INTEGRATION, the VLSI journal 102 (2025) 102333

Contents lists available at ScienceDirect

Integration, the VLSI Journal

journal homepage: www.elsevier.com/locate/vlsi

CompreCity: Accelerating the Traveling Salesman Problem on GPU with data compression

Salih Yalcin [a], Hamdi Burak Usul [b], Gulay Yalcin [a,*]

[a] Abdullah Gul University, Kayseri, Turkey
[b] Technische Universitaet Braunschweig, Braunschweig, Germany

ARTICLE INFO

Keywords:
Traveling Salesman Problem
Graphical Processing Unit (GPU) programming
Iterated Local Search
Compute Unified Device Architecture (CUDA)

ABSTRACT

Traveling Salesman Problem (TSP) is one of the significant problems in computer science which tries to find the shortest path for a salesman who needs to visit a set of cities and it is involved in many computing problems such as networks, genome analysis, logistics etc. Using parallel executing paradigms, especially GPUs, is appealing in order to reduce the problem solving time of TSP. One of the main issues in GPUs is to have limited GPU memory which would not be enough for the entire data. Therefore, transferring data from the host device would reduce the performance in execution time. In this study, we applied three data compression methodologies to represent cities in the TSP such as (1) Using Greatest Common Divisor (2) Shift Cities to the Origin (3) Splitting Surface to Grids. Therefore, we include more cities in GPU memory and reduce the number of data transfers from the host device. We implement our methodology in Iterated Local Search (ILS) algorithm with 2-opt and The Lin–Kernighan–Helsgaun (LKH) Algorithm. We show that our implementation presents more than 25% performance improvement for both algorithms.

1. Introduction

Traveling Salesman Problem (TSP) is one of the significant problems in computer science which tries to find the shortest path for a salesman who needs to start from an initial city, visit a set of cities by stopping by at each of them exactly once and return back to the initial city at the end [2]. This problem is involved in many real-life issues like drilling of printed circuit boards, computer wiring, design of global navigation satellite system surveying networks, genome analysis, logistics and many more [3]. Thus, finding an optimal solution for a TSP in a feasible amount of time is an essential and also challenging concern.

TSP can be represented as finding the Hamiltonian cycle in an N-vertices weighted graph (for N cities) in which cities are represented by vertices and edges are showing the distances between cities.[1] In addition, if the distances between two cities in both directions are identical, the number of solutions is halved and it is called as a symmetric TSP which can be expressed by an undirected graph.

It is trivial that finding the exact solution for TSP with a pure brute force algorithm, in which the length of each possible tour is calculated, requires factorial time (i.e. for a selected initial city among n cities, there will be n-1 options for the second city and n-2 options for the third city and so on). Therefore, TSP is classified as an NP-hard problem in which finding the exact solution is not possible in polynomial time [4]. Due to the difficulty of finding the exact solution, other problem solving approaches utilizing heuristics are implemented to find an acceptable solution in an acceptable time such as genetic algorithm [5–7], ant algorithm [8–11], tabu search [12,13], neural network [14] etc. which are discussed in Section 2.1. We also explain the details of Iterated Local Search (ILS) algorithm [15] and the 2-opt algorithm in Section 2.1.1 which we use for our evaluations.

Although several algorithmic methods are implemented to find an optimum solution for TSP, it is still time-consuming for an average microprocessor especially when the problem size is relatively large like

* Corresponding author.
E-mail addresses: salih.yalcin@agu.edu.tr (S. Yalcin), h.usul@tu-braunschweig.de (H.B. Usul), gulay.yalcin@agu.edu.tr (G. Yalcin).

[1] Let G = (V, E) be a complete undirected graph, where: $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices representing cities, with $| v | = n$, and, $E = \{(V_i, V_j) \mid V_i, V_j \in V, i \neq j\}$ is the set of edges connecting every pair of cities. The distance $d(v_i, v_j)$ between any two vertices v_i and v_i is given by the Euclidean distance formula: $d(v_i, v_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. The objective of the Symmetric Traveling Salesman Problem (STSP) is to find a Hamiltonian cycle that minimizes the total travel distance. In another word, the problem is to find a permutation of the vertex set V, such that the total distance traveled is minimized [1].

https://doi.org/10.1016/j.vlsi.2024.102333
Received 10 July 2024; Received in revised form 14 October 2024; Accepted 6 December 2024
Available online 13 December 2024

in many real-life applications. In order to reduce the execution time of those algorithms, it is possible to use parallel execution paradigms provided in the hardware level such as multiple threads in CPUs, Field Programmable Gate Arrays (FPGAs) or Graphical Processing Units (GPUs). Those hardware improvements can provide great amount of performance efficiency in the execution time which is discussed in Sections 2.2 and 2.3.

