Solving the Travelling Salesman Problem

Niklas Bäckström, Philip Sköld, Jonas Daniels

May 12, 2014

ABSTRACT
Few problems in Computer Science has been given as much attention as the Traveling Salesman Problem (TSP). In this paper, several methods for solving TSP are presented, aiming to give an overview of the problem and how it can be solved. The paper is part of the course DD2440 - Advanced Algorithms at the Royal Institute of Technology (KTH), Stockholm, and evaluation of the implementation is made by the Kattis System (KATTIS).

CONTENTS

1 INTRODUCTION

TSP is a renowned problem within Computer Science (CS). A travelling salesman, given a list of cities, wants to find the shortest tour visiting each city exactly once and finally returning to the first city. In graph theory, the problem statement can be expressed as follows:
Given a weighted graph, $G(V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ is the set of cities and $E = \{(p, q) : r, s \in V\}$ and each edge (a, b) has a cost d_{ab}, find a Hamiltonian path such that the total edge-cost, $D = \sum_{(ab)\in E} d_{ab}$, is minimised.

This paper aims to give a comprehensive overview of the TSP problem and, several algorithms for solving TSP are implemented, analysed and discussed. Common data structures, optimisations and other solving methods are also presented, to get a broader picture of what approaches to solving TSP exist today.

1.1 KATTIS

This paper is part of a course in advanced algorithms held at the Royal Institute of Technology, Stockholm. KATTIS[1] is an online judge and is the formal judging system used. The specific TSP problem that KATTIS provides is an Euclidean TSP with $1 \leq N \leq 1000$ cities, a (Euclidean) distance metric rounded to the nearest integer and a 10^6 bound on the absolute value of the city coordinates. The time limit is 2 seconds per instance and the memory limit is 64 megabytes.

1.2 Euclidean TSP

Whereas the general TSP has no restrictions on the distance metric d_{pq}, the distance metric of the Euclidean TSP is the Euclidean distance between two city points p_1 and p_2. The property that $d_{ab} = d_{ba}$ is referred to as symmetrical TSP. Also notable is that the Euclidean distances between cities satisfies the triangle inequality: $d_{ab} \leq d_{ac} + d_{cb}$.

This project does not make any considerations for any TSP variations other than the symmetric Euclidean TSP as specified by KATTIS. Some results may be applicable for other variations of TSP but that is outside the scope of this project. So, when TSP is referred to in this report, it generally refers to Euclidean TSP.

1.3 Computational Complexity and Academic Interest

It has been shown that the Hamiltonian Circuit Problem is Non-deterministic Polynomial-time complete (NP-complete)[2] and by reduction to TSP that TSP is Non-deterministic Polynomial-time hard (NP-hard)[3]. The complexity despite its simple structure has drawn much attention to TSP and it has gotten a lot of attention within Computer Science.

1.4 Basic Notation & Conventions

Throughout this paper the following definitions and conventions are used, unless stated otherwise.

N is the number of cities for a given test case.

TSPcase, or C, is a TSP test case, which states the position of the N cities the travelling salesman must visit. N is referred to as the *size* of the TSPcase.

TSPcaseData, or D, is a collection of case data that is pre-calculated for a certain C. It consists of cities' pair-wise distances values as well as lookup tables for various objects.

d_{ab} denote the distance from city a to city b.

$a \sim b$ denote that two cities, a and b are adjacent in some tour, T, and that b comes after a i.e there is an edge from a to b. $a \sim b$ is also implicitly used to denote d_{ab}.

TSPtour, or T, is a solution to a TSPcase. A tour represents the order in which the travelling salesman visits all the cities of the corresponding TSPcase C.
This paper presents many approaches to solving TSP. It will be specified which algorithms and optimisations were implemented within the scope of this project.

