Optimizing preprocessing and local search heuristics of the traveling salesman problem with parallel programming

Pietari Kaskela

School of Science

Bachelor's thesis
Espoo 6.12.2020

Supervisor

Prof. Fabricio Oliveira

Advisor

MSc (Tech.) Juho Andelmin

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the bachelor's thesis

Author Pietari Kaskela

Title Optimizing preprocessing and local search heuristics of the traveling salesman problem with parallel programming

Degree programme Engineering Physics and Mathematics

Major Mathematics and Operations Research **Code of major** SCI3029

Teacher in charge Prof. Fabricio Oliveira

Advisor MSc (Tech.) Juho Andelmin

Date 6.12.2020 **Number of pages** 0 **Language** English

Abstract

This thesis aims to optimize algorithms for solving the traveling salesman problem (TSP) with parallel programming. The traveling salesman problem is about finding the shortest possible route, which visits all of the given cities, returning to the starting city. The problem is NP-hard, which means that there are no fast exact algorithms for solving the problem. However, inexact heuristic solution methods work very well in practice and are capable of producing near optimal solutions for large problem instances.

The heuristic TSP solver implemented in this thesis first preprocesses the input graph to identify those edges, which most likely belong to the shortest route. The identification is done with α-values, which have been empirically verified to produce good candidate edges. The generated set of candidate edges is then used to form a number of initial routes, which are then improved with local heuristic algorithms. The most computationally heavy parts of the algorithms are implemented using CUDA parallel programming language. CUDA allows the distribution of the workload to thousands of parallel threads in graphics processing units (GPUs).

The optimization of α-values in graph preprocessing achieves a speedup of up to 10 times compared to a normal implementation, by parallelizing the minimum spanning tree calculation. Parallel programming is also applied to the 2-opt heuristic, which achieves speedups of up to 10-40 times, compared to a normal implementation. The effectiveness of combining these parallel implementations into a full solver is demonstrated by obtaining competitive results in solving a set of benchmark instances from TSPLIB. The developed implementations are also easily generalizable to n-dimensional problem instances with different metrics.

Keywords Traveling salesman, TSP, CUDA, Parallel programming, 2-opt, alpha-nearness

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Tekniikan kandidaatintyön tiivistelmä

Tekijä Pietari Kaskela

Työn nimi Kauppamatkustajan ongelman esiprosessoinnin ja heuristiikkojen optimointi rinnakkaisohjelmoinnilla

Koulutusohjelma Teknillinen fysiikka ja matematiikka

Pääaine Matematiikka ja Systeemitieteet **Pääaineen koodi** SCI3029

Vastuuopettaja Prof. Fabricio Oliveira

Työn ohjaaja MSc (Tech.) Juho Andelmin

Päivämäärä 6.12.2020 **Sivumäärä** 0 **Kieli** Englanti

Tiivistelmä

Kauppamatkustajan ongelmassa ratkaistaan lyhyintä mahdollista reittiä, joka kulkee kaikkien ongelman kaupunkien läpi tasan kerran palaten lähtökaupunkiin, olettaen että kaupunkien välisten matkojen pituudet tiedetään. Tässä tutkielmassa käsitellään kauppamatkustajan ongelman ratkaisua nopeuttamalla algoritmeja rinnakkaisohjelmoinnin avulla. Ongelman tiedetään olevan NP-vaikea, eli ei ole olemassa algoritmia, joka pystyisi ratkaisemaan ongelman polynomisessa aikavaativuudessa suhteessa kaupunkien määrään. Käytännössä tämä tarkoittaa sitä, että pelkästään hyvin pieniä ongelmailmentymiä pystytään varmuudella ratkaisemaan optimaalisesti. Epätarkat heuristiset algoritmit toimivat kuitenkin käytännössä hyvin isoillekin ongelmailmentymille ja niillä voidaan saavuttaa jopa vain muutaman prosentin virhe verrattuna parhaaseen mahdolliseen ratkaisuun.

Tässä kandidaatintyössä toteutettu algoritmi pyrkii ensin löytämään kaikkien verkon kaarien joukosta lupaavimmat kaaret, jotka todennäköisimmin kuuluvat lyhimpään mahdolliseen reittiin. Nämä kaaret tunnistetaan käyttäen hyväksi alfa-arvoja, joiden on todettu antavan hyvän arvion kaarien kuulumisesta optimiratkaisuun. Täten muodostetun lupaavien kaarien ehdokasjoukon avulla tuotetaan useita lupaavia reittiehdotuksia, joita pyritään parantamaan paikallisten heuristisen menetelmien avulla. Kaarien ehdokasjoukkoa voi myös käyttää paikallisten heuristisen menetelmien parantamiseen. Algoritmit suunniteltiin ja ohjelmoitiin käyttämään hyödykseen tietokoneiden grafiikkasuorittimia CUDA-rinnakkaisohjelmointikielen avulla. Grafiikkasuorittimia hyödyntävällä rinnakkaisohjelmoinnilla voidaan saavuttaa jopa 10-100 -kertaisia nopeutuksia verrattuna algoritmien toteutuksiin, jotka on suunniteltu tavallisille suorittimille. Vaikka monet algoritmit eivät ole nopeutettavissa rinnakkaisohjelmoinnin avulla, useat kauppamatkustajan ongelman ratkaisuun soveltuvat algoritmit nopeutuvat rinnakkaisohjelmoinnilla huomattavasti.

Kauppamatkustajan ongelman verkon esikäsittelyn alfa-arvojen optimointia nopeutettiin siirtämällä pienimmän virittävän puun laskenta grafiikkasuorittimelle, millä saavutettiin yli 10-kertainen nopeutus verrattuna normaaliin algoritmin toteutukseen. Paikallinen 2-opt heuristiikka toteutettiin myös grafiikkasuorittimella saavuttaen yli 10-kertainen nopeutus verrattuna algoritmin tavalliseen toteutukseen. Yhdistämällä nopea esikäsittely ja paikallinen 2-opt heuristiikka kokonaiseksi kaup-

pamatkustajan ongelman ratkaisijaksi päästiin kilpailukykyisiin tuloksiin euklidisten kaksiulotteisten ongelmailmentymien ratkaisussa. Kehitetty ratkaisin ja algoritmit ovat helposti yleistettävissä ratkaisemaan myös n-ulotteisia ja muita kuin euklidista mittaa käyttäviä kauppamatkustajan ongelman variaatioita.