Graphical Processing Units are used to increase the performance of applications by using Data Level Parallelism in which different small execution units operate on different portion of the data with the same instruction. GPUs are first developed for graphical operations specifically in the game industry in which performance is important to make graphical operations faster. For instance, computation of each pixel in an image is done in a different execution unit, a.k.a a processing element in a streaming multi-processor. Due to their data-parallel feature, GPUs have also been used in many different areas besides the game industry such as machine learning applications, bioinformatics and blockchain applications. Applications can be programmed by using CUDA (Compute Unified Device Architecture) [16] to be able to distribute the data parallel sections to the processing elements. It is shown that the speedup provided by GPUs can be tremendously high since many processing elements operates in parallel while they are more power efficient due to the simplicity of each processing element compared to a CPU. For instance, it is claimed that Hopper, NVIDIA's newest GPU architecture, can provide 7x Dynamic Programming Performance[2] with its new instruction set named DPX. Thus, Hopper DPX instructions will speed up optimization algorithms up to 40 times [17]. Although GPUs are used to reduce the execution time of TSP in several studies [4,18–20] (which are discussed in Section 2.4), we believe there is room for further improvement by efficient memory usage.

One of the major limitations in GPUs, like in all high-performance computing systems, is memory wall which hinders achieving potential speedup provided by the highly parallel architecture due to the fact that many computing units are requesting data simultaneously. Therefore, the shared-memory area in GPUs is limited and not enough to fit all data of the TSP.

In this study, our goal is to provide a methodology for compressing data to represent cities in the TSP in order to reduce the memory usage of GPUs and fit more cities in the shared memory area. To this end, we represent each city with smaller numbers(i.e. 16-bit numbers) instead of 32-bit integer numbers after applying three main steps. Firstly, we shift cities to (0,0) center in the coordinate system to avoid big numbers and negative numbers. Secondly, we find Greatest Common Divisor (GCD) for both the X and Y coordinates of all cities and divide the coordinates of each city to those GCD values. Thirdly, we split the entire area into grids and represent each city with respect to the base coordinate of the grid. We present the details of our design in Section 3. Our implementation presents 29% performance improvement compared to [4], the state-of-the-art GPU implementation of the TSP.

The contribution of this study is as following:

- We present a comprehensive review on hardware methods (i.e. FPGAs, Multi-threads and GPUs) used to speed up the TSP solving algorithms.
- We present a data compression methodology for representing cities to reduce the memory usage of GPUs
- We implement our methodology and show that our GPU implementation, on average, presents 29% performance improvement compared to the state-of-art GPU implementation.

In our evaluations, we use Iterated Local Search (ILS) algorithm [15] with 2-opt [21] for solving TSP in our implementation as in [4,18]. Yet, our compression algorithm is orthogonal to any heuristics applied in the algorithmic level.

[2] Dynamic Programming breaks complex problems down to simpler subproblems that are solved recursively, therefore, it reduces complexity and time to polynomial scale.

2. Related work

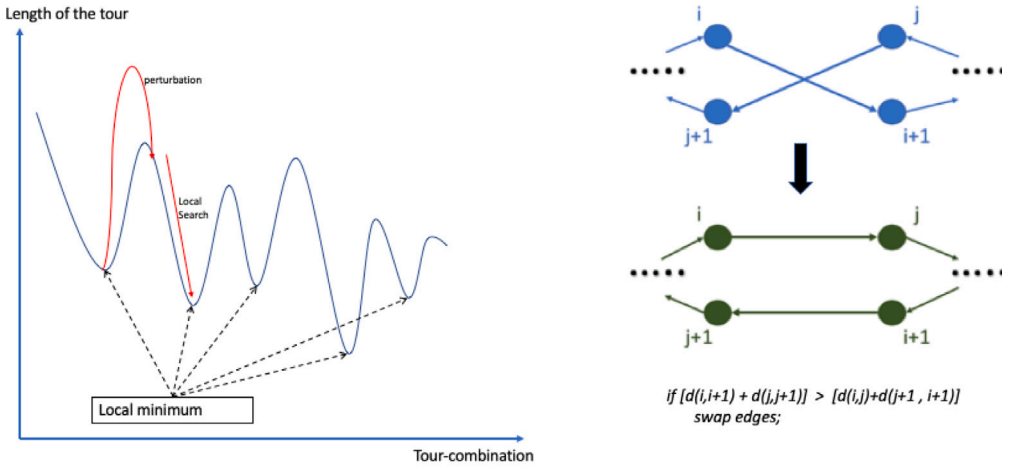
In this section, we present a literature survey about the studies addressing to solve TSP faster. First, we present the list of algorithmic methods using different heuristics and the details of Iterated Local Search algorithm that we also use in this study. Then we present methods leveraging parallel architectures to improve the performance of TSP solving methods.

2.1. Algorithm-based methods

TSP is a widely studied problem in the field of optimization. Over the years, many algorithm-based methods have been developed to solve this problem.

