The Travelling Salesman Problem

KTH - Royal Institute of Technology, CSC
DD2440 - Avalg

Langesten, Daniel	Norberg, Jesper
911105-4038	900928-2196
langest@kth.se	jenor@kth.se

2013-12-06

Abstract

The Traveling Salesman Problem is a classical NP-complete problem, and has boggled the minds of programmers since the days of yore. This report presents our approach to approximating a solution to the Traveling Salesman Problem as close to the optimal solution as possible given a time constraint of 2 seconds. Our final program implemented a Greedy Initial Solver, 2-opt, 2.5-opt and 3-opt.

In order to evaluate our program we have performance checked the various versions created, followed by analyzing the data. Finally we discuss possible improvements, in particular the implementation of the Multifragmented-heuristic.

Contents

1 Introduction

An overview of the problem at hand, as well as brief descriptions of the algorithms used in the implementations of the program.

1.1 Problem

The problem presented is to implement an algorithm that finds an approximation of the optimal solution for the TSP (The Traveling Salesman Problem) in the plane. The algorithm needs to be implemented from scratch and to be written in C/C++ or Java. The algorithm will have 2 seconds of running time per graph, in which it needs to approximate a path as close to the optimal path as possible.

The scoring as described by Kattis[1]: "Let Opt be the length of an optimal tour, Val be the length of the tour found by your solution, and Naive be the length of the tour found by a basic greedy algorithm. Define $x = \frac{Val-Opt}{Naive-Opt}$. (In the special case that $Naive = Opt$, define $x = 0$ if $Val = Opt$, and $x = \infty$ if $Val > Opt$.)"

The score on a test case is 0.02^x, and there are 50 test cases in total. As x has a minimum value of 0 (when $Val = Opt$), this leads to the maximum score on a case being $0.02^0 = 1$, implying that the maximum score for the entire problem is 50 points.

1.2 Background

The TSP problem in the plane can be defined as: Given a list of cities, find the shortest path that visits every city exactly once and returns to the origin city. This problem is proved to be NP-complete[2] and is therefore in need of a good heuristic, rather than a brute-forced solution.

1.3 Approach

In order to complete the task presented in the project specification we initially needed to decide on which algorithms to use and in which programming language to implement them. The original algorithms of choice were greedy best first and 2-opt, followed by Simulated annealing. The reason for choosing these algorithms is that they are when put together an exceptionally good approximation, which got one of the top scores in a tournament.[3] We later realised that Simulated annealing is far too slow an algorithm for our time frame, and such replaced it with 2.5-opt and 3-opt.

The programming language we initially decided upon was Java. This was because we both felt that Java was the language we had the most experience in, and therefore the language in which it would be easiest to implement the algorithms. We soon changed to C however (right after implementing 2-opt), as we from the previous project became aware just how great the difference in score can be between the languages using the exact same algorithms. Indeed, we immediately got a better score, and as such kept on coding in C.

2 Algorithms

In this section we discuss different algorithms of interest. Each is implemented in the final version of our program.

2.1 Greedy solver

The greedy algorithm is quite simple. The idea is that a short path consists of cities close to each other. Hence the algorithm starts with a city and then connects it with the closest neighbouring city. Then it repeats this for the new city connected until no cities remain. It has then created a complete path, including all the cities. This path is however not even remotely as good as the one resulting from more complex graph creation algorithms. Yet, it does provide a good starting point, and is very quick. Provided by Kattis.[1]

2.2 2-opt

2-opt is an algorithm used to improve a path in a TSP-problem. The idea is to take a path that crosses itself and reorder it so it does not. This will result in a shorter path. To identify such a case in code we will need to check if we can get a shorter path by switching the ends of two edges with each other. If we do get a shorter path we have an improvement. As the time limit will be hard to beat, we had to implement a maximum vicinity for the comparisons. Each edge will only be compared with the maxVicinity specified when calling the algorithm. Next follows Figure 1 illustrating the rearrangement in 2-opt followed by pseudo-code for our implementation of 2-opt, inspired by Wizche.[3]