Avainsanat Kauppamatkustajan ongelma, 2-opt, alpha-läheisyys, rinnakkaisohjelmointi, CUDA

Contents

1 Introduction

The traveling salesperson problem (also called the traveling salesman problem or TSP) is one of the most studied problems in optimization literature. In essence, given the list of cities and distances between them, the objective is to construct the shortest route possible that visits all cities once and returns to the origin city.

While the problem itself has roots somewhere in the 19th century or earlier, methodological attempts at solving the problem have been documented only from the 20th century. **?** states that Merrill M. Flood was the first person to try to mathematically solve the problem in the 1930s. Continued study during the 1950s and 1960s saw instances up to 20–50 cities being solved by formulating the problem as an integer linear program and solving it with a cutting plane method. During these decades, the necessary study for the background and subroutines of current state-of-the-art methods was conducted. One such algorithm is the 2-opt **?** and its generalization λ-opt, which can be a part of a larger heuristic, such as the Lin-Kernighan heuristic by **?**.

The Lin-Kernighan heuristic with subsequent additions and refinements by **?** is considered to be the state-of-the-art heuristic to solve the TSP. This implementation is called LKH, and successors and different versions of the LKH solver with source code are freely available on the web. With the increase in personal computing capability, especially in the format of parallel processing capability in Graphical processing units (GPUs), effective parallel heuristics have also been studied recently in **?**.

It is easy to see how solving the TSP effectively could be of help to a large number of practical logistics and planning problems, for example with optimizing postal delivery routes. Other uses include optimizing drilling routes for CNC-machines and designing electronic circuit boards with the least amount of solder used.

The objective of this thesis is to first identify algorithms for solving the TSP which would greatly benefit from parallel programming and then implement them. The focus will be on the building blocks of the LKH, as it is a battle-tested and old state-of-the-art solver. To achieve this, the first part goes over the necessary background and methods in Section **??** and then Section **??** discusses the implementation details and some enhancements to the algorithms presented previously. Finally, Sections **??** and **??** compare the parallel and CPU implementations of the algorithms against each other.

2 Background and Methods

2.1 Graph theory

A graph $G = (V, E)$ consists of a set V of vertices and a set $E \subset \{(i, j) \mid i, j \in V\}$ of edges. In this thesis, a graph refers to an undirected simple graph.

Path is then a set of edges

$$\{(i_1, i_2), (i_2, i_3), ..., (i_k, i_{k+1}) \mid i_p \neq i_q \; \forall p \neq q\}.$$

Cycle is a path with $i_{k+1} = i_1$:

$$\{(i_1, i_2), (i_2, i_3), ..., (i_k, i_1)\},$$

and a tour is a cycle with $k = n$.

A graph G is connected if, for all pairs of vertices (i, j), there exists a path $\{(i, i_2), (i_2, i_3), ..., (i_k, j)\}$ connecting the vertices i and j.

Most graphs in this thesis are dense, which means that there is an edge between all pairs of vertices. A single vertex is then associated with $n - 1$ edges, one for all other vertices in the graph. The number of edges associated with a vertex is called the degree of a vertex.

For later concepts, it is also important to define what a minimum spanning tree (MST) is. MST of a connected graph $G = (V, E)$ is graph $T = (V, E')$, in which $E' \subset E$, such that T is connected and $|E'| = n-1$, that is, the number of edges is one less than the number of vertices. If edges have an associated cost matrix $C = (c_{i,j})$, then the edges E' of an MST are defined such that the combined length $\sum_{e' \in E'} c_{e'_i, e'_j}$ is minimized. Algorithms for computing the MST of a graph are discussed in Section **??**.

2.2 Travelling salesperson problem

As previously mentioned, the TSP is a problem in which we must construct the shortest route possible, that visits all vertices of the graph and returns to the starting vertex. More formally, given a list of vertices numbered from 1 to n, where n is the number of vertices and an associated cost matrix $C = (c_{i,j})$, which denotes the cost of moving from vertex i to j, find a permutation $i_1, i_2, i_3, ..., i_n$ of integers 1 through n, that minimizes the cost $c_{i_1,i_2} + c_{i_2,i_3} + ... + c_{i_n,i_1}$. The permutation of vertices is also called a tour. Unless otherwise stated, n will denote the number of vertices of the TSP for the rest of this thesis. Figure **??** shows a suboptimal and the optimal tour of a graph.

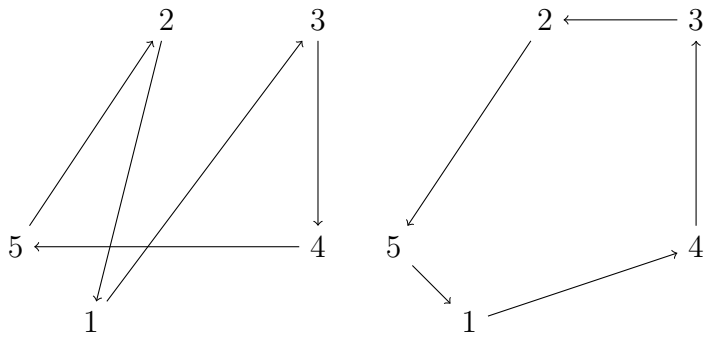


Figure 1: A suboptimal tour and an optimal tour.

The properties of the cost matrix C can be used to classify and optimize certain subsets of problems. If C is symmetric, that is $c_{i,j} = c_{j,i}$, for all i and j, then the TSP is said to be symmetric and otherwise asymmetric. This thesis will focus on the

symmetric TSP, as an asymmetric TSP of size n can be transformed to a symmetric instance of size $2n$ **?**.

The TSP optimization problem has been shown to be NP-hard by **?** and its corresponding decision problem version to be NP-complete. These constraints usually remain even for rather restricted TSP problems, such as when all vertices are on a plane with Euclidean distances.