Ant Colony (ACO) is inspired by the behavior of real ant colonies, where ants communicate with each other by depositing pheromones to mark the shortest path to food sources. In ACO, a similar pheromone-based mechanism is used to construct candidate solutions to the TSP. The algorithm iteratively builds a set of solutions by simulating the behavior of ants that follow paths with higher pheromone levels, while also exploring new paths with a certain probability [8–11,22,23]. Artificial Bee Colony emulates the foraging behavior of honeybees, and its effectiveness has been demonstrated in solving TSP instances of varying magnitudes [24–28]. Cuckoo Search is a metaheuristic algorithm that is inspired by the reproductive behavior of cuckoo birds. It is a population-based algorithm that uses Levy flights to explore the search space and has been shown to be effective in finding near-optimal solutions for TSP instances of varying sizes [23,29–31]. Differential Evolution Algorithm works by evolving a population of candidate solutions through the use of mutation, crossover, and selection operations [32]. Firefly Algorithm simulates the flashing behavior of fireflies and their attraction to brighter ones to find optimal solutions for TSP [33,34]. Genetic Algorithm is a heuristic optimization algorithm that simulates the natural selection process in genetics which has been widely used to solve TSP [5–7,20,35–37]. Particle Swarm Optimization is a well-known meta-heuristic optimization algorithm inspired by the social behavior of bird flocks and fish schools which has been applied to solve various optimization problems including TSP [38,39]. Simulated Annealing mimics the physical annealing process of metals by iteratively adjusting the temperature and searching for lower energy states and it is applied to solve TSP [40,41]. Water Cycle Algorithm is based on the movement of water droplets from higher to lower potential energy, which simulates the flow of water in the water cycle process. It can handle the constraints of TSP such as visiting all cities only once and returning to the starting point [42]. Deep Reinforcement Learning (DRL) is a relatively new approach to solving optimization problems, including the TSP. DRL models use neural networks to learn from experience and make decisions that lead to the best solution [14]. DNA computing has been applied to the Generalized TSP (GTSP), using parallel biochemical reactions to achieve reduced computational complexity, down to $O(n^2)$ [43].

Each of these approaches has benefits and drawbacks. For instance, the Ant Colony algorithm can find a good solution to the TSP problem in a relatively short amount of time while the Genetic Algorithm can find a solution closer to the optimal one. Overall, the selection of an algorithm-based method for solving TSP depends on various factors such as the size of the problem, the required level of accuracy, the available computational resources, and the time constraints. In this study, we use Iterated Local Search with 2-opt that we explain its details in the next section.



(a) Iterated Local Search (b) Perturbation

Fig. 1. In Fig. 1(a), we present how ILS solves TSP problem. When the algorithm finds a local minimum, it perturbates the city list and starts searching for another local minimum. The algorithm stops searching when the defined iteration number (N) is reached. In Fig. 1(b), we present perturbation of 2-opt. For all the cities in the travel list, the perturbation algorithm finds the best (i,j) combination to perturbate.

2.1.1. Iterated Local Search (ILS) and the Lin–Kernighan–Helsgaun (LKH) algorithms

In this section we explain the details of ILS with 2-opt and LKH algorithms that we used in this study. In Fig. 1(a), we present how ILS finds a solution for the TSP problem. First, ILS generates an initial solution (i.e. a tour which represents cities in an array with their visiting order). Then, it makes a local search on close combinations and finds the local minimum value. Then, it perturbates the current local minimum value by hoping there is a better configuration with a shorter path in another local minimum value. After the perturbation, ILS searches again for the next local minimum value in the modified solution. The algorithm for ILS can be represented as:

```
Generate an initial solution S
Apply a local search to S and find local best S'
Repeat N times:
   Perturbate S' and find T
   Apply local search and find T'
   if T' is better than S':
     S' = T'
```

Here when we increase the number of repetition N, the final solution will be closer to the optimum one.

2-opt, on the other hand, is a local search algorithm that iteratively swaps two edges in a TSP tour to obtain a better solution. The basic idea behind 2-opt is that if two edges in a tour cross over each other, then swapping the endpoints of those edges will result in a shorter tour (see Fig. 1(b)). Therefore, each pair of edges are swapped if after the swap operation, the distance would be shorter. More precisely, while i and j are indices of cities in a tour and $d(k,n)$ donating the distance between two cities, local search algorithm can be given as following:

```
For each pair of edges (i,i+1) and (j,j+1)
  if d(i,i+1) + d(j,j+1) > d(i,j+1) + d(j,i+1)
       swap j and i+1 cities in the tour
```

This process is repeated until no further improvements can be made meaning when the local minimum is reached. 2-opt is computationally efficient and can often improve the quality of a solution significantly.

On the other side, the Lin–Kernighan–Helsgaun (LKH) algorithm is a highly efficient heuristic method for solving the TSP [44]. The algorithm builds upon the Lin–Kernighan (LK) heuristic, which is based on local search techniques, by improving the way tours are generated and refined. It extends the basic idea of 2-opt and 3-opt moves (swapping segments of a tour to find shorter paths) and incorporates more complex k-opt moves, where "k" can dynamically vary based on the local topology of the solution. The Helsgaun extension improves the original LK algorithm by introducing more sophisticated search strategies, such as the use of candidate sets, which limit the number of neighbors considered for each city, and probabilistic sampling to diversify search paths. This results in a more robust performance, particularly for large instances of TSP, allowing for more rapid convergence toward optimal or near-optimal solutions. The LKH algorithm is widely recognized for providing state-of-the-art results in practical applications, combining efficient local search with advanced tour management techniques [44].

