A MASSIVELY PARALLEL EXACT TSP SOLVER

FOR SMALL PROBLEM SIZES

by

Benila Virgin Jerald Xavier, B.E.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2022

Committee Members:

Martin Burtscher, Chair

Vangelis Metsis

Kecheng Yang

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

This work is dedicated to my husband Prasanth, who has been a constant source of support and encouragement during all the challenges over the last couple of years. I am truly thankful for all his sacrifices. I would also like to dedicate this work to my daughter who has been putting up with my rigorous schedule and driving me to be the best version of myself every day.

In loving memory of my father, who will always be on my side.

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my advisor Dr. Martin Burtscher, for giving me the opportunity to work with him and being supportive throughout this thesis. I am thankful for his encouragement, guidance, and patience during my research.

I would also like to extend my sincere thanks to Dr. Metsis and Dr. Yang for being a part of my thesis committee and contributing their time and support.

I am thankful for the supportive environment and great education system at Texas State University, which has helped me immensely throughout my Master's degree.

Finally, I am extremely grateful to my family and friends for all the love and support. None of this would have been possible without them.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

The Traveling Salesman Problem (TSP) is a combinatorial optimization problem tasked with finding the shortest tour for visiting a set of cities such that each city is visited exactly once, and the tour ends in the starting city. This problem has gained attention among researchers because it is easy to describe yet difficult to solve.

TSP has numerous important real-life applications, but its NP-hardness makes it difficult to find an optimal solution even for relatively small problem sizes. The literature describes many heuristic algorithms that solve the problem approximately but only few exact algorithms.

The TSP solver implemented in this study is a GPU-accelerated exact solver for small problem sizes. The goal is to exploit the computing capabilities of modern GPUs for finding an optimal solution using simple algorithms. The algorithms used are exhaustive search for up to 7 cities and branch and bound for problems up to 30 cities.

The branch and bound algorithm performs an irregular traversal of the search tree, making it challenging to parallelize efficiently, especially for massively parallel GPUs. I implemented the algorithm in CUDA and tested it on GPUs with different compute capabilities. My solver is exact and very fast. It outperforms the CONCORDE and LKH solvers for problems with up to 15 cities. On a 13-city instance, my solver is 36 and 15 times faster than CONCORDE and LKH, respectively.

1. INTRODUCTION

The Travelling Salesman Problem (TSP) is a classical optimization problem that finds the shortest possible tour for a given set of cities visiting each city exactly once and returning to the origin. The problem statement of TSP may look simple but is very difficult to solve. Since TSP is an NP-hard problem, it becomes geometrically more difficult to find an optimal solution when more cities are added to the problem. This characteristic of TSP has made it an interesting area of research for decades. The objective of this thesis is to design a simple exact TSP solver for small problem sizes running on massively parallel GPUs.

TSP is one of the best-known NP-hard problems. There is no known exact algorithm to solve it in polynomial time. Mathematically, it is represented as a set of n cities, and these n cities have $(n-1)!$ possible tours. So, an increase in the number of cities results in an exponential increase in the number of paths. The goal is to find the minimum distance tour (Hamiltonian path) in which all cities are visited. TSP can be represented as an undirected graph $G = (V, d)$, where V is the set of cities to be visited and $d_{i,j}$ is the cost of traveling from city i to city j. This thesis' goal is to design a GPU version for finding the exact solution quickly, using exhaustive search for small inputs (up to 7 cities) and a branch and bound algorithm for larger inputs (up to 30 cities).

Applications of TSP in its pure form and as a sub-problem have made it an interesting area of research. For example, it has a wide range of applications in the field of science and technology such as routing of delivery trucks where it is used to find the optimal vehicle route with minimum total cost [17]. It is also used in planning of robot arm movements to enhance production levels [11], scheduling the drilling of holes in

printed circuit boards [18], analyzing the structure of crystals [19], and connecting components on a computer board [19]. It has further been used for photographic mask plotter control in PCB production and wireless sensor networks [20]. Because of its broad application, it is important to develop efficient TSP solvers.

Exact algorithms for finding the TSP solution are exhaustive enumeration algorithms [21], dynamic programming [1], branch and bound [22], constraint programming [23], and hybrid methods [24]. Enumeration algorithms generate all possible tours for a given set of cities and evaluate them to find the shortest distance.

The broad application of TSP in different fields and its NP-hardness have motivated researchers to find faster "heuristic" solutions. Several approximation algorithms that use heuristics have been developed to solve the TSP problem. Heuristic algorithms such as Ant colony optimization [5, 8, 18] and iterative hill climbing [25] can find a near optimal solution and focus mostly on solving larger instances.

The rest of this thesis is organized as follows. In Section 2, the approach used to design my TSP solver is discussed. In Section 3, related literature on exact solutions for the Traveling Salesman Problem is reviewed. Section 4 explains the evaluation methodology. Section 5 discusses the results and analyzes the solver implemented in this study. Section 6 summarizes the thesis and discusses the conclusion and possible future improvements.

2. APPROACH

The exact TSP solver designed in this thesis uses a hybrid approach (both CPU and GPU) to find the solution quickly. To achieve maximum performance, I decided to use exhaustive search on the CPU for small problem sizes up to 7 cities and the branch and bound algorithm on the GPU for problem sizes between 8 and 30 cities. The TSP solver I implemented for this study reads inputs from TSPLIB [13]. It supports most of the TSPLIB formats, including the two-dimensional Euclidean, Explicit, Geo, and CVRP formats.

For a given set of *n* cities, the exhaustive algorithm enumerates the search space for all possible paths and determines the least cost path. The distance between all city pairs is pre-computed and stored as a distance matrix. Consider the example graph shown in Figure 2.1, where the graph represents a TSP of 4 cities with the distance between each city pair marked. Figure 2.2 shows the search tree to describe the exhaustive algorithm.