2 METHOD

2.1 Data Structure

2.1.1 Pre-calculations

An essential part of every TSP solver is looking at distances between cities, because it needs to swap roads that are longer for roads that are shorter. In this project there will only be at most 1000 cities, which means there are only at most $\binom{1000}{2} = 499500$ different pair-wise distances to compute. With a total of 2 seconds of CPU time available, pre-calculating all of these distances is relatively cheap. Subsequent city distance lookups use the precomputed values which reduces total computational cost.

A benefit of pre-calculating all pair-wise distances is that it, at the same time and with little extra cost, allows the program to build neighbourhoods for all cities. So, for each city a there will be a pass that finds the distance to all other cities b. To create an adjacency list Adj for a of size K (for some $K \leq N$) the program maintains a max-heap of size at most K, holding possible neighbours sorted in descending order by their distances from a. The first K neighbours of a are added to Adj (regardless of distance). Then, each subsequent b is compared to the top element of Adj. If it is closer to a than the top element, it replaces the top element and becomes one of the new K closest neighbours. When all b have been considered, Adj will contain the K cities closest to a. Reading the city indices out of the heap into an array gives a sorted adjacency list.

With the time complexity of the original pre-computation pass being $\mathcal{O}(n^2)$, adding a heap operation to each step increases that complexity to $\mathcal{O}(n^2 log(K))$. This is not a problem, especially if K is small.

2.1.2 Tour Structure

The most important data structure within TSP is TSPtour. There are many ways to store a tour, with some being advantageous only for very large test cases. For example, data structures based on splay trees are currently seen as the most efficient choice[**online:fredman**] in the theoretical sense.

However, achieving good *asymptotic* time complexity is not the goal of this project. In KATTIS, N is at most 1000 and for such small cases it becomes apparent[**online:fredman**] that sophisticated tree-structures have too much overhead. An array of city indices is a better choice for these small cases, and it is also much easier to implement.

In this project, the amount of memory a tour occupies is not so important either, because it can be assumed that plenty of memory is available. CPU time is the limiting factor, so a desirable characteristic of the tour data structure is that common operations can be performed with great speed. Two common operations are reversing a fragment of the tour and moving a city from one place in the tour to another. These two operations have trivial $\mathcal{O}(n)$ time implementations (when TSPtour is an array of city indices), and it so happens that they are fast enough for the small test cases encountered in this project (see 3.2).

2.1.3 Square Root Data Structure

If $\mathcal{O}(n)$ would turn out to be too expensive, a two-level tree structure can be used instead of a flat array. In such a tree, there are $\sqrt{N}$ buckets, each bucket holding roughly $\sqrt{N}$ cities. The naive implementation for *reverse* becomes cheaper ($\mathcal{O}(\sqrt{N})$ time) using such a structure, because an entire bucket can be reversed in $\mathcal{O}(1)$ time by simply flipping a traversal order bit. So, when reversing a sub-tour, $\mathcal{O}(\sqrt{N})$ buckets are reversed (in $\mathcal{O}(\sqrt{N})$ time) and $\mathcal{O}(\sqrt{N})$ cities at the beginning and end of the sub-tour are reversed city per city (also in $\mathcal{O}(\sqrt{N})$ time). If the tree becomes unbalanced, it can be rebalanced in $\mathcal{O}(N)$ time, and hopefully it will not be necessary to do so many times. The *relocate* operation can be implemented trivially in $\mathcal{O}(N)$ time as before.

2.2 Exact Solutions

Although the TSP complexity renders it very impractical, there are algorithms that solve TSP exactly; for very small C they may be applicable.

2.2.1 Brute Force

It is possible to systematically try all possible tours for a given C, that is *all permutations of cities*. Even though a tour can start at any city and in any direction the brute force algorithm has a complexity of $(\frac{(N-1)}{2})! = \mathcal{O}(N!)$ tours.