2.2. Multi-threaded methods

In Multi-threaded execution, algorithms are divided into threads using programming methods such as OpenMP [45] and Pthread [46], and these threads are executed in parallel in the hardware. This mechanism is called as Thread Level Parallelism in which different threads may have different instruction sequences. During the execution, multiple threads can be scheduled to the same core as in Simultaneous Multi-Threading (SMT) [47] or the threads can be executed in different cores as in Chip Multiprocessing (CMP) [48]. According to Amdahl's Law [49], in which it is presented that the minimum execution time can be as low as $Sequential\_Execution\_Time/N$ when N threads are used and the application can be fully parallelized. Compared to GPU, it is easier to divide algorithm into CPU threads since each thread can operate on different execution sequence, however, the potential speedup provided by GPUs is higher since number of parallel execution units (N) is much higher in GPUs. Multi-threaded execution is used to reduce the execution time of TSP in several studies by using PThread [19] and OpenMP [18,50,51].

2.3. FPGA methods

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. FPGA is used to increase the performance of TSP in FPGA [52,53]. Despite the good performance improvement of FPGAs, their programming and hardware overhead is considered as high, since the FPGA configuration is generally specified using a hardware description language (HDL).

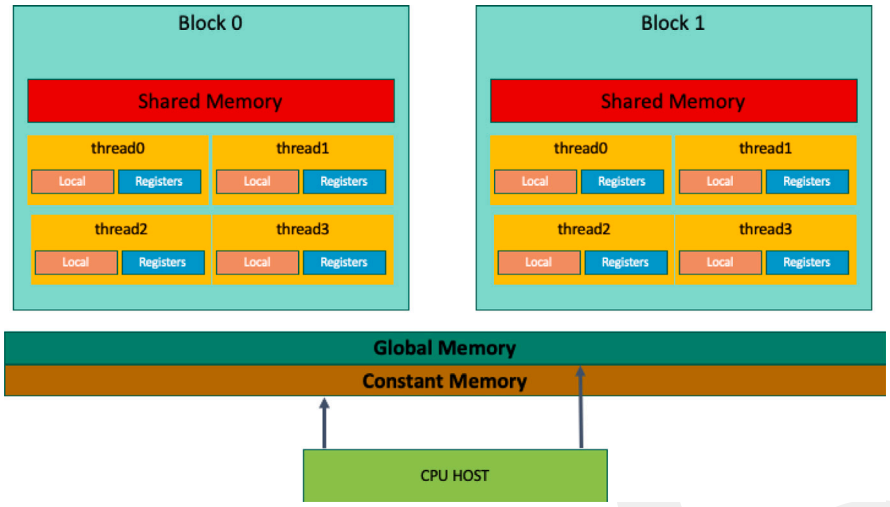


Fig. 2. Memory spaces in GPU.

2.4. GPU methods

Graphical Processing Units are used to increase the performance of applications by using Data Level Parallelism in which different small execution units operate on different portion of the data with the same instruction. GPUs are first developed for graphical operations specifically in the game industry in which performance is important to make graphical operations faster. For instance, computation of each pixel in an image is done in a different execution unit, a.k.a a processing element in a streaming multi-processor. However, due to their data-parallel feature, GPUs have also been used in many different areas besides the game industry such as machine learning applications, bioinformatics and blockchain applications. Applications can be programmed by using CUDA (Compute Unified Device Architecture) [16] to be able to distribute the data parallel sections to the processing elements. It is shown that the speedup provided by GPUs can be tremendously high since many processing elements operates in parallel while they are more power efficient due to the simplicity of processing elements compared to CPUs.

GPUs are used to reduce the execution time of TSP in several studies [4,18–20]. Chen et al. converted a TSP solution based on genetic algorithms into a parallel code to be executed in Fermini GPU [20]. Unfortunately, their experiments did not present a significant performance improvement on GPU. In [19], authors parallelized the Iterative Hill Climbing algorithm (a.k.a iterative local search) with 2-opt for both CPU and GPU. In their results, a GPU with 448 cores can have a similar performance with a CPU using 256 threads (a pthread implementation on 32 CPUs with 8 cores each). However, the maximum problem size in the study is limited to 110 cities due to 48 kB of shared memory limit in the GPU.

Although GPUs present a high computation power, their memory is limited as it can be seen both from [19,20]. Therefore Rocki and Suda [4] show that re-computation can be faster than reading it from the memory and implemented 2-opt and 3-opt iterated local search solution in GPU. In their algorithm, in stead of accessing pre-calculated distances between cities from the global memory, only coordinates are stored in the limited shared memory of GPU and distances are re-calculated.