Exact solution methods for the TSP include a simple brute-force search of all possible vertex permutations, having a time complexity of $O(n!)$, and the Held-Karp algorithm **?**. The Held-Karp algorithm is a dynamic programming solution method with a time complexity of $O(n^2 2^n)$, which in essence keeps track of the minimum tour length of all subsets of vertices ending at a specific vertex, starting from smaller subsets and ending in a full solution. The running times of these exact algorithms are however too large for most practical problems, with the theoretically faster Held-Karp algorithm being able to solve instances of only up to size 30. For practical problem sizes, inexact heuristic approaches are typically the most preferred solution method, although carefully designed branch and cut algorithms can solve larger instances up to thousands of cities. However, even the best exact method Concorde by **?** can struggle solving practical TSP problems with up to 2000 vertices on a standard desktop. For example, the drilling route optimization problem u2319.tsp from TSPLIB **?** can take hours to optimize with Concorde when the number of available cores is limited.

2.3 λ-opt algorithm

The λ-opt algorithm is a generalization of the previously mentioned 2-opt algorithm, in which at each iteration λ edges of the tour are replaced in a way which shortens the tour. A simple example of a 2-opt move can be seen in Figure **??** where the edges (B,D) and (C,A) are replaced with (B,C) and (D,A) which results in a shorter tour. Notice that the edge (D,C) is reversed in the process. In general, each 2-opt move requires reversing all edges between the two replaced ones in an appropriate direction.

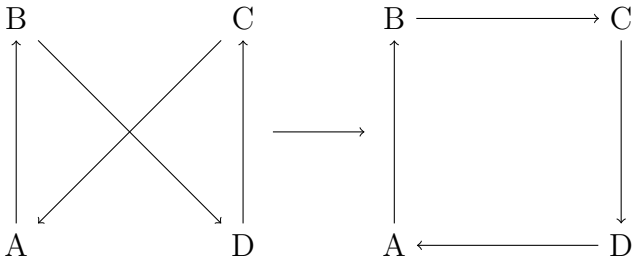


Figure 2: A 2-opt move.

These iterations are applied until the tour cannot be improved further by replacing λ edges. A tour optimal in this sense is called λ-optimal. By **?**, any λ-optimal tour is also λ'-optimal, for all $\lambda' \leq \lambda$. Furthermore, if we have a tour of length n which is n-optimal, then the tour is globally optimal, with respect to the TSP. λ-opt can

be a great tool for solving TSP, but the time complexity of the algorithm makes the use of large λ values impractical. A single iteration of the algorithm has a time complexity of $O(n^\lambda)$ where n is the length of the tour. n-opt computations in $O(n^n)$ is thus even slower than the naive brute-force method with $O(n!)$ time complexity. Pseudocode for 2-opt can be seen in Algorithm **??**.

Algorithm 1 2-opt

```
1:  improvement ← true
2:  while improvement do
3:      dist_best ← 0, i_best ← 0, j_best ← 0
4:      for i = 2, ..., n − 2 do
5:          for j = i + 1, i + 2, ..., n − 1 do
6:              dist_impr. ← c_{i,i−1} + c_{j,j+1} − (c_{i,j+1} + c_{i−1,j})
7:              if dist_impr. > dist_best then
8:                  dist_best ← dist_impr., i_best ← i, j_best ← j
9:              end if
10:         end for
11:     end for
12:     if dist_best is 0 then
13:         improvement ← false
14:     else
15:         reverseTour(i_best, j_best)
16:     end if
17: end while
```

Another variant of 2-opt does not keep track of the maximum improvement but instead applies the move as soon as it finds an improving one. As seen from the pseudocode, 2-opt has a time complexity of $O(n^2)$, which makes it quite slow when n becomes larger.

2.4 Candidate set generation

As the cost matrix defines distances for all pairs of vertices, it seems reasonable to assume that it is not worthwhile to consider some edges as potential candidates for being part of the optimal tour, for example, edges connecting to vertices that have relatively large distances between them compared to edges connecting to vertices with smaller distances. Furthermore, restricting the use of such edges when computing a tour reduces the running time of the algorithm. Deciding which edges are most promising with respect to being part of the optimal tour is called candidate set generation.

2.4.1 Nearest-neighbours

A simple heuristic for generating candidate sets with promising edges would be to only consider those edges which consist of a vertex and d of its closest neighboring

vertices. For example, the original Lin-Kernighan algorithm used 5-nearest neighbors as the candidate set (**?**).

2.4.2 α-nearness

α-nearness, which is derived from minimum spanning trees, is in practice a superior measure for candidate set generation compared to nearest-neighbours. This measure is based on the concept of a 1-tree **?** which, for a graph $G = (V, E)$, is formed by first excluding a vertex (traditionally 1, hence the name). A spanning tree is then formed on a graph without the excluded vertex. Finally, two edges incident to the excluded node are connected to the formed spanning tree. This is illustrated in Figure **??**.

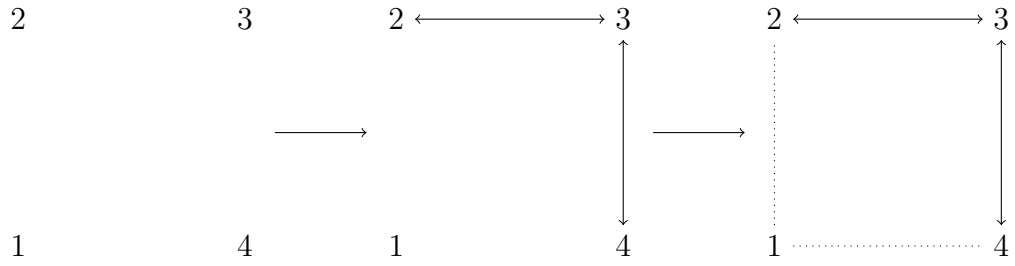


Figure 3: Forming a 1-tree.