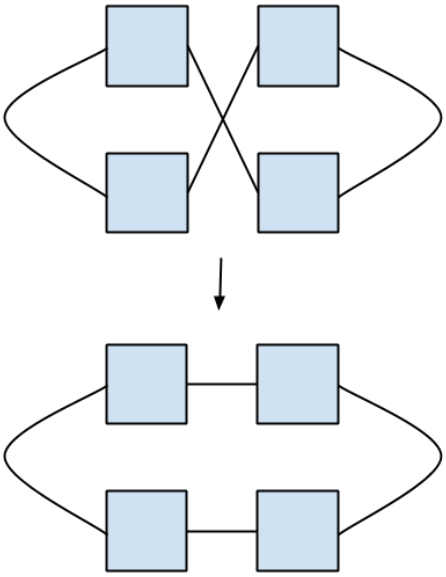

Figure 1: A graph describing how the rearrangement in 2-opt is performed.

Algorithm 1 2-Opt

```
1: procedure twoOpt(Graph, maxVicinity)
2:     while no improvement is made do
3:         loop:
4:         bestDistance = calculateDistance(existingPath)
5:         for i = 0; i < number of cities eligible to be swapped - 1; i++ do
6:             for k = i +1; k < maxVicinity -1; i++ do
7:                 newPath = twoOptSwap(existingPath, i, k)
8:                 newDistance = calculateDistance(newPath)
9:                 if newDistance < bestDistance then
10:                    existingPath = newPath
11:                    goto loop
12:                end if
13:            end for
14:        end for
15:    end while
16: end procedure
17:
18: procedure twoOptSwap(path, i, k)
19:    newPath                                          ▷ A new empty path
20:    Add path[0] to path[i-1] and add in order to newPath
21:    take path[i] to path[k] and add them in reverse order to newPath
22:    take path[k+1] to path[path.length-1] and add them in order to newPath
23:    return newPath
24: end procedure
```

2.3 2.5-opt

The 2.5-opt is an algorithm that improves the path by changing the order in the path of one city. This is done by checking if we have a shorter path if we remove one city from the path and insert it in an other location of the path. If an improvement is made, we have successfully performed a 2.5-opt. In Figure 2 that follows it is graphically illustrated how 2.5-opt is performed.

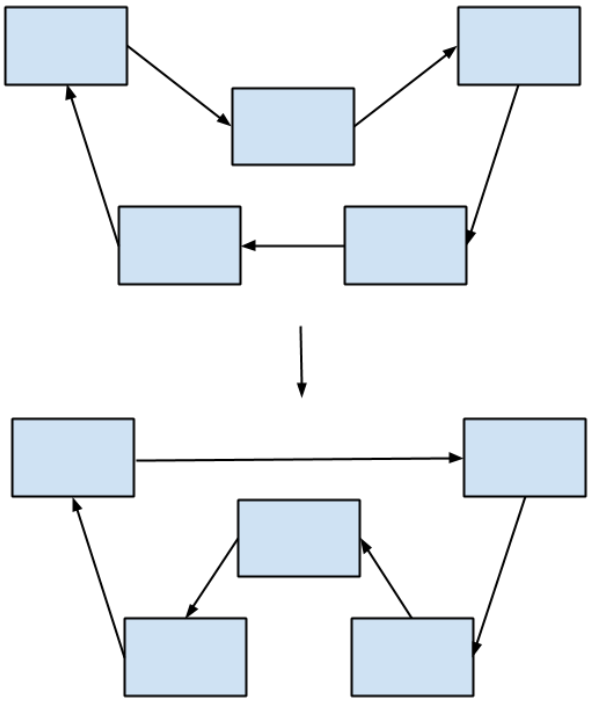

Figure 2: A graph describing how the rearrangement in 2.5-opt is performed.

2.4 3-opt

The 3-opt is an algorithm similar to the 2-opt. The main difference is that the 3-opt does not find two edges that cross. Instead it finds three edges that crosses and rearranges them so that they do not cross anymore. The result is just as in two opt a shorter path through the cities.