O'Neil and Burtscher [18] also stored only coordinates in the share memory of GPU. Moreover, they applied tiling to reduce the number of accesses to the global memory by optimizing memory access patterns. The tiling process involves breaking down a computation into smaller tasks that fit within the GPU's available memory resources. These tasks are then executed in a loop, where each iteration operates on a tile-sized portion of the data. Tiling offers several benefits such that enhancing spatial and temporal data locality by reducing the need for frequent memory accesses and, better memory coalescing . However, it requires careful consideration of tile size, as a balance must be struck between reducing memory latency and minimizing synchronization overhead.

The base algorithm that we utilized in this study is based on Iterated Local Search Algorithm with 2-opt and it applies tiling while making calculations on GPU. Thus our data compression method is orthogonal to tiling.

3. CompreCity: Data compression for TSP

Graphical Processing Units (GPUs) may provide a substantial speedup due to high number of parallel execution units included in GPUs. One of the main limitation hindering to achieve the highest speedup is the memory requirement of the parallel executions for many algorithms. In order to reduce the memory access time in GPU, CUDA supports several low-capacity, high-performance memories to keep the data closer to the execution cores rather than accessing DRAM slowly.

In Fig. 2, we present memory spaces in GPUs. Global memory is relatively slower memory in GPU (it is still faster than DRAM) which can be accessed by all running threads as well as the host CPU. Global memory is allocated and deallocated by the host by using `cudaMalloc`, `cudaFree`, `cudaMemcp` and `cudaMemset` commands. Shared Memory is very fast (it can be in the speed of a register) compared to global memory and it can be accessed by all the threads in the same block.

In this study, our goal is; while solving Traveling Salesman Problem in GPU, to be able to fit as much data as possible in the GPU memory, possibly in the shared memory, so that the DRAM access is reduced and the overall performance is improved. To this end, we apply three-step compression to the input file to represent cities with less number of bits. In this section, we present these compression steps.

3.1. Using greatest common divisor

In many data analysis and visualization tasks, it is common to encounter datasets with varying scales. When dealing with datasets that have coordinates, such as those in the Cartesian plane, it can be useful to scale the values to a smaller scale. In this study, we proposed using Greatest Common Divisor (GCD) to shrink the numbers in x-coordinates and y-coordinates. The GCD can be calculated using a simple algorithm, and once obtained, it can be used to scale the x and y coordinates to a common factor. This ensures that the dataset is scaled proportionally and that the relationships between the points in the

Table 1
The table demonstrates an example of how coordinates are compressed after applying GCD and Shift operation to a map with five cities.

Index	x	y
1	515 725	507 650
2	520 000	507 650
3	507 725	507 650
4	512 000	507 650
5	203 875	506 925
	GCD_x:	25
	GCD_y:	25
	X_{MAX}:	507 650
	Y_{MAX}:	520 000

$\xrightarrow{GCD}$

Index	x	y
1	20 629	20 306
2	20 800	20 306
3	20 309	20 306
4	20 480	20 306
5	8155	20 277
	X_{MIN}:	8155
	Y_{MIN}:	20 277
	X_{MAX}:	20 800
	Y_{MAX}:	20 306

$\xrightarrow{Shifting}$

Index	x	y
1	12 474	29
2	12 645	29
3	12 154	29
4	12 325	29
5	0	0
	X_{MIN}:	0
	Y_{MIN}:	0
	X_{MAX}:	12 645
	Y_{MAX}:	29

dataset are preserved. By using the GCD for scaling, it becomes easier to analyze and visualize datasets with varying scales, and to compare datasets with one another. In our design, we followed this approach and computed the GCD of the x-coordinates and y-coordinates separately for our dataset. Then, we divided each x-coordinate and y-coordinate by their respective GCD values. This scaling process ensures that our dataset was proportionally scaled, and the relationships between the points were preserved.

In Table 1, we demonstrate how GCD is applied for a map with five cities. In the example, the first table represents the raw data which has the (x,y) coordinates of cities. In our compression algorithm, we first find the GCD of all x coordinates and the GCD of all y coordinates. In the given example both GCD_x and GCD_y are calculated as 25. Then, each x coordinate is divided to GCD_x and each y coordinate is divided to GCD_y. The values after applying gcd is presented in the second table. In the example, in the row data, the maximum value for x coordinates was 507 650 which can be represented by 19 digits in binary while after applying GCD, the new maximum value becomes 20 800 which can be represented by 15 digits.

In GPU, after applying GCD method to all coordinates, the host needs to pass GCD_x and GCD_y values to GPU together with city coordinates. Note that the overhead of these two GCD values are negligible compared to the size of entire city coordinates.

3.2. Shift city

The second technique that we used for data scaling is shifting. Shifting involves finding the minimum x-coordinate and y-coordinate values in a dataset and subtracting them from all the x-coordinates and y-coordinates in the dataset. This process moves the entire dataset so that the minimum x-coordinate and y-coordinate values becomes the origin of the coordinate system. By shifting, the entire picture and the distances between the cities do not change. Instead, the entire map shifts in the coordinate system to the origin and the values become smaller. Also note that the minimum values in the row data can be a negative value and shifting helps eliminate negative values in the coordinates and we can use unsigned integers to represent cities.