2.2.2 Dynamic Programming

In this approach, consider that the tour starts at city c_1. The minimum cost of the cycle from c_1 to some other city c_i (visiting each city exactly once and then back to c_1) can be expressed as $cost_{1i} + d_{i1}$ where $cost_1 i$ is the cost of the shortest path from c_1 to c_i.

$cost_{1i}$ can be calculated with dynamic programming by considering all subsets $S \in C$ where $c_1 \in S$. Let $D(S, c_i)$ be the shortest path from c_1 to c_i visiting each city in S exactly once. If $|S| = 2$, then $D(S, c_i) = d_{1i}$. If $|S| > 2$ then the $D(S, c_i)$ is specified by the recurrence $min(D(S - \{c_i\}, j) + d_{ji}) : c_j \in S, j \neq i, j \neq 1$.[4]

The number of sub-problems are $\mathcal{O}(n2^n)$ and each sub-problem takes linear time, which results in a time proportional to $\mathcal{O}(n * n2^2) = \mathcal{O}(n^2 2^n)$.

Although better than the brute-force approach, this algorithm quickly becomes impractical and cannot be used for slightly larger problem instances.

2.3 Initial Tours

As mentioned in section 1.3, TSP is NP-hard and thus cannot be solved in polynomial time. For this reason, the reasonable way to solve TSP is to construct suboptimal algorithms (often heuristics) for finding *initial tours*. Then, to find an even better solution, local optimisations are performed after having constructed one or several initial tours. Local optimisations are discussed later in section 2.4.

In this section, a number of different algorithms for constructing initial tours are presented.

2.3.1 Random Tour

The first and simplest initial tour algorithm is the random tour. It works by shuffling all cities in C. The procedure is fast, easy to implement and can lead to good results in many cases.

2.3.2 Nearest Neighbour (NN) Heuristic

The NN heuristic is a naive and greedy algorithm that works by repeatedly adding the city closest to the city that was added to the tour most recently, i.e the nearest neighbour. Of course, the algorithm only considers cities that haven't already been added to the tour. When all cities have been added, it may be so that that the distance between the first and the last city is really large. That is an inherent problem of this heuristic (and several others).

2.3.3 Double-Ended Nearest Neighbour (DENN) Heuristic

The DENN heuristic is very similar to the NN heuristic. The only difference is that instead of only looking for the nearest neighbour of the most recently added city, it looks for the nearest neighbour to the other tail of the tour so far as well. Essentially, the tour is constructed as a single segment that grows in both directions (in whichever direction is appropriate at the time). As with the NN heuristic, the last road goes straight between the two tail ends of the constructed tour segment to form a complete cycle.

2.3.4 Multiple Fragment (MF) Heuristic

Now consider how the previously mentioned heuristics can be extended. Instead of incrementally building a single tour segment (in one or two directions), is it possible to greedily construct multiple segments (fragments) at the same time and combine them in the end? The MF heuristic suggested by Bentley [5] does exactly this. It is reminiscent of Kruskal's

algorithm for finding a minimum spanning tree, in that it looks for the shortest road on a global level. An added constraint is that it will only add roads that don't form a cycle in the tour. And of course it can not add roads that result in cities of degree larger than two (the salesman can only traverse each city once). When the $N-1$ globally shortest roads (satisfying the constraints) have been added, a single road linking the two tail cities is added as usual. Bentley [5] tried several initial tour heuristics and found that the MF heuristic was the top contender for solving cases where $N \leq 1000$.

The implementation of the MF heuristic in this project is largely based on the pseudo code suggested by Bentley [5]. The result of the algorithm is a set of edges, so it is necessary to transform it into the appropriate tour structure (see 2.1.2). Fortunately, doing so is trivial. First make a note of the two neighbours for each city. Then add two cities that are bound together by an edge (chosen arbitrarily). Finally, pick either city and (looking at the noted neighbours of that city) add the neighbour that has not been added yet. Repeat this until all cities are in the tour city indices array.

