind of lower bound for our solution. We are able to guarantee that the solution that is produced will be at least as good as just doing the greedy algorithm with 2-opt once. This is good because we cannot be sure that our randomized solutions with

2-opt will produce a better solution. Secondly, the algorithm uses a time limit instead of a limited number of iterations. This enables the algorithm to perform the maximum amount of iterations within a given time limit thus increasing its chances of producing a better solution.

4 Issues faced

We did not face too many issues trying to implement these algorithms, the biggest issue and loss of time was that we tried to solve the TSP by using Minimum Spanning Trees(MST). This solution did not work out well so we switched to implementing 2-opt with randomization. The MST should have produced a good solution but we just did not have enough time to implement it since we ran into big problems. That being said, the implementation had its issues that we had to solve.

One problem with our initial 2-opt implementation was that it was too slow. The reason for this was that the algorithm would perform a swap, check if the swap generated a lower cost path and then decide to either keep it or revert back to the previous path. This resulted in a lot of unnecessary swapping which increased the running time. This was solved by adding a piece of code that checked if a swap would produce a lower cost path before the swap was performed. By doing this the algorithm only performed swaps that would produce a lower cost path thus reducing the running time.

Another issue that made the implementation slow was the way that the cost of a new tour(after swapping to edges) was calculated. This was initially done by iterating over all the vertices in the new path and adding its distances, but having to do this every time a swap was performed was not efficient. So instead a distance matrix was used, which was made up of a vector of vectors. The distances matrix is calculated once the input has been read which means that the distances between two points does not have to be recalculated multiple times. Since the distances between points was used in several places in the code, it lead to better running time.

During implementation we were trying to improve the construction algorithm by combining Nearest Neighbor with a degree of randomness. This was done by implementing a Nearest Neighbor algorithm that randomly added one of the three closest vertices to the path instead of just the closest every time. The idea was that this would combine the two good properties of Nearest Neighbor and randomization, thus creating many different good initial solutions that would produce a better solution once 2-opt was applied. But once we had implemented this we saw that it did not improve the algorithms score, so we removed it.

We ran into a few problems when attempting to implement the shortest edge heuristic mentioned earlier which we believe could have improved our solution. The main issue was detecting cycles when adding a new edge which we tried to solve in the MST approach by storing vertices in disjoint subsets.

5 Conclusion

It seems unlikely that further small improvements to construction would improve our final result when using 2-opt as our only tour improvement heuristic. However, an option for better construction is still a greedy shortest edge heuristic. If we had more time to implement it, we would have done it.

Another viable option seems to be *3-opt* or *k-opt.* This would remove 3 or k edges and reconnect them in all the ways possible to see which yields the best result. For each increase in number of edges to swap, the time will increase quite a lot as well. Since the improvement for swapping more edges is not as dramatic after a few $k : s$ it does not seem worthwhile to go too far down this route though.

The final adjustment we can think of is Lin-Kernighan's algorithm which is a variation of the *k-opt* algorithm, deciding which k is best for each iteration. This algorithm seems rather complex though, and I would imagine it is pretty hard to implement.