To perform 3-opt we iterate through all cities in a given path until three edges that crosses each other is found. Lets denote the cities in the start of each of the crossing edges as a, b and c and the cities following them as a.next, b.next and c.next. What we now want to do is to set the city after a to b.next, the city after b to c.next and lastly the city after c to a.next. When this is done we have reduced the total distance in the path. An illustration of this rearrangement follows in Figure 3. To find a three-edge crossing practically we just compare distances to see if we can optimize the path by performing the order exchange. Below we present the pseudo-code for our implementation of the 3-opt.

Algorithm 2 3-Opt

```
1: procedure threeOpt(numOfNeigh, maxVicinity)
2:     if number of cities in graph < numOfNeigh then
3:         numOfNeighs = number of cities in graph-1
4:     end if
5:     while 0 < maxVicinity do
6:         a = city[0]
7:         while we have not moved through all cities in path do
8:             for it1 = 0; it1 < numOfNeighs; it1++ do
9:                 x = a.neighbour[it1]
10:                for it2 = 0; it2 < numOfNeighs do
11:                    y = a.neighbour[it2]
12:                    if x==y then
13:                        Continue
14:                    end if
15:                    b = city[x]
16:                    c = city[y]
17:                    temp = a.prev                    ▷ The city visited before a
18:                    while (temp = temp.next) != b do
19:                        if temp == c then
20:                            c = b
21:                            b = temp
22:                            break
23:                        end if
24:                    end while
25:                    dist1 = dist(a, a.next) + dist(b, b.next) + dist(c, c.next)
26:                    dist2 = dist(a, b.next) + dist(b, c.next) + dist(c, a.next)
27:                    if dist2 < dist1 then
28:                        anext = a.next
29:                        bnext = b.next
30:                        cnext = c.next
31:                        a.next = cnext
32:                        bnext.prev = a
33:                        b.next = cnext
34:                        cnext.prev = b
35:                        c.next = anext
36:                        anext.prev = c
37:                        break
38:                    end if
39:                end for
40:            end for
41:            a = a.next
42:        end while
43:    end while
44: end procedure
```

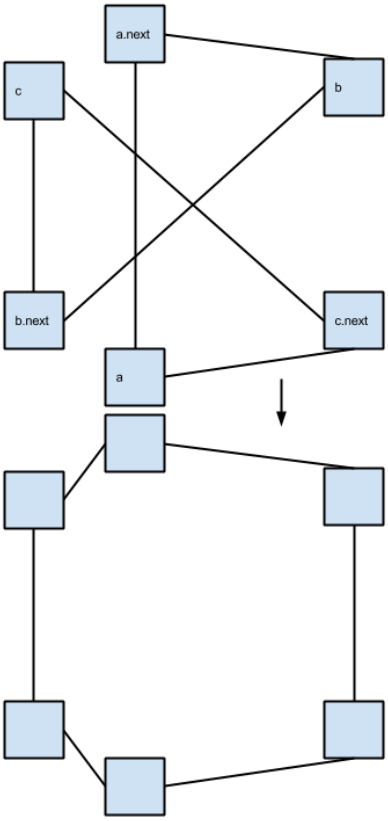

Figure 3: A graph describing how the rearrangement in 3-opt is performed.

2.5 Final program

Here is the pseudo-code for the final program. Our twoOpt also utilizes twoFiveOpt to reach as good result as possible. The x_i and y_i values represent the respective maxVicinity's, and will affect the score, as presented in section 3.

Algorithm 3 Approximate a solution for TSP

```
1: procedure TSP(Graph)
2:     PreCalculateDistances(Graph)
3:     solution := greedySolver()
4:     if numberOfCities0< 200 then
5:         twoOpt(x_0)
6:         threeOpt(y_0)
7:     else if numberOfCities < 400 then
8:         twoOpt(x_1)
9:         threeOpt(y_1)
10:    else if numberOfCities < 600 then
11:        twoOpt(x_2)
12:        threeOpt(y_2)
13:    else if numberOfCities < 800 then
14:        twoOpt(x_3)
15:        threeOpt(y_3)
16:    else
17:        twoOpt(x_4)
18:        threeOpt(y_4)
19:    end if
20:    return solution
21: end procedure
```

3 Results

Here the results of the different program implementations are presented.