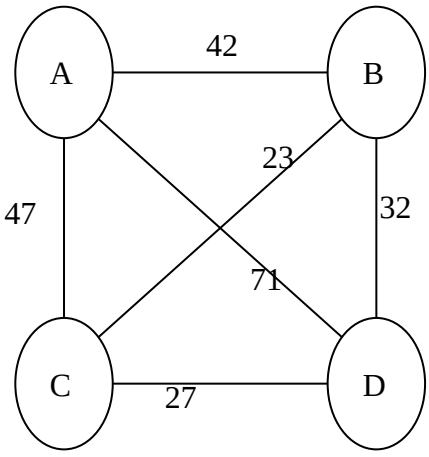

Figure 2.1: Example TSP problem

2.1 Exhaustive Algorithm

The algorithm starts the search from city *A*. At this level it has *n-1* children. It moves to the first child city *B* and travels down adding the distance value at each level. The search continues with the next child of city *B* until all children are explored. The minimum distance value is propagated back to city *B* from its children and then to the root level. At any point, if the obtained distance is lower than the best distance so far, the value is updated. The search then continues with all other children of city *A* and the optimal distance value is updated if it is a new minimum value. Since this search explores all possible city permutations, the final distance is the optimal distance. However, the nature of TSP is that the number of permutations increases exponentially as the problem size increases, making it hard to find the solution using exhaustive search. So, this algorithm is only suitable for finding solutions for small problem sizes. For larger problem sizes, I used parallelization to speed up the search.

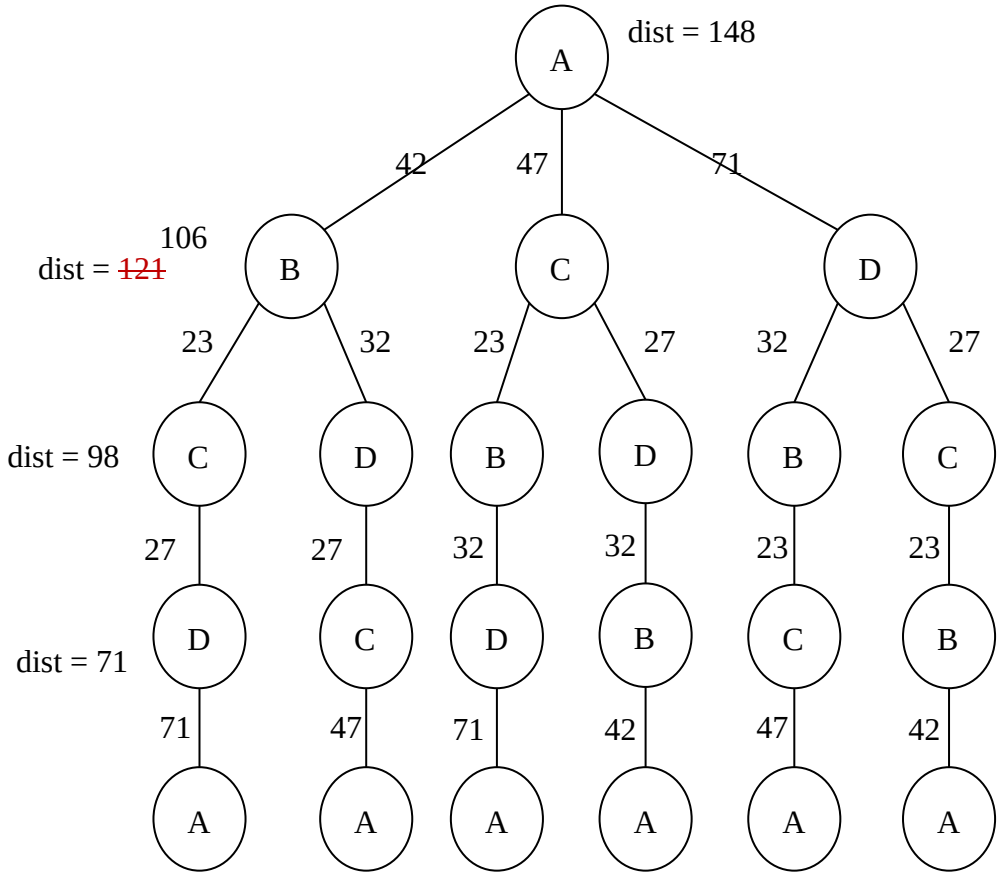

Figure 2.2: Search tree for the example TSP problem

The idea is to assign the search to several threads so that each thread can run it in parallel, and the minimum distance is calculated. But parallelizing at or close to the root of the search tree would not work due to a small number of children present. So, to take advantage of parallelism, we need to travel down the tree to a level with a higher number of children to spawn multiple parallel threads. To achieve this, an alternate exhaustive search approach that uses a modulo operation to enumerate all possible permutations is implemented. This approach launches *(n – 1)!* threads for a problem size with *n* cities. For each permutation, it determines the cities to visit and adds the distance for each visited city. Using this approach, I was able to produce enough parallelism, but it has a

limitation. The required modulo and division operations used to generate the permutations made it slow.

To speed up the approach, the search is stopped at a certain tree depth and the information of the tour such as the distance, unvisited cities, and the last city visited are stored in a worklist. Each element in the worklist is a unique tour path, which is suitable to run in parallel. Threads equal to the total number of elements in the worklist are launched and each of them is assigned a unique path to find the distance. Finally, a global atomicMin operation is performed to determine the shortest distance. This implementation made the TSP solver much faster, but it was still expensive for problem sizes with more than about 14 cities. A further improvement is to exclude parts of the search space that cannot contain the optimal solution. This can be done using the branch and bound technique [26].