2.3.5 Christofides Algorithm

Christofides algorithm works by building a Minimum Spanning Tree (MST), T, over C and a perfect matching, M, over the subset consisting of odd vertices of C. Then compute a Eulerian path over the multi-graph {M, T}. Finally, a Hamiltonian path by shortcutting (skipping visited nodes) the Eulerian path.

This is an approximation algorithm (not a heuristic) and has an approximation ratio of $\frac{3}{2}$ and a running time of $\mathcal{O}(N^3)$[6].

2.4 Local Optimisations

After finding an initial tour, local optimisations can be made to further improve the tour. While local optimisation, quite efficiently improves the solution, there is no guarantee that the optimisation approaches a global minimum rather than a local minimum. There are other algorithms that utilise local optimisations but increases the chance of approaching a global minimum instead. These will be discussed in section 2.5.

2.4.1 2-opt

The 2-opt local optimisation works by finding two pairs of adjacent cities, $a \sim b$ and $c \sim d$, in the existing tour. A rearrangement is then made if $a \sim c + b \sim d \leq a \sim b + c \sim d$.

Since $a \sim c + b \sim d \leq a \sim b + c \sim d \implies a \sim c < a \sim b$ or $b \sim d < c \sim d$ (or both), it's sufficient to only test all cities c that are closer to a than b, i.e check all cities c that appear before b in a's adjacency list. When a possible 2-opt has been found, the tour from c to b needs to be reversed, to achieve $a \sim c \sim ... \sim b \sim d$. This is done with the *reverse* operation on the tour data structure.

2.4.2 2H-opt

2H-opt (or 2.5-opt) is a another commonly used local optimisation. Assume that $a \sim b \sim c$ and $d \sim e$. 2H-opt works by checking if it is better to place b in between d and e such that $d \sim b \sim e$ and $a \sim c$ instead. In other words, 2H-opt tries to find an edge $d \sim e$ such that $a \sim c + d \sim b + b \sim e < a \sim b + b \sim c + d \sim e$.

If a possible 2H-opt has been found, the "update procedure" on the tour differs from 2-opt. Instead of reversing a segment of the tour, a single city is relocated. b is taken out of the tour, all cities up until $d \sim e$ are shifted one step and finally b is inserted between d and e.

2.4.3 3-opt

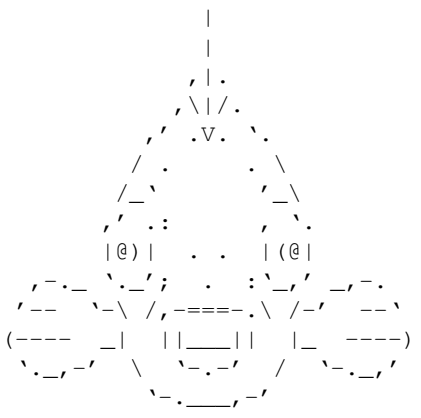
3-opt is an extension of the same concept used in 2-opt. The goal is to find three edges (as opposed to two edges in 2-opt) such that a new arrangement of edges between the six distinct cities forms a shorter tour. An optimisation criteria similar to the one in 2-opt is used to speed up the search for possible 3-opt moves. The program picks $a \sim b$, finds a d in $c \sim d$ such that d is close to a, finds a f in $e \sim f$ such that f is close to b, checks if $a \sim d + b \sim f + c \sim e < a \sim b + c \sim d + e \sim f$ and if the condition holds, a sequence of two *reverse* operations (chosen depending on the relative indices of a, c and e in the tour city index array) are performed. The operations must be carried out carefully to avoid creating an infeasible tour.

2.4.4 k-opt

k-opt is a generalisation of the optimisations mentioned in previous sections. A k-opt algorithm can be constructed for any k such that $2 \leq k \leq N$. Picking $k = N$ is equivalent of removing every edge in the tour and trying a new configuration; obviously not a very viable value for k, even with optimisations like the ones used in 2-opt and 3-opt. Anyhow, additional algorithms (such as 4-opt) can be constructed in a similar fashion as the algorithms mentioned previously. However, they are also more expensive to run, so they may not be as viable as 2-opt or 3-opt. In this project, $k = 3$ was the maximum value of k that was implemented.