3.1 Java with 2-Opt

Our java implementation only made use of the 2-Opt algorithm and gave us a score of 18.978096 on Kattis.

3.2 C with 2-opt

This implementation essentially is the same as the first only rewritten in C. Its best score on Kattis was 19.452694. Figure 4 that follows, shows how well it performed on Kattis with different number of maxVicinities.

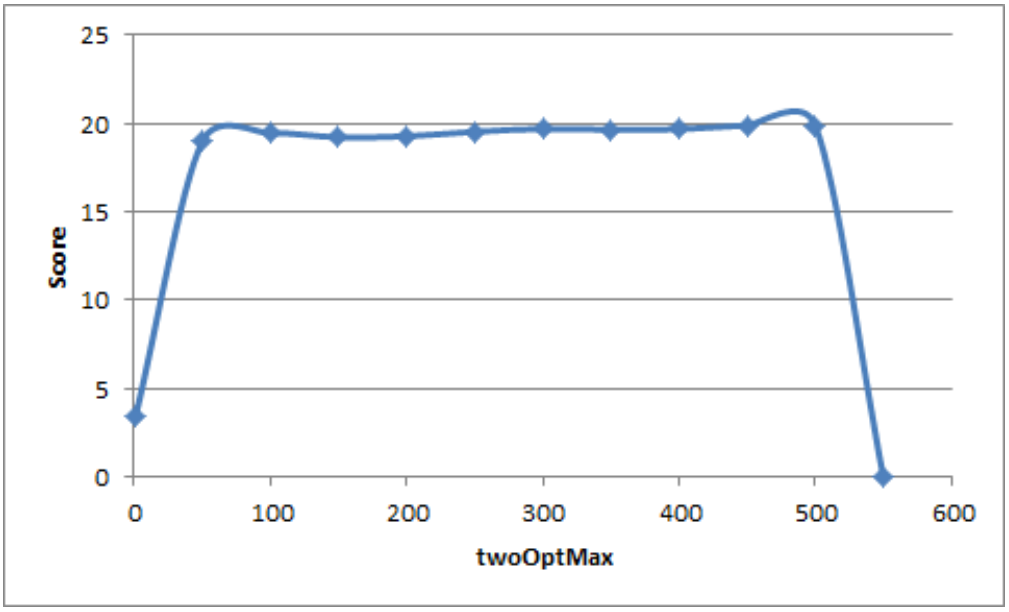

Figure 4: A graph describing how well C with 2-opt performed on Kattis with different number of 2-opt maxVicinities. A zero value means that it exceeded its time limit.

3.3 C with 2-opt and 2.5-opt

This version is the C with 2-opt enhanced with 2.5-opt and its best score on Kattis was 26.712371. Figure 5 that follows, shows how well it performed on Kattis with different number of maxVicinities.