A minimum 1-tree is a 1-tree with a combined minimum length of the edges. The minimum 1-tree can be computed by forming the spanning tree without the excluded vertex as a minimum spanning tree and choosing the two shortest edges connecting the excluded vertex to the spanning tree. By **?**, an interesting property of an optimal TSP tour is that it corresponds to a minimum 1-tree and similarly if a minimum 1-tree is a tour, then that tour must be optimal. Symmetric TSP can thus also be formulated as a problem of finding a minimum 1-tree whose vertices all have degree 2. According to **?**, an optimal tour normally contains between 70 to 80 percent of the edges of a minimum 1-tree which is why it leads to a good heuristic on whether an edge is part of the optimal tour. α-nearness is a measure based on these findings, formally defined as:

$$\alpha(i, j) = L(T^+(i, j)) - L(T),$$

where $\alpha(i, j)$ denotes the α-nearness of an edge (i, j), $L(T)$ is the length of a minimum 1-tree, and $T^+(i, j)$ is a minimum 1-tree containing the edge (i, j). The minimum α-value an edge can obtain is thus 0 when the edge is part of some minimum 1-tree. The candidate sets consist of edges with minimal α-values, usually with approximately 5-10 edges chosen per vertex.

For α-values to be useful in practice, the time complexity and memory requirements for computing them must be carefully considered. A naive implementation would require computing a minimum spanning tree for each of the n^2 edges, leading to a $O(n^2 n^2) = O(n^4)$ time complexity when MSTs are calculated with Prim's algorithm (**?**). This is too much for larger TSP problems, as we hope to solve instances of

sizes up to tens of thousands of vertices. The naive memory requirement $O(n^2)$ of storing all α-values is much more reasonable, but still unpractical for instance sizes of over ten thousand. To reduce the memory requirement, we can compute the candidate set size before running the algorithm and discard all edges that are not part of the final candidate set during computation, resulting in a practical $O(kn)$ memory requirement where k is the number of candidate edges per vertex.

? describe a more efficient algorithm for calculating α-values in $O(n^2)$. The algorithm starts by computing the minimum 1-tree, as previously described, in $O(n^2)$ time. Then we iteratively calculate α-values for all edges (i, j) by using information from the previous step. For the α-value of an edge (i, j) we have three cases:

1. If (i, j) belongs to T (the minimum 1-tree), then $T + (i, j)$ is equal to T and $\alpha(i, j) = 0$.

2. If either i or j is the excluded node, then longer of the edges incident to 1 in T is replaced with (i, j) and $\alpha(i, j) = c_{i,j} - max_{k \in T} \{c_{\text{excluded},k}\}$

3. Otherwise, (i, j) is inserted into T, creating a cycle in the spanning tree part of T. $T^+(i, j)$ is obtained by removing the longest edge in this cycle.

Cases 1. and 2. can be treated trivially in constant time with suitable 1-tree representations, but case 3. is more difficult. As there are possibly n edges in the formed cycle, naively checking them would lead to a total time complexity of $O(n^3)$ which is impractical. **?** solve this by introducing $\beta(i, j)$ values for each edge (i,j) which denotes the length of the edge to be removed from the spanning tree when edge (i, j) is added. Using this notation, we have:

$$\alpha(i, j) = c_{i,j} - \beta(i, j) \tag{1}$$

Now if (j_1, j_2) is an edge in the MST, i is a vertex of the MST, and j_1 is on the cycle that formed, then

$$\beta(i, j_2) = max(\beta(i, j_1), c_{j_1,j_2}) \tag{2}$$

This is illustrated in Figure **??** which shows the final $\beta(7, 5) = max(\beta(7, 4), c_{4,5})$ calculation. Notice that in this example $\beta(7, 4) = max(\beta(7, 2), c_{2,4})$ must be calculated first, as well as all $\beta(7, i)$, for all vertices i on the cycle. $\beta(7, 3) = c_{7,3}$ can be used as a starting point for the other calculations.

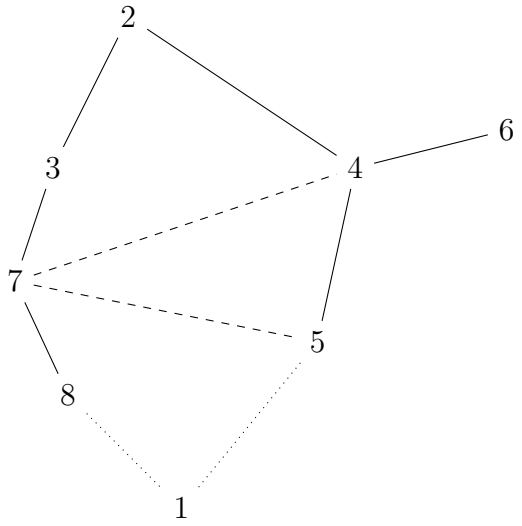


Figure 4: $\beta(7, 5)$ can be calculated from $\beta(7, 4)$.

After calculating the β-value as shown in Figure **??** and Equation **??**, the α-value can be simply calculated with Equation **??**. By combining these ideas with topological ordering, the α-values can be calculated efficiently with the following pseudocode shown in Algorithm **??**.

Algorithm 2 Efficient α-values

```
1: for i ← 0, ..., n − 1 do
2:     mark(i) ← −1
3: end for
4: for i ← 0, ..., n − 1 do
5:     from ← topo(i)
6:     β(from) ← −∞
7:     for to = from, to ≠ root, to = dad(to) do
8:         β(dad(to)) ← max(β(to), c_{to,dad(to)})
9:         mark(i) ← from
10:    end for
11:    for j ← 0, ..., n − 1, j ≠ i do
12:        to ← topo(j)
13:        if from = excluded ∨ to = excluded then
14:            if c_{from,to} ≤ excludedEdge then
15:                α(from, to) ← 0
16:            else
17:                α(from, to) ← c_{from,to} − excludedEdge
18:            end if
19:        else
20:            if mark(to) ≠ from then
21:                β (to) ← max(β(dad(to)), c_{to,dad(to)})
22:            end if
23:            α(from, to) ← c_{from,to} − β(to)
24:        end if
25:    end for
26: end for
```

In Algorithm **??**, $\beta(j) = \beta(from, j)$ is used to keep track of β-values for the *from*-vertex, *mark* is an array for keeping track of when β-values have been updated, *excluded* marks the excluded vertex and *excludedEdge* is the second-longest edge originating from *excluded*-vertex. Arrays *dad* and *topo* are computed during the calculation of the minimum spanning tree, where *dad(i)* marks the predecessor of node i in the MST and *topo* is a topological ordering of the MST, starting from the root of the tree which is denoted by *root*. These auxiliary arrays can be computed at almost no additional cost during a serial MST computation, for example, with Prim's algorithm. The excluded vertex is not part of the MST, but can be added to the topological ordering as the last value and *dad(special)* can be set as the closest vertex to it.