2.4.5 Full Fish Mode

A generalisation of k-opt, Full Fish Mode heuristic solves TSP optimally in sub-constant time.



2.4.6 Lin-Kernighan Heuristic

While k-opt rearranges a fixed number of edges, the Lin-Kernighan heuristic determines k at run-time; it is essentially a variable (or adaptive) k-opt. In each step of the algorithm, k steps has been executed and tests are made to see whether an additional step should be considered. Otherwise, the k edges are rearranged (k-opt for the current value of k) and a better tour is achieved. This algorithm (including how to decide whether or not to terminate the search for additional edges) is discussed thoroughly in [7].

2.5 Other Solving Methods

Creating initial tours and then doing local optimisations is the basis for solving TSP. There are many algorithms that have local optimisations, or variants of these, as subroutines, many of which emphasises on finding a global minimum. Below, a variety of such algorithms are presented.

2.5.1 Simulated Annealing

Simulated annealing is a method where local optimisations are performed without the requirement that they yield better results. Rather, worse solutions are accepted with some probability, p, which decreases as time goes (hence "Simulated Annealing"). In this way, the search is broadened. Allowing worse solutions gives the program the ability to avoid local minimums and climb into another part of the solution landscape.

2.5.2 Genetic Algorithms

It is possible to optimise TSP tours using genetic programming. The idea is that a population of tours is created, genetic mutations (random local shuffles of the tour) and crossovers (random combinations of several tours) are applied, and finally a winner is selected (or several) using natural selection. The process can be repeated indefinitely, constantly adding a few new random tours and improving the general quality of the population by combining these new random tours with the existing evolved ones, applying mutations in the process.

When combining several tours using crossover, it is necessary (for any gain to take place) that the resulting tour is feasible and somewhat intelligently pieced together. Thus, it is appropriate to run a light pass of local optimisations after performing crossover (or just mutation). Otherwise, the resulting tours would look mostly random (and stay random), which would be worthless. This is because there would be long roads wherever the tour (the DNA string) had been cut and re-combined, unless the cities around to the cut happened to be located close to each other.

2.6 Other Optimisations

To further improve the efficiency for solving TSP, many important optimisations were performed. In this section, the most influential optimisations are presented.

2.6.1 Smart Tour Reverse

As was stated in section 2.1.2, reversing a sub-tour (part of the tour) can be done in linear time. There is an easy optimisation that can be applied that greatly reduces the constant factor. Dubbed the "smart reverse" by the authors of this report, it considers the cost of performing a sub-tour reverse by looking at the number of elements that would have to be moved. Then, if that number is greater than $N/2$ it is cheaper to invert the indices and perform an "outer reverse". It reverses the remaining range of elements that wraps around the start/end of the array by swapping elements and moving the two pointers in opposite direction.

Using smart tour reverse, the upper bound on the size of the sub-tour being reversed shrinks from N to $N/2$, resulting in a performance gain.

2.6.2 Random Re-tours

After constructing an initial tour and applying local optimisations exhaustively, there is a large chance of having CPU time left. One way of spending the remaining CPU time is to perform random re-tours. What it does is essentially restarting the whole solver but with a completely random initial tour (see section 2.3.1). With luck, a better solution is found.

2.6.3 Local Random Shuffles

A drawback with random re-tours is that it does not re-use anything from previously calculated tours. Even if an optimised tour is stuck in a local minimum, surely the program should be able to make some use of it? So, instead of random re-tours, the program can run local random shuffles. It loads the best optimised tour known thus far, shuffles a small sub-tour, applies local optimisations and analyses whether the resulting tour is better than the previously known record. If a better tour was found then it is saved, and the process is restarted regardless of that, and repeats until time runs out.