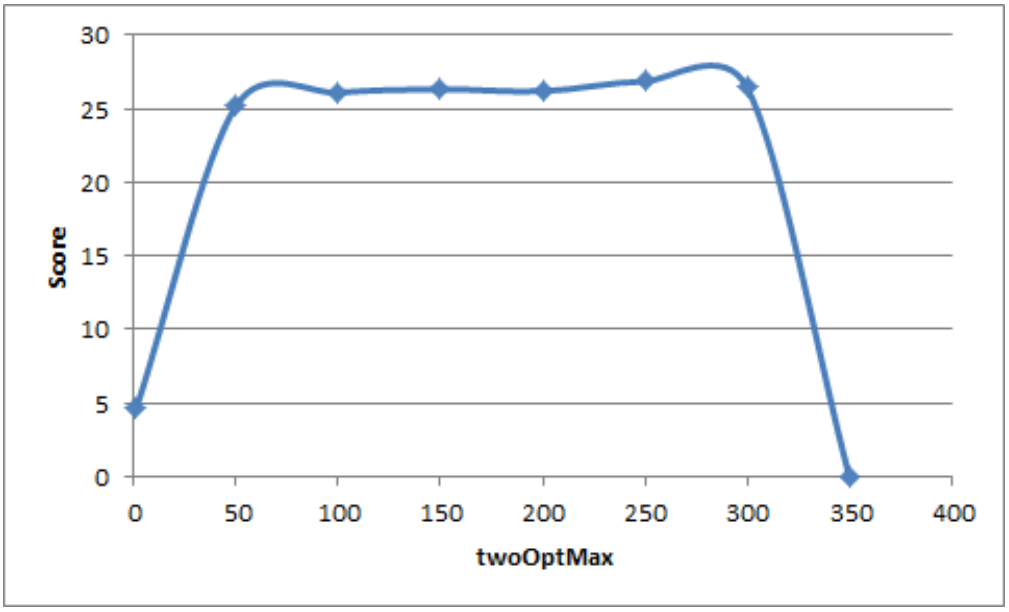

Figure 5: A graph describing how well C with 2-opt and 2.5-opt performed on Kattis with different number of 2-opt maxVicinities. A zero value means that it exceeded its time limit.

3.4 C with 2-opt, 2.5-opt and 3-opt

This is our final implementation which utilizes 2-opt, 2.5-opt and 3-opt. After tweaking the number of maxVicinities for each of the three algorithms we managed to reach a score of 29.430724 on Kattis. To make the algorithm as efficient as possible, we handled graphs based on the number of cities in them. Depending on the number of cities, different maxVicinities were used for each algorithm. We then optimized each Figure 6 describes how well this program performed on Kattis based on the graphs with less than 200 cities, Figure 7 on those with less than 400, Figure 8 on those with less than 600, Figure 9 on those with less than 800 and figure 10 on the remaining less or equal to 1000.

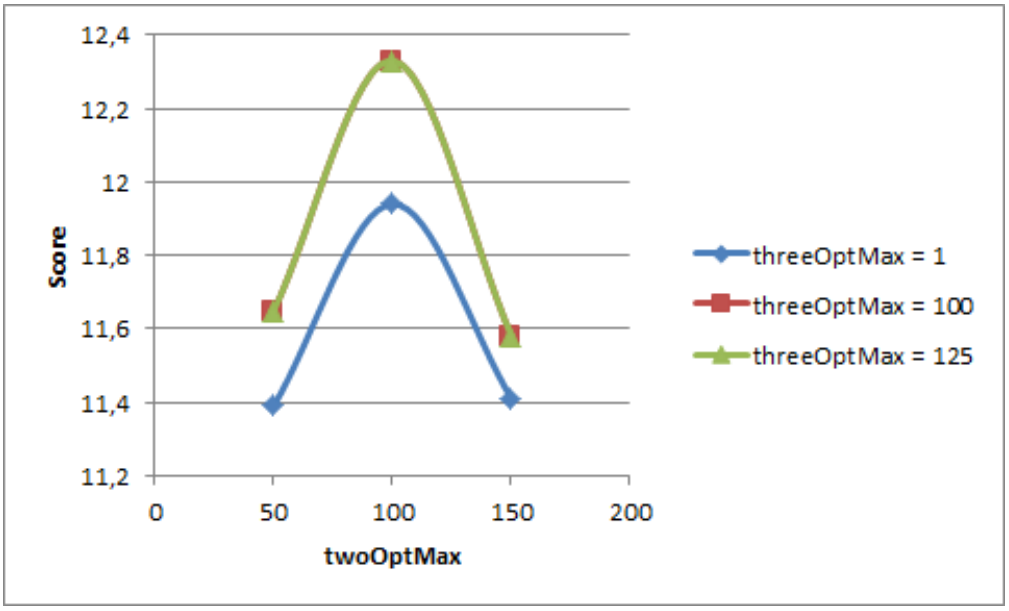

Figure 6: A graph describing how well the final implementation performed on Kattis while tweaking the maxVicinities of 2-opt and 3-opt for graphs with less than 200 cities.