The following describes in more detail how Algorithm **??** handles the previously mentioned three cases:

1. If $(from, to)$ is part of the 1-tree, then lines 7-9 set $\beta(to) = max(\beta(to), c_{from,to}) = c_{from,to}$, as $\beta(to)$ was just set to $-\infty$. On line 23 we calculate the α-value as in Equation **??** to be $\alpha(from, to) = c_{from,to} - c_{from,to} = 0$.

2. If either $from$ or to in $(from, to)$ is the special node, then this is handled on lines 14-18. If the length of the edge is smaller or equal to the longer special edge, then the edge is part of the 1-tree and α-value is set to zero. Otherwise, α value is calculated by subtracting the special edge length from $c_{from,to}$.

3. Otherwise, the formed cycle in the MST can be decomposed into two parts (from, ..., root) and (root, ..., to). On lines 7-9, the maximal edges of paths (from, ..., to) are stored in $\beta(to)$, up to the *root*-vertex, which contains the maximal edge on that half of the cycle. Lines 11-12 start a for loop in which values of $\alpha(from, to)$ are calculated in topological order. Lines 20-21 propagate the maximum of the maximal edge of the first half from the *root* and the maximal edge of the second part of the cycle to $\beta(to)$. This value is then subtracted from $c_{from,to}$ to obtain the α-value.

This achieves a practical $O(n^2)$ time complexity with $O(kn)$ memory usage when only the k-smallest α-values are tracked.

2.5 Subgradient optimization

Unoptimized α-values can be used to generate a good candidate set, but the results can be improved with the following observations by **?**: As every tour is a 1-tree, then the length of a minimum 1-tree is a lower bound for the length of an optimal tour. Also, the lengths of all edges incident to a given vertex can be changed by the same amount π without changing the optimal tour. The tour still has to visit this vertex - only the length of the tour is increased by 2π. This transformation can be applied to all vertices and edges by defining a new cost matrix D:

$$d_{i,j} = c_{i,j} + \pi_i + \pi_j$$

An optimal tour with the old cost matrix is still optimal with d, only $2\sum \pi_i$ longer. The minimum 1-tree however is changed with the new cost matrix. By the first observation, if T_π is a minimum tree with the new cost matrix, then its length $L(T_\pi)$ is a lower bound for the length of an optimal tour for D. The lower bound for the original cost matrix and the optimal tour is then:

$$w(\pi) = L(T_\pi) - 2\sum \pi_i$$

By changing the values of π we can adjust the 1-tree to being closer to the optimal tour. As previously stated, an optimal tour is a minimum 1-tree with all vertices having a degree of two. We can guide the vertices of the 1-tree closer to having a degree of 2 by increasing the π value of the vertex if the vertex has a degree greater than two and by decreasing π if the degree is less than two. A suitable algorithm for calculating the optimum π-values is called a subgradient method. The pseudocode by **?** for the subgradient method is presented in Algorithm **??**.

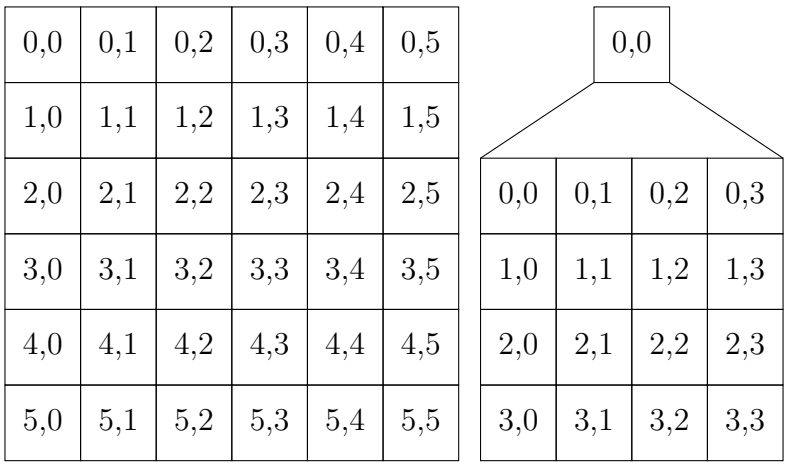
Algorithm 3 Subgradient optimization

1: $k \leftarrow 0$, $\pi^0 \leftarrow 0$, $W \leftarrow -\infty$
2: $T_\pi \leftarrow$ findMinimum1Tree(π^k)
3: $w(\pi^k) \leftarrow L(T_\pi) - 2\sum \pi_i$
4: $W \leftarrow max(W, w(\pi^k))$
5: $v^k \leftarrow d^k - 2$
6: **if** $v^k = 0 \vee$ stop criterion is satisfied **then**
7: return π^k
8: **end if**
9: $t^k \leftarrow$ chooseStepSize()
10: $\pi^{k+1} \leftarrow \pi^k + t^k v^k$
11: $k \leftarrow k + 1$
12: Go to Line 2

In Algorithm **??**, k denotes the current iteration of the method, W the maximum of $w(\pi^k)$, t^k the step-size and d^k the degrees of each individual vertex. **?** has proven that if $t^k \to 0$ as $k \to \infty$ and $\sum t^k = \infty$, then W will always converge to the maximum of $w(\pi)$.

2.6 Parallel programming with CUDA

Modern computers and smartphones usually come with a dedicated graphics processing unit (GPU) designed to accelerate parallel tasks. There are multiple APIs and programming languages for creating code that makes use of such auxiliary parallel processors, such as CUDA by **?** and OpenCL, OpenGL, and Vulkan for a wider range of devices. All of these implementations operate on mostly similar concepts of which most important are grids, blocks, and threads, illustrated in Figure **??**.



(a) A 6 by 6 grid of blocks.

(b) Block (0,0) with 16 threads.

Figure 5: Grid, blocks and threads.