Since this process re-uses previously computed tours that are quite good (locally optimal), better results can be achieved. It can be considered to be a kind of simulated annealing, where instead of allowing changes that deteriorate the tour it just completely destroys a part of the tour straight away, then applies the same kind of local optimisations to reach a (potentially new) local minimum.

3 RESULTS

3.1 Testing

Within this project a lot of effort was put into initial boiler-plate code and overall system design. In this section, the most important features of the system is described.

3.1.1 Test Suite

A local test suite was created that runs six distinct test cases, each with a 2 second CPU time limit. Three of the test cases were collected from the Forschungsinstitut für Diskrete Mathematik at the Bonn Institute and they have known optimal solutions. This allows the program to compare the solution found during the 2 seconds with the known optimal solution and present the ratio between these solutions. The three remaining test cases were initially generated by the program itself, and the optimal solutions for these cases are not known. They are nevertheless evaluated and the best tour found can be compared to subsequent solutions. If the test suite indicates that a program build finds better solutions than before then it typically means that the program will perform better on KATTIS as well. It is impossible to know if the local test suite correlates well with the test cases used by KATTIS, but over the course of the project it turned out that it correlated quite well.

3.1.2 Visualisation

Part of the boiler-plate code is code for visualising the TSP tours. For this, Windows Forms were used.

Visualising tours is a very powerful tool for debugging and also for getting ideas on how to proceed; what optimisations and algorithms to implement next.

3.2 Profiling

CPU profiling was used in this project to find and fix defects to boost the run-time performance of the program. The final version of the program has the following distribution of CPU time (measured using the test suite mentioned in section 3.1.1). The numbers are slightly affected by print statements and other overhead that is not compiled when running on KATTIS.

Module	CPU time (inclusive)
Input from file	≤ 0.1 %
Distance pre-calculation	3.8 %
Neighbourhood pre-calculation	1.5 %
Generate initial tour	0.1 %
Optimise initial tour	1.0 %
Local random shuffle*	89.8 %
Geometric random shuffle*	3.4 %
Optimise**	86.3 %
2-opt	15.2 %
2H-opt	14.4 %
3-opt	56.3 %
Other	1.4 %

* Includes mostly local optimisations
** Includes program-wide 2-opt, 2H-opt and 3-opt

3.3 Performance

In this section, an attempt to show the efficiency of the implemented algorithms is made. Because the problem is NP-hard, there is a big interest in whether found solutions are near-optimal. All performance tests are made on test-cases within the scope of the KATTIS assignment.

3.3.1 KATTIS Performance Chart

Apart from local test suites, KATTIS also gives good feedback; it is essentially a test suite. For each submission, KATTIS gives a score based on how many, and how accurately, the program perform on the 50 test-cases provided.

In figure 1, the score from the major algorithms and algorithm combinations is presented in a bar-graph. The maximum score is 50, which corresponds to perfect solutions for all 50 test cases.

3.3.2 Solution Quality

The best solution uses pre-calculated pair-wise city distances, pre-calculated neighbourhood lists of size $\frac{N}{17}$, double-precision floating point distances, city index array tour representation, lookup table for said array, MF tour initialisation, local optimisations (using 3-opt, 2-opt and 2H-opt) and local/geometrically local random shuffles until deadline. The submission ID of the (at the time of this writing) best KATTIS submission is 471235 and it yielded roughly 47 points.

4 CONCLUSION

In the end, the results of the implementation for the KATTIS assignment were satisfactory. The pre-calculations were very important to the overall performance, and the local random shuffles improved the quality of the results remarkably.