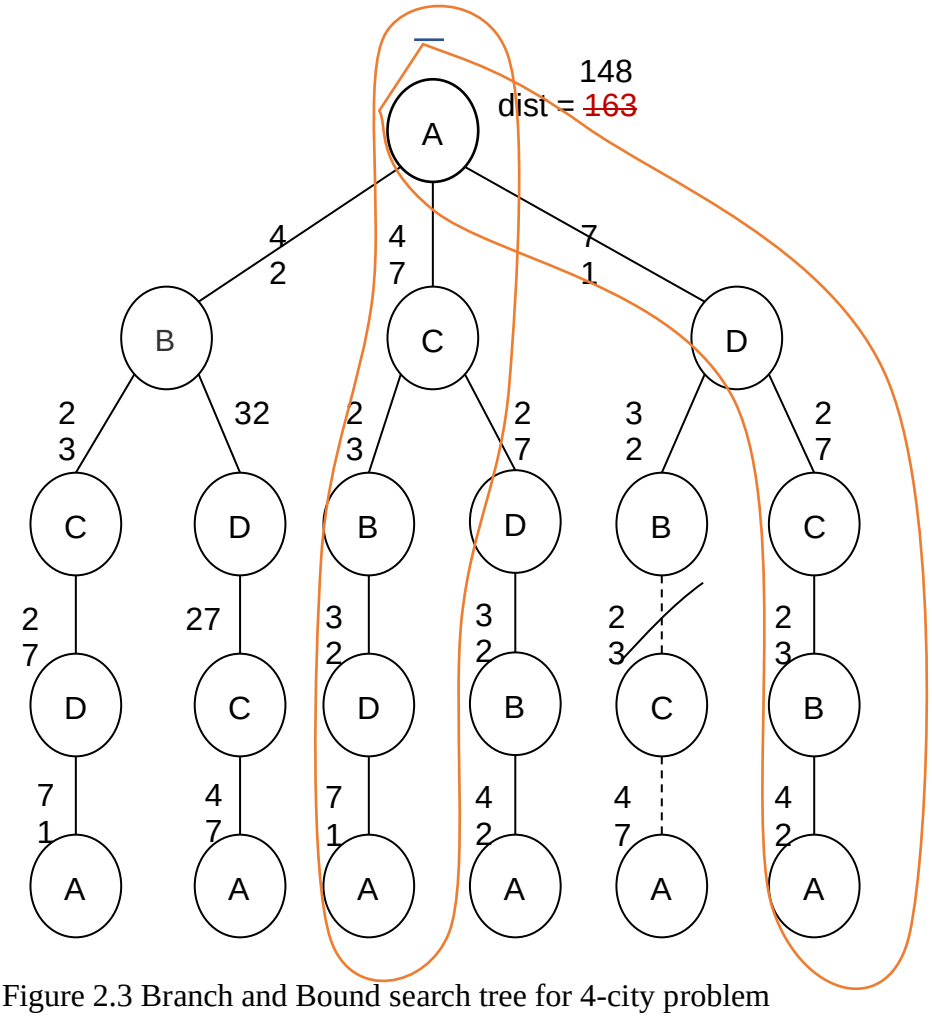

Figure 2.3 Branch and Bound search tree for 4-city problem

City: A B C D

42	23	23	27

Figure 2.4 Lower bound values for 4-city problem

2.2 Branch and Bound Algorithm

Branch and bound is an approach to speed up combinatorial optimization problems. Since TSP is one such problem, limiting the search space using a bounding factor can improve the algorithm for larger problem sizes. Figure 2.3 illustrates the

branch and bound algorithm on the search tree for the 4-city problem shown in Figure 2.1. Figure 2.4 shows the shortest distance from every city to every other city. The lower bound is calculated as the sum of all the shortest distances shown in Figure 2.4. When a city is added to the tour, a new lower bound estimate is calculated by adding the current distance at the level to the remaining shortest distance values of all unvisited cities. If this lower bound estimate is larger than the best distance found so far, the path is pruned as it cannot lead to a better solution. The algorithm calculates the distance of the first complete tour and uses it as the initial best distance. Whenever a better distance is found, this value is updated. For some inputs, the initial distance value calculated may not be close to the optimal distance. This causes the search to travel down many suboptimal paths, which increases the search space of the branch and bound algorithm and lowers performance. The circled paths in Figure 2.3 shows how suboptimal paths are not pruned, causing the branch and bound algorithm to travel down the entire way to the leaf. To avoid this problem and prune the search space more effectively, a heuristic algorithm is executed first. This heuristic algorithm finds a near optimal distance, which is then used as the initial best distance to better prune the search space.

2.3 Performance improvement using heuristics

There are several simple heuristic algorithms to find good quality approximate solutions for TSP. In this implementation I used greedy construction heuristic [27] followed by the 2-opt improvement heuristic [30]. The greedy TSP algorithm takes all the edges in a complete, undirected graph and sorts them in non-decreasing order. The edge with smallest distance is added to the tour if the vertices associated with the edge have degree less than 2 and adding the edge does not form a cycle. Cycles in the greedy tour

path are detected using a disjoint set data structure [28]. This data structure is used to create non-overlapping subsets using two simple operations: union and find. Initially, there is only one vertex in each subset. When an edge is added to the tour, I check if both the vertices of the edge belong to the same subset. If the degree of the vertices is less than 2 and the subsets are different, there is no cycle, and I add the edge to the tour. The source and the destination vertices of the edges that forms the tour is stored in an array and this information is used in improving the solution using the 2-opt heuristic.

The 2-opt heuristic checks if the current tour distance can be decreased and modifies the tour with the best such move if possible. 2-opt removes two edges of the tour generated using the greedy heuristic, splitting the tour into two subtours. One of the subtours is reversed and then again attached to the other subtour. 2-opt attempts this operation for all possible edge pairs and selects the pair that results in the greatest tour-length reduction. Applying this move generates a new (shorter) tour. 2-opt then iteratively tries to shorten the new tour in the same manner until a local minimum is reached. This way, the 2-opt improvement operations further refine the greedy tour. Ultimately, the distance of the final tour is calculated. This distance value is used as the initial distance when launching the exact TSP solver. This approach significantly improves the performance of the branch and bound implementation as the overhead of running the greedy and 2-opt algorithms is much smaller than the time saved in the exact solver.

2.4 GPU Implementation

The full performance of a GPU is unleashed by running many parallel threads that do similar tasks. As discussed above, sufficient parallelism was achieved in the algorithm by traveling down the tree to a level where it has a sufficiently large number of children. Each of these children can be assigned to a unique thread to calculate the tour distance. So, the GPU kernel only needs the total number of permutations, the depth, and the number of cities to launch the threads in parallel. The GPU launch configuration in this implementation was optimized based on the number of nodes at the tree depth of 6. This depth was determined based on several performance experiments. This cut-off depth enables enough parallelism for my GPU implementation. If there is a need for more or for less parallelism on a different computing device, the depth can easily be adjusted accordingly. The thread count per block was tuned based on the number of nodes at the specified depth, the maximum threads per SM, the maximum threads per block and the maximum number of blocks. These values are calculated based on the queried GPU properties before launching the kernel. Each thread starts its tour from city 0 and the next cities in the tour are based on the index of the thread. This makes sure that each thread runs a different permutation. At the set depth, each thread calls the *tsp* device function by passing the details of the tour at that depth level. Information such as unvisited cities, estimated bound, current distance and the city from which the tour should be continued is passed to the *tsp* function. These values are used to continue the tour until all the cities are visited.