In our analysis, we applied the shifting technique by computing the minimum x-coordinate and y-coordinate values in our dataset and subtracting them from all the x-coordinates and y-coordinates. This approach ensured that our dataset was centered at the origin of the coordinate system, making it easier to interpret and analyze. By using both GCD scaling and shifting techniques, we created a more standardized and comparable dataset for our analysis.

In Table 1, in the third table, we showed the data values after applying the shift operation. In the example, the minimum value for x in the second table was 8155 while the minimum value for y was 20 277 that subtract those values from the x coordinates and y coordinates of all the cities respectively. After the shifting operation, the minimum values for the x and y coordinates become zero. In the example, the maximum x value becomes 12 645 which can be represented by 14 bits in binary (it was 15 bits in the second table) while the maximum value for y becomes 29 which needs only 5 bits which is a substantial drop in the data size compared to 15 bits in the second table.

3.3. Splitting surface to grids

In addition to GCD and shifting techniques, another important aspect of our data pre-processing is the splitting of surfaces into grids and referencing each city relatively to the cell that it locates. This technique involves dividing a surface into a set of smaller regions, or grids, to enable more granular analysis and modeling. In our analysis, we split our surface into a set of 16×16 grids, with each grid assigned a unique number ranging from 0 to 255, from left to right. To determine the range for each grid, we used a simple formula that divided the difference between the maximum and minimum x-coordinate values by 16. This approach ensured that each grid covered an equal range of x-coordinate values. We applied a similar formula to determine the range for each grid in the y-coordinate values. By splitting our surface into grids, we were able to have more scalable and compressed data.

In Fig. 3, we present an example for representing a city after splitting the surface into grids. In the example, we assume that the maximum x value is 4000 and the maximum y value is 16 000 after the shift operation and the surface is divided into 4×4 grid for simplicity. The city A, which is located in cell 10, has the coordinates of 2100 and 10 500 in the example. After the splitting operation, we now represent cities with three components such as ($X_{relative}$, $Y_{relative}$, $cellNumber$). In the example, the base coordinates of cell 10 is (2000,8000) which can be calculated as;

$$(\frac{x_{max}}{4} \times [(int)(10/4)], \frac{y_{max}}{4} \times (10 \bmod 4))$$

New coordinates of A after applying the Split Surface technique is calculated as subtracting the base coordinates of the cell from the coordinates of A. Also, we add the cell number to the coordinates. Therefore, A's new coordinate becomes (100, 2500, 10).

Here, we reduce the number of bits required to represent each coordinate. However, we also add the cell number which increases the number of bits in each coordinate. Nevertheless, each coordinate of a city is represented with less number of bits in total and overall data usage is reduced.

4. Evaluations

In this section, we first present detailed explanation about the benchmarks that we use, then we present our experimental results for our CompreCity method.

4.1. Benchmarks

In our evaluation, we use 7 benchmark data which are as followings:

fnl4461: This dataset contains 4461 cities in Finland. The goal is to find the shortest route that visits each city exactly once and returns to the starting point. The optimal solution for this problem is known to have a total distance of 182,566 units.

rl5915: This instance contains 5915 cities and is named after Gerhard Reinelt, a researcher who contributed significantly to the study of TSP problems. The cities are distributed in a Euclidean 2D plane, and the goal is to minimize the total distance for a tour visiting each