The smallest unit of work is a thread. Some amount of threads are contained within a block and blocks are arranged in a superstructure called a grid. From a programmer's perspective, threads of a block are executed in parallel and sharing memory between threads is possible and recommended. As the theoretical computational performance (for example $11.3 \cdot 10^{12}$ single-precision floating-point operations per second on an Nvidia GTX 1080 Ti) is much larger than the memory bandwidth (484.4 GB/s on Nvidia GTX 1080 Ti), a recommended strategy is to load data in parallel from the main memory to a block's shared memory and utilize this fast shared memory as much as possible in computations while minimizing the data stored in the main memory.

With CUDA, each of the threads runs the same function called a kernel with block and thread indices given as parameters. Each kernel call receives a unique set of block and thread indices, which can be used to decide which part of the problem the kernel solves to avoid overlapping computation.

3 Algorithm & Implementation

In this section, we combine previously defined algorithms to form a full TSP solver. The main idea is to first preprocess the problem network to generate a candidate set using α-nearness and to optimize the α-values using subgradient optimization. The resulting candidate set is used to generate some initial tour candidates, which are then optimized by 2-opt to obtain a final candidate for an optimal tour.

3.1 α-nearness computation & optimization

After implementing the α-value computation and optimization using the subgradient method on CPU, the computation of an MST during each of the subgradient optimization iterations was identified to be the most costly operation and the best candidate for parallel optimization. The CPU implementation uses Prim's algorithm to compute the MST, but for the parallel implementation, Boruvka's algorithm by **?** was chosen. In contrast to the serial computation in Prim's algorithm which adds a new vertex to the MST each iteration, Boruvka's algorithm connects multiple connected components of a graph during each iteration. Pseudocode for Boruvka's algorithm is presented in Algorithm **??**.

Algorithm 4 Parallel Boruvka

```
1: Initialize each vertex as a component of a graph without any edges.
2: while |components| > 1 do
3:     Find a closest neighbor for each component.
4:     Remove cycles from edges found in previous step.
5:     Use pointer-doubling on all components to find which components to combine.
6:     Update components, degrees of the MST, and length of the MST.
7: end while
```

Parallel implementation of Boruvka's algorithm is straightforward to implement from a high-level description of the algorithm, but some ideas such as pointer-doubling were used from **?**. The main computational bottleneck of the algorithm is on line 3 of Algorithm **??**, where we must loop through all of the n^2 pairs of vertices to find each component's nearest neighbor. For all pairs of indices (i, j), the work is split so that each thread in a block of 64 threads loads a single value from range $j = blockIdx.y \cdot 64, \ldots, blockIdx.y \cdot 64 + 63$, respectively. Once all values are loaded, a thread calculates the minimal distance for all 64 pairs $i = threadIdx.x + blockIdx.x \cdot 64$ and $j = blockIdx.y \cdot 64, \ldots, blockIdx.y \cdot 64 + 63$ and saves it to shared memory. At the end of the kernel execution, the thread with index 0 loops through all the results saved by other threads in shared memory and commits them to global memory with atomic operations if they constitute a new global minimum for the component.

The following improvements to the previously stated algorithms are taken straight from **?**:

- Instead of excluding one vertex and adding it later to the MST, the MST is computed first for all of the vertices, and an edge is added to one of the leaf vertices of the MST. The edge is chosen to be the longest second-shortest edge incident to a leaf vertex.
- The subgradient update rule is changed to $\pi^{k+1} = \pi^k + t^k(0.7v^k + 0.3v^{k-1})$, where $v^{-1} = v^0$.
- The step size t^k stays constant for a fixed number of iterations called a period. The length of the first period is n/2 and the length of a period, as well as the step size, are halved each time a period ends.
- At the beginning, however, the step size is doubled until W does not increase. Also, if the last iteration of a period leads to an increase of W, then the period length is doubled.
- The algorithm terminates when the step size, length of the period, or v^k becomes zero.

The actual α-value computation and the calculation of the last MST were done fully on the CPU because of their serial nature.

3.2 Initial tour generation

The initial tour generation is a simplified version of that presented in **?** we start with a random vertex and iteratively choose the next tour vertices. The next vertex is chosen from the candidate set associated with the current vertex, by the smallest α-value. If multiple vertices share the same α-value, ties are broken with the smallest cost in the original cost matrix. If all vertices from the candidate set are already chosen, the closest vertex not in the candidate list is selected. Randomly choosing the first vertex seems to induce enough randomness to generate diverse-enough tour candidates.

3.3 2-opt

In addition to the non-restricted 2-opt variant described above, we also evaluate an improvement to the algorithm used in the original Lin-Kernighan heuristic. The improvement made by **?** is as follows: if we denote the edges removed in a λ-opt operation with $(x_1, x_2, \ldots, x_\lambda)$ and the edges replacing them as $(y_1, y_2, \ldots, y_\lambda)$, then each y_i should belong to the candidate set. In theory, this should lead to a better runtime, as there are fewer edges to check each iteration, and to a better quality solution because there are fewer local minima in which to get stuck.

For the unrestricted variant, the kernel is very similar to that in Section **??**. The problem is the same, as the algorithm must loop through all of the n^2 pairs of vertices and find a minimum. The improved variant is more difficult to effectively parallelize, as the candidate set is spread around the whole tour. Only modest speedups are made compared to a CPU implementation with having each thread looping through a vertex's candidate set and committing the minimum to global memory if necessary.

The CPU implementation includes some parallel optimization to make the comparison more relevant. The outer-loop of the 2-opt is parallelized between cores available on the machine so that each core gets approximately the same amount of work. This is done using the OpenMP by **?**.

4 Results

The experiments were conducted on the author's computer with the following hardware specifications: Intel i7-8700k processor at 4.8GHz, 32GB 3200MHz DDR4 memory, and Nvidia GTX 1080 Ti GPU.

4 different variations of the algorithm are compared, each evaluated 100 times with different random starting tours.

1. (RAND) Random tour initialization with normal 2-opt.
2. (ALPHA1) α-value initial tour generation as in Section **??**, with normal 2-opt.
3. (ALPHA2) α-value initial tour generation as in Section **??**, with 2-opt restricted to candidate set edges.
4. (ALPHA3) α-value initial tour generation as in Section **??**, with 2-opt restricted to candidate set edges, followed by normal 2-opt.