Though generating the optimal tour length is the main criterion for TSP solvers, doing it quickly is important in real-world applications. To make the solver more

efficient, I tweaked the performance multiple times and implemented different techniques. Initially, the distance matrix, lower bound value, and lower bound array were using the GPU's global memory. But this caused a strain on the runtime. Since constant memory has its own cache and threads can access it much faster, moving the global memory contents to constant memory resulted in better performance.

Shared memory in a GPU is allocated per thread block and every thread has access to the memory shared by a thread block. This meant shared memory could potentially provide further improvement in performance, so the next experiment was to copy the distance matrix and the lower bound array from constant memory to shared memory. To avoid race conditions and to ensure correct results, the synchronization primitive __syncthreads() is used, which blocks all the threads until the distance matrix and the lower bound array are copied. This change from constant to shared memory proved to be helpful in improving the runtime. To further optimize the algorithm when the number of unvisited cities reaches 3, the possible tour from that level is split into three different functions *tsp3, tsp2* and *tsp1*, which are highly optimized for these specific problem sizes.

3. RELATED WORK

The Traveling Salesman Problem is an interesting area of research due to the different challenges it presents. A lot of research, detailing different approaches, has been done over the years in solving the traveling salesman problem. In 1962, Held-Karp and Bellman proposed an algorithm based on dynamic programming for the classical traveling salesman problem [1]. It is considered the best exact dynamic programming algorithm for TSP to this day. The input for the algorithm is a distance matrix between a set of cities, and the starting city is designated arbitrarily. Set S consists of all cities in the problem excluding the starting city and $l \in S$. The algorithm calculates the shortest one-way path from the starting city to every city in the set S. It then recursively calculates the minimum cost from the starting city to all cities in the set *S* terminating at city *l*. The values are computed starting with the smallest sets of *S* and finishing with the largest. The shortest path is finally reconstructed from the stored values of the second-to-last city on the path from 1 to *l* through *S*.

In 1995, Dyduch [2] parallelized the Branch and Bound (B&B) algorithm for TSP on CPUs. The Branch and Bound method is based on the construction of a decision tree by partitioning the feasible solutions into smaller subsets. The paper parallelizes a sequential B&B for TSP by modifying its branching rule. The sequential version partitions the solution subset into two subsets, one subset with a specific edge (i, j) between the cities and another subset without the specific edge. In the parallel algorithm, the branching rule partitions the subsets of the root and the first level of the decision tree into *n-1* and *n-2* mutually disjoint subsets and the subsets of further levels are partitioned into two subsets, like in the sequential algorithm. The subproblems are assigned

dynamically to free processors. Process 0 determines the basic distribution row (root) as well as the local distribution row (first level) and assigns each processor with these first stage subtree elements. Nodes of the subtree are examined with the use of a backtracking strategy. The second stage subtrees are generated and tested recursively and are assigned to free processors dynamically by process 0.

In 1997, Caseau and Laburthe [3] proposed a set of techniques that made constraint programming a choice for solving the traveling salesman problem with small instances. The paper discusses the local and global constraints that describe TSP. The local constraint is that there can be only one incoming edge and one outgoing edge in a node while the global constraint is that there can be no cycles in the tour. The paper describes a propagation scheme for local constraints by associating each node to its immediate successor. The node that has the largest cost difference with the parent node is chosen for branching. To propagate the no-cycle constraint (global constraint), the start, the end, and the length of the tour going through the starting node are all stored and checked if the sub-chains associated with the successors are already built. The standard branching strategy chooses the most critical variable over all the next available nodes in the tour. Applications of constraint programming to solve TSP have been used in pickup-delivery [7] and parcel delivery with drones [9].

The Concorde TSP solver, an exact method for solving the traveling salesman problem, was written by Applegate in 1998 [4]. It is primarily based on the cutting-plane method for solving the TSP, but this method is believed to be slow even for some small instances. By finding subtour cuts and combining them with branch-and-cut, the cutting plane method can be improved to solve the problem much faster. The quality of a cut

(i.e., the number of edges passing a solution boundary) is based on its contribution in reducing the total running time of the cutting-plane method. The paper proposed a template paradigm for cuts using different separation algorithms. The separation algorithms are exact separation algorithms for subtour cuts, blossom cuts, comb cuts and a greedy heuristic for certain path cuts. As the number of cuts added increases, the increase in finding the optimal value of cutting plane methods LP relaxation decreases. So, branching is used when the increase is too small. The set of tours are partitioned into two subproblems and the cutting-plane method is applied to each. Further, either one of these subproblems or both can be divided into sub-subproblems. In the end, the subproblems will be solved by the cutting-plane method without recourse to branching. The Concorde TSP solver is probably the most widely used exact solver to solve large instances quickly with the largest problems comprising 85,900 cities.

In 1971, Lin and Kernighan developed a highly effective heuristics that produces optimal and near-optimal solutions for the traveling salesman problem [29]. The idea of the heuristic is a generalization of the interchange transformation. A non-optimal but feasible tour *T* is not optimal because it has *k* nodes that are out-of-place, and it can be made optimal by replacing it with a different set of *k* nodes in the tour. The algorithm first identifies the most-out-of-place pair and then finds another pair in the remaining set of the tour to replace it. The algorithm uses a selection rule that chooses the most out-of-place pair. It chooses pairs in which the total gain associated with the proposed set of exchanges is positive. The selection process stops only when the total gain is less than 0 for all *k* out-of-place nodes. The algorithm stops when there is no further profit in searching for out-of-place nodes, i.e., the gain in exchange does not yield any

improvement. When there is no gain found in the nearby nodes, a backtracking function is invoked, which selects replacement nodes with increasing order of length. If all the choices are searched and there is no gain, then the algorithm returns to the node at the other side of the selected edge and continues to find a better replacement. The algorithm backtracks only when no gain can be found and only at levels 1 and 2. It uses a limited backtracking to effectively compromise between exponentiality and running time. Only a maximum of five nodes are checked against each other to see if it can be replaced with the selected out-of-place node. This algorithm considers time taken to reach the first local optimum *T* where no further progress can be made and ignores the time spent on the later cases where it arrives at the same tour. Once the locally optimum tour has been found, the procedure checks if the links of the tour can be broken to yield further improvement. If this process produces an improvement, it may convert the non-optimal solution to an optimal solution. This heuristic developed by Lin-Kernighan is considered to be the most effective heuristic to generate optimal and near-optimal solutions for the traveling salesman problem. However, the implementation of this algorithm is complex, and a decision must be made at each step, most of which influence the performance.