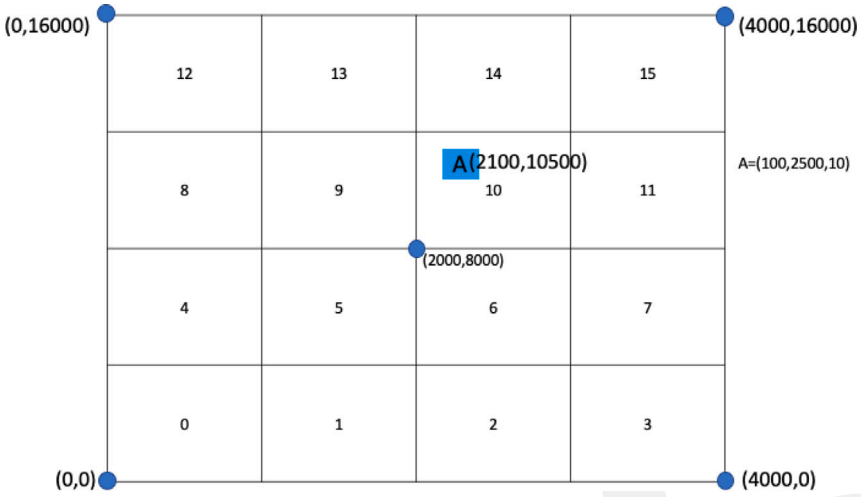
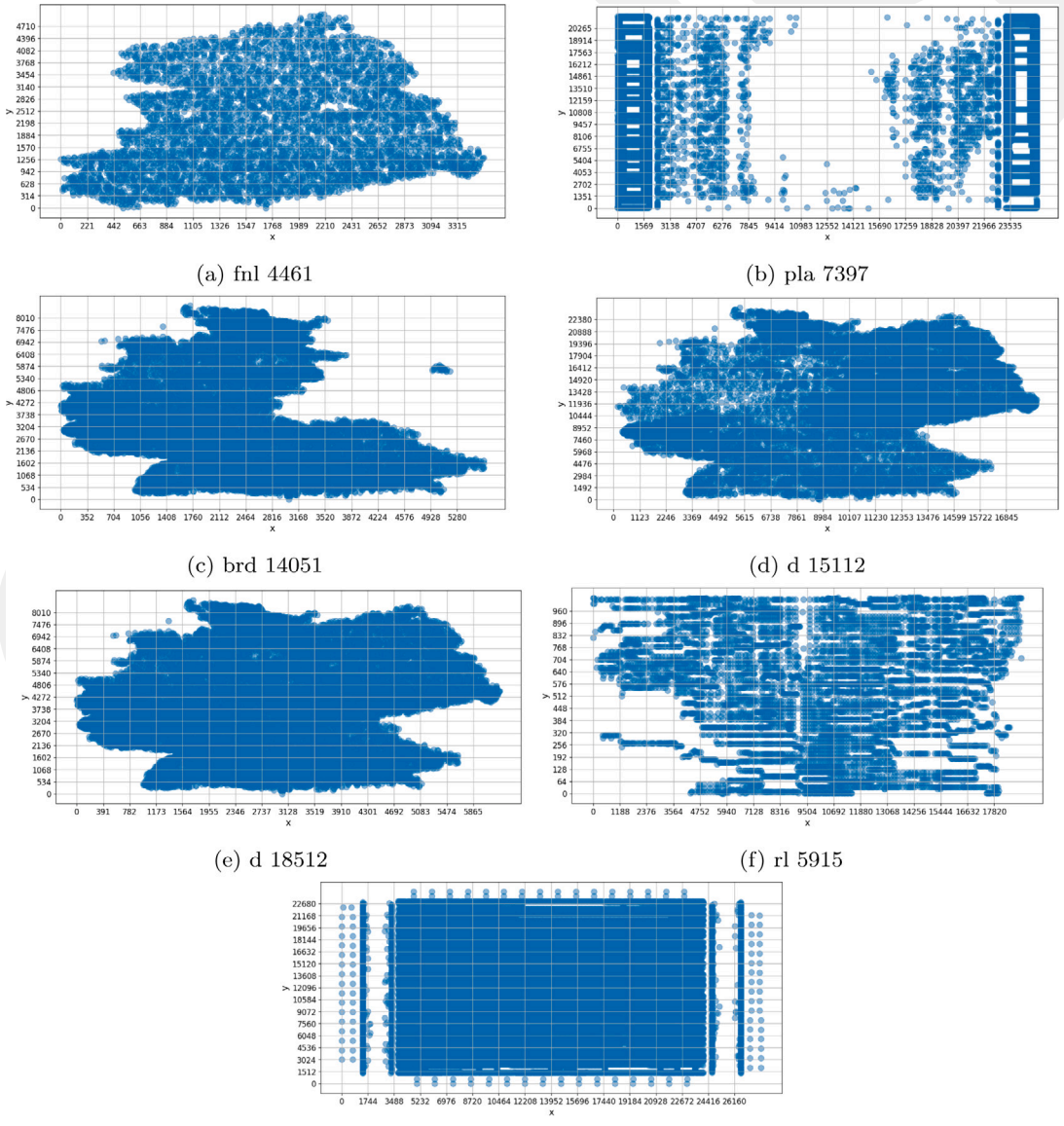


Fig. 3. Example of splitting surface to grids.



(a) fnl 4461

(b) pla 7397

(c) brd 14051

(d) d 15112

(e) d 18512

(f) rl 5915

(g) pla 33810

Fig. 4. Maps of benchmark cities.

Table 2
The table shows maximum and minimum values of X and Y coordinates after applying each technique. GCD value for each input is also shown.

Benchmark	gcd_x	gcd_y	Initially				After GCD				After Shifting				After CompreCity			
			X_{MIN}	X_{MAX}	Y_{MIN}	Y_{MAX}	X_{MIN}	X_{MAX}	Y_{MIN}	Y_{MAX}	X_{MIN}	X_{MAX}	Y_{MIN}	Y_{MAX}	X_{MIN}	X_{MAX}	Y_{MIN}	Y_{MAX}
fnl_4461	1	1	5639	9176	5648	10 675	5639	9176	5648	10 675	0	3537	0	5027	0	222	0	317
pla_7397	25	25	0	627 925	0	540 725	0	25 117	0	21 629	0	25 117	0	21 629	0	1582	0	1364
brd_14051	1	1	2918	8555	2407	10 966	2918	8555	2407	10 966	0	5637	0	8559	0	357	0	549
d_15112	1	1	168	18 148	0	23 878	0	18 148	0	23 878	168	17 980	0	23 878	0	1135	0	1498
d_18512	1	1	2918	9176	2407	10 966	2918	9176	2407	10 966	0	6258	0	8559	0	393	0	549
rl_5915	1000	100	11 200 000	1 912 200 000	20 900 000	1 155 000 000	11 200	1 912 200	209 000	11 550 000	0	1 901 000	0	11 341 000	0	1190	0	71
pla_33810	25	25	0	697 900	0	604 900	0	27 916	0	24 196	0	27 916	0	24 196	0	1756	0	1516