2D euclidean instances from TSPLIB by **?** were used as benchmark instances and all algorithms were run with a candidate set size of 10.

4.1 Solution quality

We measure solution quality only on the GPU-variants of the algorithm, as the CPU implementations have similar performance but are just slower. The results are detailed in Table **??**, where an asterisk (*) after the optimum denotes that the value is an upper bound for an optimum solution.

Instance	RAND	ALPHA1	ALPHA2	ALPHA3	Optimum
eil101	680	**668**	739	669	629
a280	2864	**2748**	3239	2751	2579
att532	94720	91159	108755	**90857**	86729
pr1002	283706	**271534**	322240	272928	259045
d1291	59736	53835	58693	**53350**	50801
d1655	71483	**65789**	75145	65944	62128
d2103	93343	82268	87406	**82127**	80450*
pcb3038	155614	145993	167543	**145544**	137694
rl5915	682396	606610	680485	**602801**	565530*
rl11849	1091468	**982572**	1122952	982636	923307*

Table 1: Best results for each algorithm, with best overall bolded.

As can be seen from Table **??**, variants ALPHA1 and ALPHA3 perform quite similarly and considerably better than ALPHA2 and RAND variants. The worse performance of ALPHA2 came as a bit of a surprise because of the results of **?**, which are consistently better with a small candidate sets applied to λ-opt. The worse results might be due to the use of only 2-opt in this solver, as the LKH uses a 5-opt move as its base move. Table **??** details the errors as percentage distances from the optimum values.

Instance	RAND	ALPHA1	ALPHA2	ALPHA3
eil101	8.1	6.2	17.5	6.4
a280	11.1	6.6	25.6	6.7
att532	9.2	5.1	25.4	4.8
pr1002	9.5	4.8	24.4	5.4
d1291	17.6	6.0	15.5	5.0
d1655	15.1	5.9	21.0	6.1
d2103	16.0	2.3	8.6	2.1
pcb3038	13.0	6.0	21.7	5.7
rl5915	20.7	7.3	20.3	6.6
rl11849	18.2	6.4	21.6	6.4
Average error	13.9	5.7	20.2	5.5

Table 2: Errors of the algorithms from the optimal values as percentage distances.

The considerably lower error percentages of the α-value calculation augmented algorithms demonstrate the usefulness of the α-measure. The impact of the α-measure is especially visible on larger instances, in which a normal 2-opt has a huge number of local minima in which it can get stuck, thus leading to large error percentages. α-value based methods are able to circumvent most of these local minima to achieve significantly lower error percentages.

4.2 Efficiency

The running time for all four variants of the algorithms is measured in milliseconds on both CPU and GPU, and the computation times are presented in Table **??**. The average running time is calculated on instances between eil101 to pcb3038 in order to compare the efficiency of all GPU and CPU algorithms on average.

Instance	RAND GPU	ALPHA1 GPU	ALPHA2 GPU	ALPHA3 GPU	RAND CPU	ALPHA1 CPU	ALPHA2 CPU	ALPHA3 CPU
eil101	454	275	231	298	91	125	107	109
a280	1020	633	493	614	718	984	772	1376
att532	2219	1578	1297	1631	5339	6080	5006	5602
pr1002	6743	5797	3824	4424	39628	25900	21410	23377
d1291	7088	7268	6956	7409	95540	45515	40319	46100
d1655	11102	13555	12222	14298	197076	104244	89109	98342
d2103	14706	19822	19082	19785	367981	140303	126630	139569
pcb3038	24887	52771	49327	54765	1085717	579399	520704	730849
rl5915	125805	198356	188176	198295				
rl11849	913566	1084582	1026429	1046155				
Average	8527	12712	11679	12903	224011	112818	100507	130665

Table 3: Combined running times for all algorithms in milliseconds.

While small instances are still faster with CPU implementaions, as can be seen in Table **??**, GPU implementations are significantly faster on average. Larger instances were not even computed with CPU implementations, as the running time became excessively large. To illustrate the speedup better, Figure **??** shows how the algorithms scale as a function of instance size.

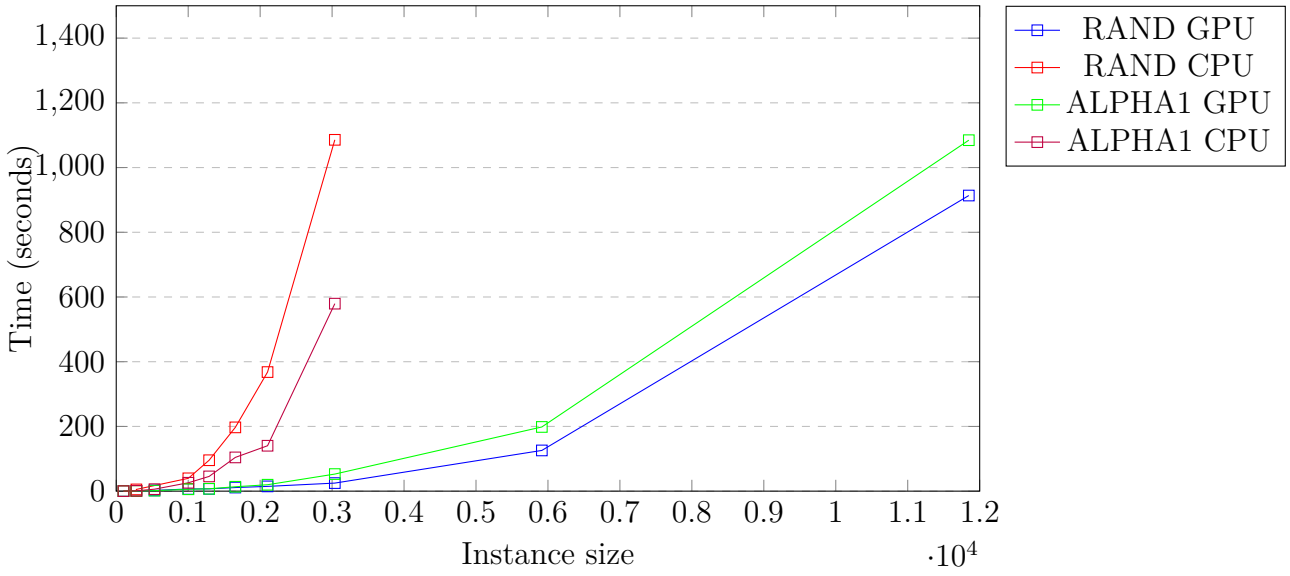


Figure 6: RAND and ALPHA1 solver computation times as a function of instance size.