Whereas there are different exact algorithms as described above, there are also heuristic methods to solve the TSP approximately. Heuristic algorithms solve TSP problems by settling on near optimal tours. A lot of research has been done in designing heuristic algorithms for TSP with larger instances [6], one such meta heuristic approach is Ant Colony Optimization. Ant colony optimization was used by Ivan and Zuzana in 2011 [5] to solve small traveling salesman problems. The base of Ant Colony Optimization is to simulate the real behavior of ants. It uses two parameters α and β to

simulate the probability based on pheromone quantity. Every virtual ant has its own memory to save the path it travelled and moves to a new state if no ending constraint is compiled. The next motion depends on the probability calculated based on the pheromone quantity of the edges. The movement of ants provides a parallel and independent search of the route. Based on this approach, the solution quality depends on the number of virtual ants and deviates a little from the optimal solution.

This related work gives us an idea of the techniques that have been used to solve the traveling salesman problem over the years. Despite all this research, finding an optimal solution for TSP is NP-hard and most of the techniques proposed so far are to find a near optimal tour. Also, most of the parallel TSP algorithms for GPUs are based on heuristic approaches and focus only on solving larger instances. My thesis proposes a GPU version for an exact solution approach based on exhaustive search (up to 7 cities) and branch and bound (up to 29 cities).

4. EXPERIMENTAL METHODOLOGY

The primary objective of the TSP solver is to figure the shortest tour distance. This means the optimal length calculated by a TSP solver and the runtime taken to produce this optimal length are the primary metrics for determining the effectiveness of a TSP solver. The input instances used to evaluate the optimal TSP solver are from the standard TSPLIB and the solutions obtained are compared. All the execution times are measured in seconds; in my approach, this time also includes the time taken by the heuristic TSP function for calculating the initial distance used in the exact solver.

The time taken by the implemented TSP solver and the optimal solutions are compared with other TSP solvers to understand where my code stands. The solvers that are used for this comparison are the CONCORDE TSP solver [4] and the LKH TSP solver [29]. CONCORDE and LKH are the best-known solvers available today that produce an optimal solution.

All solvers were evaluated on two systems named “Austin” and “Ithaca”. Below are the specifications of the two systems.

4.1 Austin

- CPU: Two Intel Xeon Gold 6226R CPU @ 2.9 GHz
- Number of Cores: 32 (2x 16-core NUMA)
- Number of Threads: 64
- Main Memory: 64 GB

My CUDA implementation was executed on an NVIDIA GeForce RTX 3090 GPU. The specifications are as follows:

- 82 SMs

- 1536 maximum threads per streaming multiprocessor
- 24 GB memory size
- L1 cache: 128 KB (per SM)
- L2 cache: 6 MB
- CUDA compute capability: 8.6

The CUDA code was compiled on Austin using the CUDA toolkit version in v11.6.124 with the NVCC compiler build version 11.6 with the "-O3 -arch=sm_86" flags.

4.2 Ithaca

- CPU: AMD Ryzen Threadripper 2950X @ 2.7 GHz
- Number of Cores: 16
- Number of Threads: 32
- Main Memory: 64 GB

The CUDA implementation was evaluated on a Titan V GPU. The specification are as follows:

- 80 SMs
- 2048 maximum threads per streaming multiprocessor
- 12 GB memory size
- L1 cache 96 KB (per SM)
- L2 cache: 4.5 MB
- CUDA Compute capability: 7.0

The codes were compiled on Ithaca using the CUDA toolkit version v11.7.64 with the NVCC compiler build version 11.7 with '-O3 -arch=sm_70' flags.

Inputs from the TSPLIB are used as a standard to study TSP and other related problems. The inputs the solver is tested with are from the TSPLIB with fewer than 30 cities, and the same inputs were used for the CONCORDE TSP solver and the LKH TSP solver. My TSP solver can also read the CVRP inputs from the TSPLIB and calculate their shortest path distance.

5. RESULTS

I compared the performance of my TSP solver to that of the CONCORDE TSP solver designed by Applegate in 1998 and the LKH TSP solver designed by Lin-Kernighan in 1971. I measured the runtime for city counts from 7 to 29 and calculated the speedup. The optimal tour length obtained with my TSP solver always matches that obtained from the CONCORDE and LKH solvers. The runtimes were measured on the two machines called “Austin” and “Ithaca” as outlined above. Bolded speedups in the comparisons indicate that my solver is faster. The runtimes do not steadily increase with higher city counts. This is because the runtime is not only a function of the number of cities but also of how the cities are arranged in the input and how close the heuristic solution is to the exact solution, that is, how much of the search space can be pruned.

5.1 Comparison with CONCORDE

This section compares my solver to CONCORDE on the Austin and Ithaca machines. Table 5.1 outlines the results on Ithaca. The highest speedup is achieved for the TSPLIB input eil7, on which my code calculates the shortest distance 21.7 times faster than CONCORDE. Overall, my TSP solver outperforms CONCORDE for problems with up to 15 cities. Figure 5.1 shows the speedup graph for comparison against CONCORDE.

When run on Austin (Table 5.2), a machine with a higher compute capability, my code is up to 60.8 times faster (on eil7). Again, my solver outperforms CONCORDE on problems up to 15 cities as shown in Figure 5.2.

The speedup is higher for smaller problem sizes and drops below one above 15 cities, at which point CONCORDE is faster. The reason is that CONCORDE employs

branch-and-cut and several other advanced algorithms to optimize the solver, which generates additional constraints, thus reducing the search space beyond what my solution is capable of.

Table 5.1: Result comparison with CONCORDE on Ithaca. The runtimes are in seconds.