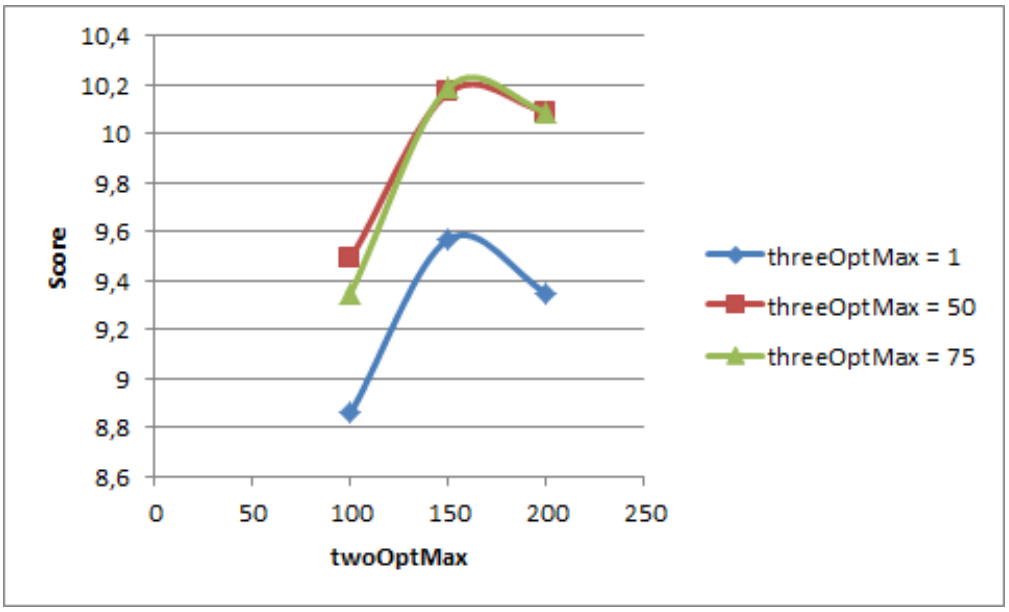

Figure 7: A graph describing how well the final implementation performed on Kattis while tweaking the maxVicinities of 2-opt and 3-opt for graphs with less than 400 cities.

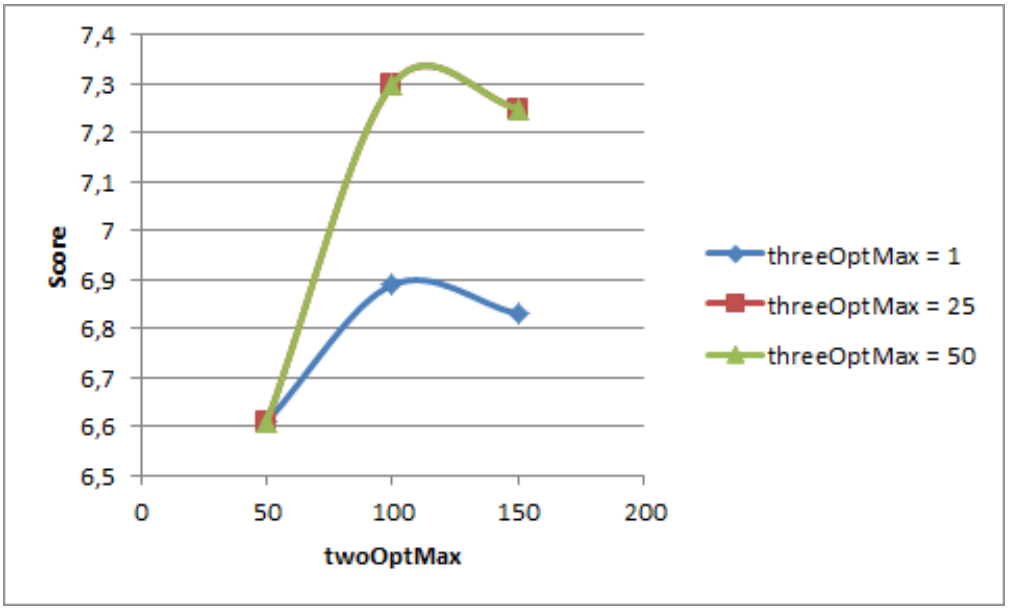

Figure 8: A graph describing how well the final implementation performed on Kattis while tweaking the maxVicinities of 2-opt and 3-opt for graphs with less than 600 cities.