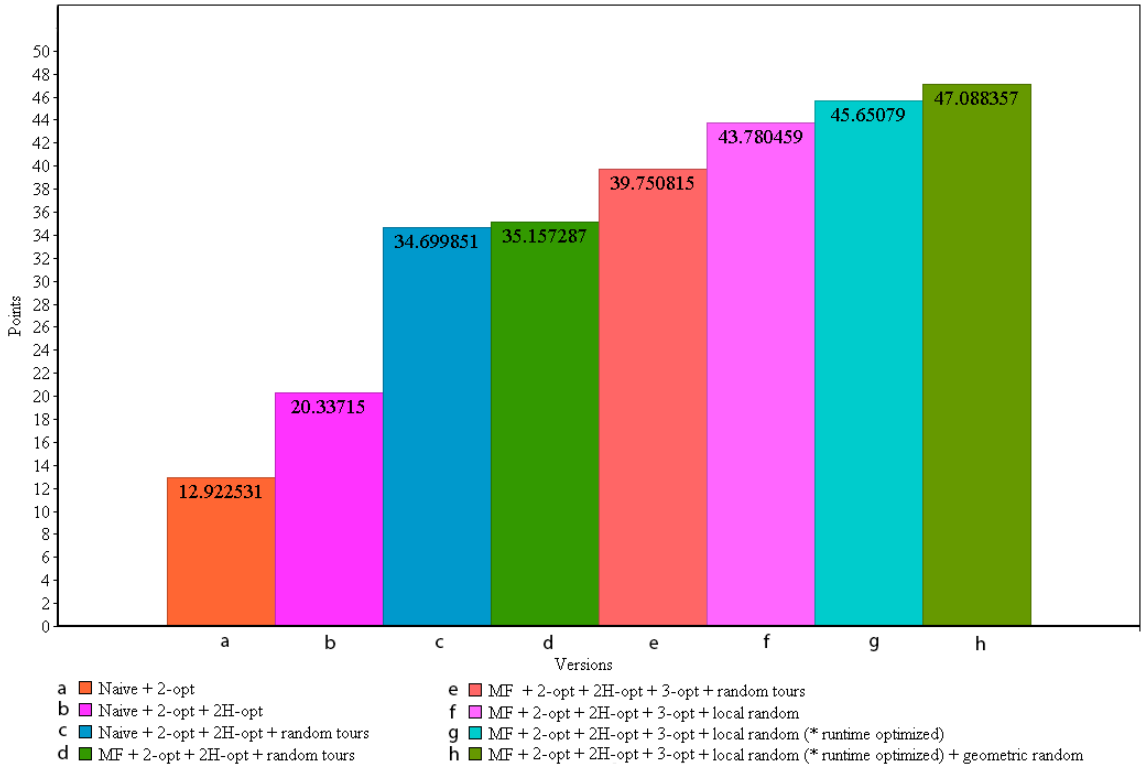


Figure 1. Chart reflecting KATTIS performance for different algorithms.

References

[1] (Dec. 2013), [Online]. Available: https://kth.kattis.scrool.se/.

[2] (Dec. 2013), [Online]. Available: http://cgi.di.uoa.gr/~sgk/teaching/grad/handouts/karp.pdf.

[3] (Dec. 2013), [Online]. Available: http://en.wikipedia.org/wiki/Travelling_salesman_problem.

[4] (Dec. 2013). Travelling salesman problem, set 1 (naive and dynamic programming), [Online]. Available: http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/.

[5] J. L. Bentley, "Fast algorithms for geometric traveling salesman problems.", *ORSA Journal on Computing*, 1992, ISSN: 08991499. [Online]. Available: http://search.ebscohost.com.focus.lib.kth.se/login.aspx?direct=true&db=bsh&AN=4472271&site=ehost-live.

[6] (Dec. 2013). Worst-case analysis of a new heuristic for the travelling salesman problem, [Online]. Available: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA025602.

[7] (Dec. 2013). An effective implementation of the lin-kernighan traveling salesman heuristic, [Online]. Available: http://www.akira.ruc.dk/~keld/research/LKH/LKH-2.0/DOC/LKH_REPORT.pdf.