Input	My Solver	Concorde	Speedup	Optimal Distance
eil7	0.00001	0.00015	**21.71**	66
eil13	0.00028	0.004	**13.90**	142
burma14	0.002	0.010	**4.91**	3323
p01.tsp (15 cities)	0.000	0.001	**3.83**	291
ulysses16	0.494	0.037	0.08	6859
gr17	0.139	0.014	0.10	2085
gr21	0.072	0.005	0.08	2707
ulysses22	3247.388	0.076	0.00	7013
eil22	0.677	0.013	0.02	278
eil23	3.319	0.009	0.00	470
gr24	50.991	0.011	0.00	1272
fri26	484.978	0.015	0.00	937
bays29		0.020		2020
bayg29		0.015		1610
eil30		0.056		381

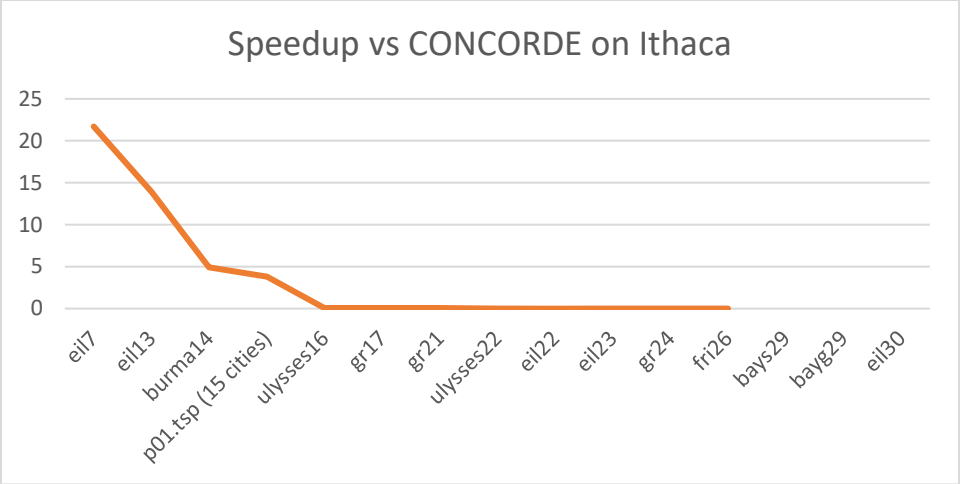

Figure 5.1 Speedup achieved in comparison with CONCORDE on Ithaca

Table 5.2 Result comparison with CONCORDE on Austin. The runtimes are in seconds.

Input	My solver	Concorde	Speedup	Optimal Distance
eil7.vrp	0.000006	0.000365	**60.83**	66
eil13.vrp	0.000192	0.007	**36.46**	142
burma14	0.001	0.016	**13.29**	3323
p01.tsp (15 cities)	0.000	0.004	**16.95**	291
ulysses16	0.294	0.027	0.09	6859
gr17	0.097	0.009	0.09	2085
gr21	0.051	0.008	0.16	2707
ulysses22	2137.547	0.069	0.00	7013
eil22.vrp	0.456	0.015	0.03	278
eil23.vrp	2.198	0.005	0.00	470
gr24	33.540	0.020	0.00	1272
fri26	322.594	0.011	0.00	937
bays29		0.011		2020
bayg29		0.022		1610
eil30.vrp		0.040		381

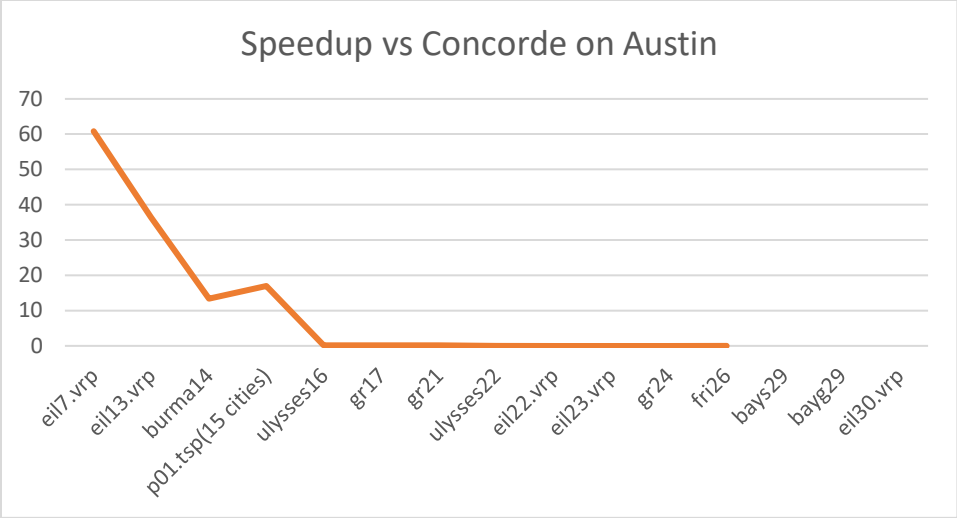

Figure 5.2 Speedup achieved in comparison with CONCORDE on Austin

5.2 Comparison with LKH

The following data show the results of the comparison between my TSP solver and LKH. Table 5.3 denotes the comparison on Ithaca. Like CONCORDE, the LKH solver performs better for problem sizes above 15 cities. Figure 5.3 shows the speedup over the LKH solver on Ithaca. The highest speedup is for the input eil7, where my solver is 82 times faster.

Table 5.4 lists the results of all tested TSPLIB instances on Austin. Although the LKH solver performs better for higher problem sizes, due to the faster GPU, the performance benefit of my GPU-based solver is twice as high compared to Ithaca and outperforms LKH by a larger factor on small problem sizes. Figure 5.4 shows the speedup achieved against the LKH solver on Austin. The highest speedup is 234 when solving the instance eil7.

Table 5.3 Result comparison with LKH on Ithaca. The runtimes are in seconds.