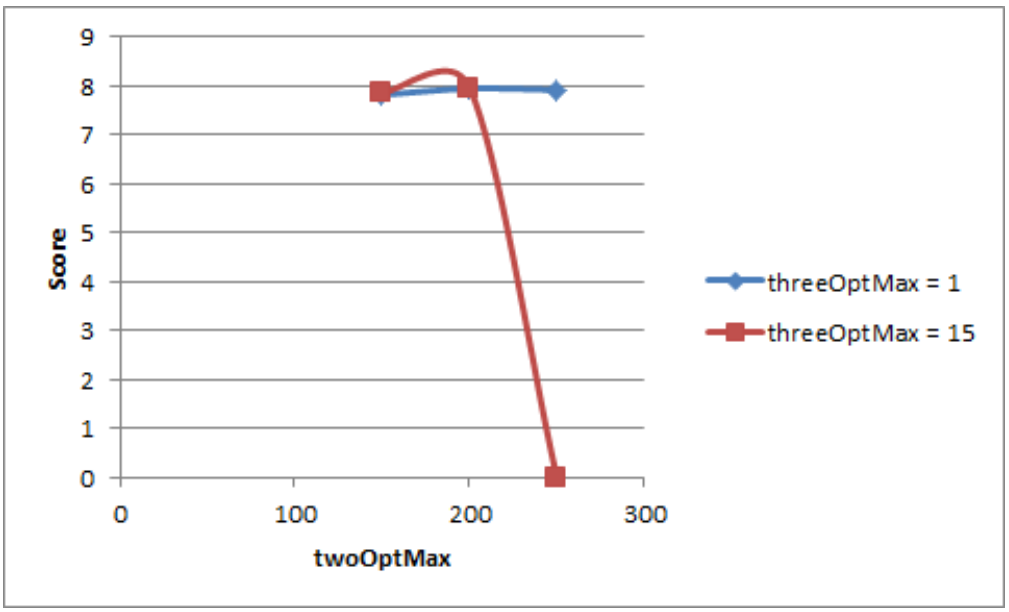

Figure 9: A graph describing how well the final implementation performed on Kattis while tweaking the maxVicinities of 2-opt and 3-opt for graphs with less than 800 cities.