The scaling of the CPU-variants is much steeper, with GPU implementations capable of solving 5-times larger instances than the CPU implementations in a similar amount of time. These results were expected, as CPUs can traditionally handle 4-16 tasks at the same time, but modern GPUs can concurrently handle thousands of threads. As all ALPHA-variants have very similar timings, the plot only shows ALPHA1. The speedups are especially visible in the largest instance solved with CPU implementations, which is detailed in Figure **??**.

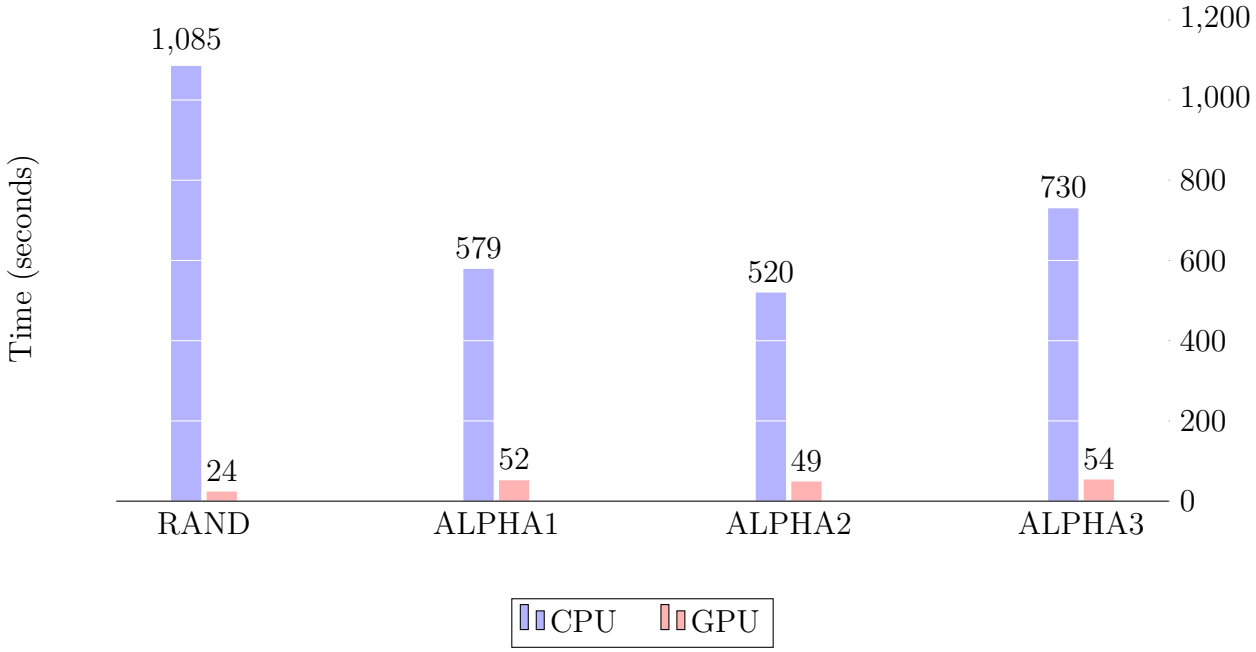


Figure 7: Running times for instance pcb3038 in seconds.

On pcb3038-instance, the GPU RAND is 40 times faster than the CPU counterpart. The GPU ALPHA algorithms also beat their counterparts by a factor of 10. Tables **??** and **??** in the Appendix details the running times for both the preprocessing (subgradient optimization and α-values) and 2-opt in milliseconds.

5 Conclusions

This thesis aimed to identify effective places for parallelization in the algorithms underlying the state-of-the-art TSP solver LKH. Parallel programming was applied to the 2-opt algorithm and α-nearness subgradient optimization, gaining speedups of up to 10–40 times for the GPU implementations over the CPU ones. The effectiveness of α-nearness as a measure for candidate set generation was also verified, and the solutions obtained by the parallel GPU algorithms achieved significantly better results compared to standard implementations, with respect to optimal solution costs. Based on the results of this thesis, a parallel implementation of a state-of-the-art solver, such as the LKH, could offer considerable gains in computation time.

6 Appendix

Instance	RAND 2-opt	ALPHA1 2-opt	ALPHA1 prep.	ALPHA2 2-opt	ALPHA2 prep.	ALPHA3 2-opt	ALPHA3 prep.
eil101	454	92	183	53	178	113	185
a280	1020	336	297	200	293	323	291
att532	2219	1008	570	731	566	1081	550
pr1002	6743	3322	2475	2507	1317	3167	1257
d1291	7088	5284	1984	4930	2026	5424	1985
d1655	11102	9944	3611	8995	3227	10084	4214
d2103	14706	14426	5396	13655	5427	14391	5394
pcb3038	24887	35433	17338	32028	17299	34484	20281
rl5915	125805	152106	46250	142345	45831	152962	45333
rl11849	913566	764269	320313	700677	325752	742629	303526

Table 4: Execution times for GPU-variants in milliseconds.

Instance	RAND 2-opt	ALPHA1 2-opt	ALPHA1 prep.	ALPHA2 2-opt	ALPHA2 prep.	ALPHA3 2-opt	ALPHA3 prep.
eil101	91	49	76	38	69	39	70
a280	718	286	698	246	526	875	501
att532	5339	1672	4408	906	4100	1522	4080
pr1002	39628	7544	18356	3063	18347	5316	18061
d1291	95540	12337	33178	5194	35125	11625	34475
d1655	197076	24490	79754	9110	79999	19204	79138
d2103	367981	32074	108229	17350	109280	30566	109003
pcb3038	1085717	132144	447255	35963	484741	244017	486832

Table 5: Execution times for CPU-variants in milliseconds.