Input	**My Solver**	**LKH**	**Speedup**	**Optimal Distance**
eil7	0.00001	0.001408	82.29	66
eil13	0.00028	0.003	7.94	142
burma14	0.002	0.003	0.62	3323
p01.tsp (15 cities)	0.0003	0.001	1.73	291
ulysses16	0.494	0.003	0.00	6859
gr17	0.139	0.003	0.01	2085
gr21	0.072	0.003	0.03	2707
ulysses22	3247.388	0.004	0.00	7013
eil22	0.677	0.007	0.01	278
eil23	3.319	0.003	0.00	470
gr24	50.991	0.007	0.00	1272
fri26	484.978	0.005	0.00	937
bays29		0.006		2020
bayg29		0.011		1610
eil30		0.009		381

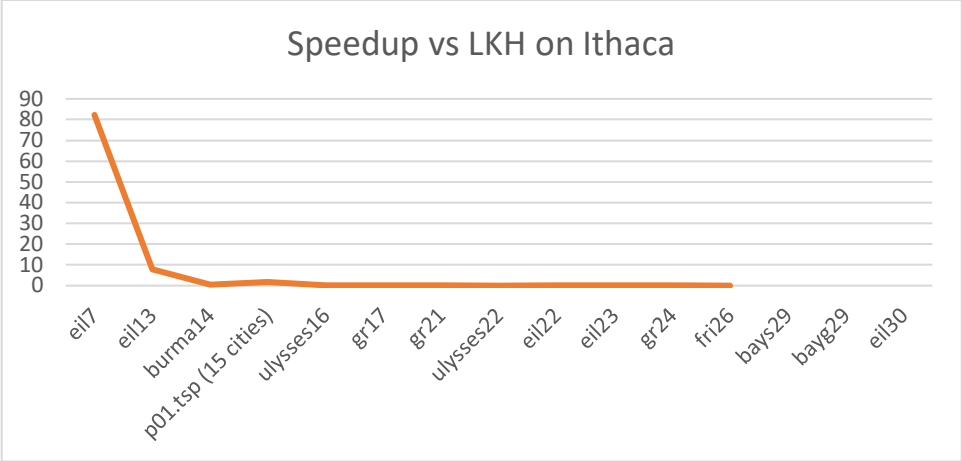

Figure 5.3 Speedup achieved in comparison with LKH on Ithaca

Table 5.4: Result comparison with LKH on Austin. The runtimes are in seconds.

Input	My solver	LKH	Speedup	Optimal distance
eil7.vrp	0.000006	0.001408	**234.67**	66
eil13.vrp	0.000192	0.003	**15.63**	142
burma14	0.001	0.003	**2.49**	3323
p01.tsp (15 cities)	0.000	0.001	**4.24**	291
ulysses16	0.294	0.003	0.01	6859
gr17	0.097	0.003	0.03	2085
gr21	0.051	0.003	0.06	2707
ulysses22	2137.547	0.004	0.00	7013
eil22.vrp	0.456	0.007	0.02	278
eil23.vrp	2.198	0.003	0.00	470
gr24	33.540	0.007	0.00	1272
fri26	322.594	0.005	0.00	937
bays29		0.006		2020
bayg29		0.011		1610
eil30.vrp		0.009		381

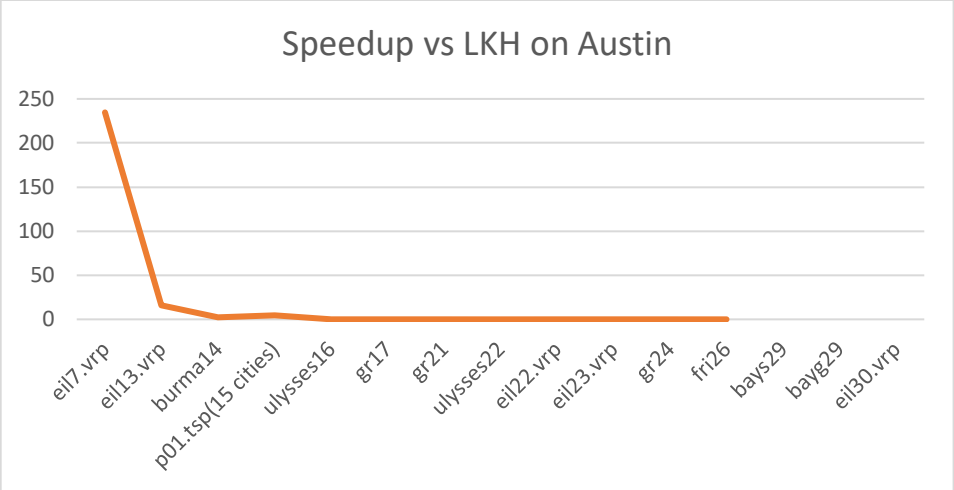

Figure 5.4 Speedup achieved in comparison with LKH on Austin

6. SUMMARY AND CONCLUSION

With the recent expansion in delivery services, solving the Traveling Salesman Problem to its optimality has gained attention. An important challenge is to find the most efficient tour very quickly. Especially the last mile challenges faced in recent delivery systems has increased the importance of finding optimal solutions. But the NP-hardness of TSP makes this difficult. This thesis proposes a CUDA implementation of TSP that is based on the simple branch and bound algorithm and exhaustive enumeration.

My TSP solver is a hybrid solver that utilizes both the CPU and the GPU. Maximum performance is achieved by running small problems with up to 7 cities on the CPU using exhaustive search and larger problems on the GPU using a branch and bound algorithm. The search starts at city *A* and visits the remaining cities; at each step the minimum distance value is propagated to the previous level. This guarantees to find the optimal solution by exploring all possible paths but is only efficient for very small problem sizes. To solve larger problems, parallelization is used to improve performance.

To parallelize the algorithm effectively and run it on a GPU, I implemented a modulo approach to travel down a few levels of the search tree until a sufficiently larger number of children is reached. At each level, the distance value is added and at level 6 the distance value and information about the current city are stored in a worklist. This approach produces enough parallelism but the use of modulo operations is expensive. To further improve the algorithm, I used branch and bound to eliminate parts of the search space that cannot yield an optimal solution.

The branch and bound algorithm limits the search space using a bounding factor. This lower bound is calculated by adding all the shortest distances from each city to every other city. At each level, the best distance obtained so far is compared with this lower bound estimate and the path is pruned if the minimally estimated distance is larger. To further improve the algorithm, I used a greedy construction heuristic and the 2-opt improvement heuristic to find a good initial guess to prune the search space. This approach improved the performance substantially.

To further boost the performance on the GPU, the launch configuration in the implementation was optimized based on the number of child nodes at depth 6. The distance matrix was copied to the shared memory. The thread count per block was tuned based on the number of children, maximum threads per SM, maximum threads per blocks and the maximum number of blocks. This improves the utilization of the GPU.