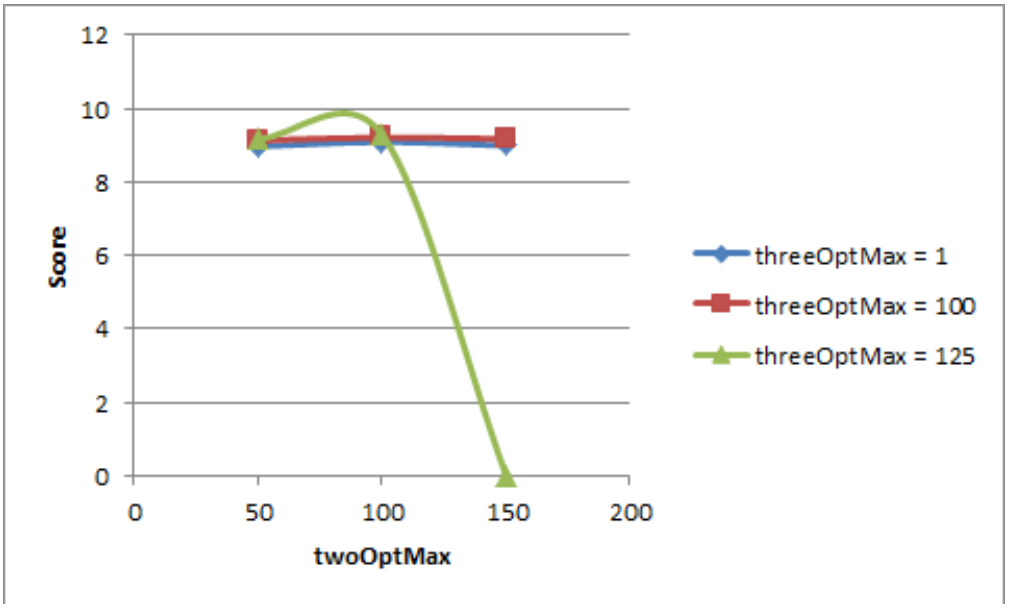

Figure 10: A graph describing how well the final implementation performed on Kattis while tweaking the maxVicinities of 2-opt and 3-opt for graphs with less or equal to 1000 cities.

3.5 C with 2-opt, 2.5-opt and 3-opt

4 Discussion

In this section we will use our results presented in 3 Results to discuss which parts of the project were successful, as well as which areas could be improved in order to heighten the performance of the program.

4.1 Language

As earlier stated in our approach we would initially use Java as development language. The results we got from it were however not as good as desired. We deduced that this was most likely a performance issue since TSP is such a hard problem. Hence we decided to re-write the code in a more efficient language; C. Indeed, as shown in the results we immediately got a better score when using C instead of Java, using the same algorithms.

A likely reason as to why Java was worse than C would be that the Java implementation ran out of time with less job done than the C implementation. The most probable reason for this is that the Java program could not run as many 2-opt operations as the C implementation before receiving a "Time Limit Exceeded" status from Kattis.

This would also explain why C is better than Java. It solves the problem faster and because of this it has time to do more improvements to the path. And the best of the C implementations is the one which utilizes all three of the implemented algorithms.

4.2 Results

The results for 3.1, 3.2 and 3.3 are rather self explanatory. The score in 3.2 is higher than in 3.1 due to the languages used, as described in 4.1. The score in 3.3 is higher as we have implemented 2.5-opt, which enables us to optimize routes even further. In all three cases the 2-opt seems to behave like a logarithmic function with regards to its maxVicinity, with a 0 in score if it gets a time-out. The optimal value for the maxVicinity is therefore the one which is closest to getting a timeout.

With 3.4 however, things get complicated. It seems that when performing both 2-opt and 3-opt, the 2-opt behaves like a quadratic function with a local maximum, with regards to its maxVicinity. Increasing the value of the 3-opt local maximum seems like a logarithmic function with regards to its maxVicinity. We therefore first found the local maximum for the 2-opt maxVicinity, and then pushed the 3-opt maxVicinity as far as we could without getting a time-out. We did this separately for each of the 5 cases we had split the problem into.

Our best implementation scored 29.430724 on Kattis. The submission id is 470359.

4.3 What are we missing?

We have implemented the core optimisation algorithms. However, we are not creating the graph in an optimal way. For this, a Multifragment-heuristic would be our first choice, as the Multifragment-heuristic is especially good for Euclidian graphs [4], and is as such ideal for our problem. Our graphs are Euclidian, as they fulfill the following two properties:

1: The triangle inequality; $d[a, b] \leq d[a, c] + d[c, b]$ for any triple of cities a, b, c

2: Symmetry; $d[a, b] = d[b, a]$ for any pair of cities a, b

5 Conclusion

Here we present our conclusions regarding how well we fulfilled the project objectives.

We are satisfied with our end score of 29.430724 points, although given the opportunity, we would have liked to implement a Multifragment-heuristic solver, as described in 4.3.

Bibliography

[1] Kattis, https://kth.kattis.scrool.se/problems/oldkattis:tsp, 2013-12-06

[2] Algorithmic Graph Theory p.219, Alan Gibbons, 1985

[3] Wizche, http://blog.wizche.ch/2010/07/tsp-problem-part-2-local-optimization.html, 2013-12-06

[4] Marist College - Algorithm Analysis and Design, http://www.academic.marist.edu/ jzbv/algorithms/approximatetspalgorithms.pdf, 2013-12-06