In conclusion, I implemented a GPU-accelerated exact solver for small TSP problem sizes and compared its performance with the two state-of-the-art solvers CONCORDE and LKH. My GPU solver, which uses simple algorithms, outperforms both CONCORDE and LKH for up to 15 cities and can solve problems with up to 26 cities in a reasonable amount of time. Future exploration would be to use more sophisticated heuristics (e.g., genetic algorithms) to find a better near-optimal solution to prune the search space and to improve the performance on larger problem sizes using more advanced algorithms.

BIBLIOGRAPHY

1. Held, Michael, and Richard M. Karp. “A Dynamic Programming Approach to Sequencing Problems.” Journal of the Society for Industrial and Applied Mathematics 10, no. 1 (1962): 196–210. http://www.jstor.org/stable/2098806.
2. Ewa Dudek-Dyducb, Tadeusz Dyducb, Travelling Salesman Problem - Parallel Algorithms, IFAC Proceedings Volumes, Volume 28, Issue 10, 1995, Pages 657-662, ISSN 1474-6670,
3. Caseau, Yves & Laburthe, François. (1997). Solving Small TSPs with Constraints. 316-330.
4. Applegate, David L., Robert E. Bixby and William J. Cook. “On the Solution of Traveling Salesman Problems.” (1998).
5. Brezina, Ivan & Čičková, Zuzana. (2011). Solving the Travelling Salesman Problem Using the Ant Colony Optimization. International Scientific Journal of Management Information Systems. 6.
6. Burtscher, Martin. "A high-speed 2-opt tsp solver for large problem sizes." (2013).
7. O'Neil, Ryan J. and Karla L. Hoffman. “Exact Methods for Solving Traveling Salesman Problems with Pickup and Delivery in Real Time.” (2018).
8. Nwamae, Believe B, Kabari, Ledisi G., 0, Solving Travelling Salesman Problem (TSP) Using Ant Colony Optimization (ACO), International Journal Of Engineering Research & Technology (IJERT) Volume 07, Issue 07 (July 2018)
9. Roberti, Roberto & Ruthmair, Mario. (2019). Exact Methods for the Traveling Salesman Problem with Drone.

10. Sara Cavani, Manuel Iori, Roberto Roberti, Exact methods for the traveling salesman problem with multiple drones, Transportation Research Part C: Emerging Technologies.

11. Nedjatia, Arman and B'ela Vizv'arib. “Robot Path Planning by Traveling Salesman Problem with Circle Neighborhood: modeling, algorithm, and applications.” arXiv: Optimization and Control (2020)

12. V. Burkhovetskiy and B. Steinberg. 2017. An exact parallel algorithm for traveling salesman problem. In Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17). Association for Computing Machinery, New York, NY, USA, Article 14, 1–5. https://doi.org/10.1145/3166094.3166108.

13. TSPLIB. RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

14. N. Thakoor, V. Devarajan and J. Gao, "Computation complexity of branch-and-bound model selection," *2009 IEEE 12th International Conference on Computer Vision*, 2009, pp. 1895-1900, doi: 10.1109/ICCV.2009.5459420.

15. Parallel Branch and Bound Algorithm - A comparison between serial, OpenMP and MPI implementations Lucio Barreto and Michael Bauer 2010 *J. Phys.: Conf. Ser.* **256** 012018

16. Blair Archibald, Patrick Maier, Ciaran McCreesh, Robert Stewart, Phil Trinder, Replicable parallel branch and bound search, Journal of Parallel and Distributed Computing, Volume 113, 2018, Pages 92-114, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2017.10.010.

17. R. A. Palhares and M. C. B. AraÚjo, "Vehicle Routing: Application of Travelling Salesman Problem in a Dairy," 2018 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2018, pp. 1421-1425, doi: 10.1109/IEEM.2018.8607472.

18. Eldos, Taisir & Kanan, Aws & Aljumah, Abdullah. (2013). Solving The Printed Circuit Board Drilling Problem By Ant Colony Optimization Algorithm. Lecture Notes in Engineering and Computer Science. 1. 584-588.

19. Matai, Rajesh & Singh, Surya & Mittal, M.L... (2010). Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches. 10.5772/12909.

20. Jonathan E. Tito, Marco E. Yacelga, Martha C. Paredes, Andres J. Utreras, Waldemar Wójcik, Olga Ussatova, "Solution of travelling salesman problem applied to Wireless Sensor Networks (WSN) through the MST and B&B methods," Proc. SPIE 10808, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018, 108082F (1 October 2018); https://doi.org/10.1117/12.2501579

21. P Tipurić, Darko [Editor:] Hruška, Domagoj [Title:] 7th International OFEL Conference on Governance, Management and Entrepreneurship: Embracing Diversity in Organisations. April 5th - 6th, 2019, Dubrovnik, Croatia [Pages:] 391-401

22. P, Rajarajeswari & D, Maheswari. (2020). Travelling Salesman Problem Using Branch And Bound Technique. International Journal of Mathematics Trends and Technology. 66. 202-206. 10.14445/22315373/IJMTT-V66I5P528.

23. Vali, Masoumeh and Khodakaram Salimifard. “A Constraint Programming Approach for Solving Multiple Traveling Salesman Problem.” (2017).

24. Focacci, Filippo et al. “A Hybrid Exact Algorithm for the TSPTW.” *INFORMS J. Comput.* 14 (2002): 403-417.

25. O'Neil, Molly A. and Martin Burtscher. “Rethinking the parallelization of random-restart hill climbing: a case study in optimizing a 2-opt TSP solver for GPU execution.” *Proceedings of the 8th Workshop on General Purpose Processing using GPUs* (2015): n.

26. Kianfar, Kiavash. (2011). Branch-and-Bound Algorithms. 10.1002/9780470400531.eorms0116.

27. Nilsson, Christian. (2003). Heuristics for the Traveling Salesman Problem.

28. Zvi Galil and Giuseppe F. Italiano. 1991. Data structures and algorithms for disjoint set union problems. ACM Compute. Surv.23, 3 (Sept. 1991), 319–344.

29. Effective Heuristic Algorithm for The Traveling-Salesman Problem. / Lin, S.; Kernighan, B. W. In: Operations Research, Vol. 21, No. 2, 1973, p. 498-516.

30. O'Neil, Molly A. et al. “A Parallel GPU Version of the Traveling Salesman Problem.” (2011).

31. Data For Traveling Salesman problem https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html