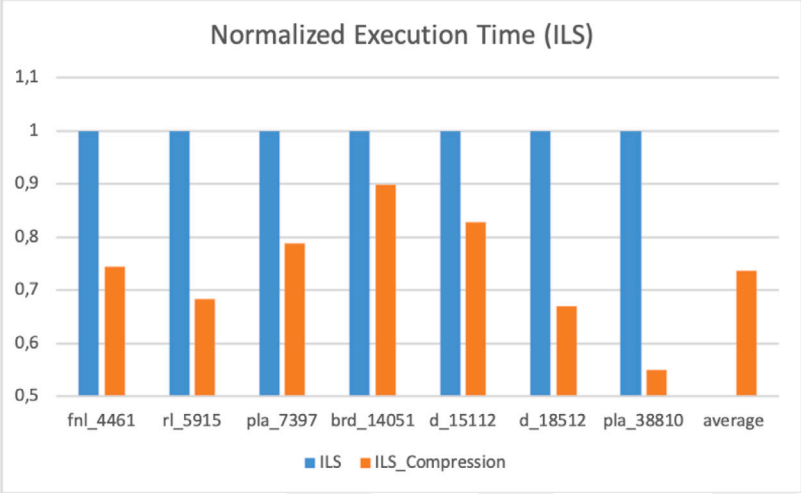


Fig. 5. Normalized execution Time of TSP with ILS Algorithm. Each execution is normalized with non-compressed version.

city exactly once and returning to the start. This dataset is part of the TSPLIB library and is commonly used to benchmark algorithms.

pla7397: This instance involves 7397 cities placed randomly in the plane. The problem belongs to the category of Euclidean TSP, where the distances between cities are based on Euclidean geometry. The best-known solution has a total distance of 23,260,728 units.

pla33810: This dataset involves 33,810 cities randomly distributed in the plane, similar to pla7397. The best-known solution lies between 65,960,739 and 66,116,530 units.

brd14051: This dataset contains 14,051 cities in Berlin, and it is part of the TSPLIB benchmark library. The optimal solution is in the range of 469,272 to 469,445 units.

d15112: The d15112 dataset consists of 15,112 cities. The best-known solution for this problem lies in the range of 1,572,863 to 1,573,152 units.

d18512: This problem involves 18,512 cities. The best-known solution for this TSP instance is approximately 645,092 to 645,300 units.

4.2. Experimental results

In Fig. 4, we present maps of each benchmark together with 16×16 grid on each map. In Table 2, for each map, we present maximum and minimum values for x and y coordinates. For instance, for fnl, the minimum X coordinate is 5639 while the maximum X coordinate is 9176. Also, in the first column, we present the GCD of all X and Y values. The GCD for pla is 25 for both X and Y values while for rl it is 1000 for x and 100 for y. For the rest, it is 1 for both X and Y. Therefore, we can apply GCD reduction only for pla and rl. After applying GCD, the maximum X value becomes 25 117 (which was 627 925) and the maximum Y value becomes 21 629 (which was 540 725) for pla_7397.

In the next step, we apply shift operation to cities which makes the minimum X and Y values as zero and it reduces maximum X and Y values relatively. As it can be seen on the table, for instance maximum Y value of fnl becomes 5027 which was 10 675 in the row data. Finally, we apply CompreCity method and we present the maximum X and Y values in the last part of Table 2. After the CompreCity operation, the maximum value for X, Y coordinates become 1756 which requires only 11 bits to represent. So, we use int16 data value to represent X and Y coordinates instead of 32-bit integer values. However, for each coordinate, we also need to add cell number which can be represented with an 8 bit integer value. Thus, representation of one city is shrunk to 40 bits instead of 64 bits which provides 38% reduction in data size and that many more cities can be fit into the GPU memory area.

In Figs. 5 and 6, we compare the execution time of the ILS algorithm with 2-opt and LKH algorithm respectively to solve TSP. According to our results, compressing city coordinates can reduce the execution time more than 15% for all input maps and on average 29% performance improvement is gained in the execution time. From these figures, we can make following observations. First, our compression methodology presents similar improvement in performance independent from the algorithm. It is because these algorithms are already optimized to minimized the memory accesses. Second, the performance improvement of our methodology is higher in pla_38810. It is because we can apply gcd and also pla has a high number of cities which fit in the shared memory. Third, performance improvement for smaller maps is limited. It is because those maps can mostly fit in the shared memory without compression.

In summary, the benefit of our compression methodology is higher when two following cases occur: (1) the number of cities are high such that shared memory is not enough to keep all of it. (2) the compressed data is small enough to mostly fit in the shared memory.

5. Conclusion

In this study, we focused on improving the performance of solving Traveling Salesman Problem (TSP) in GPUs in order to reduce the total

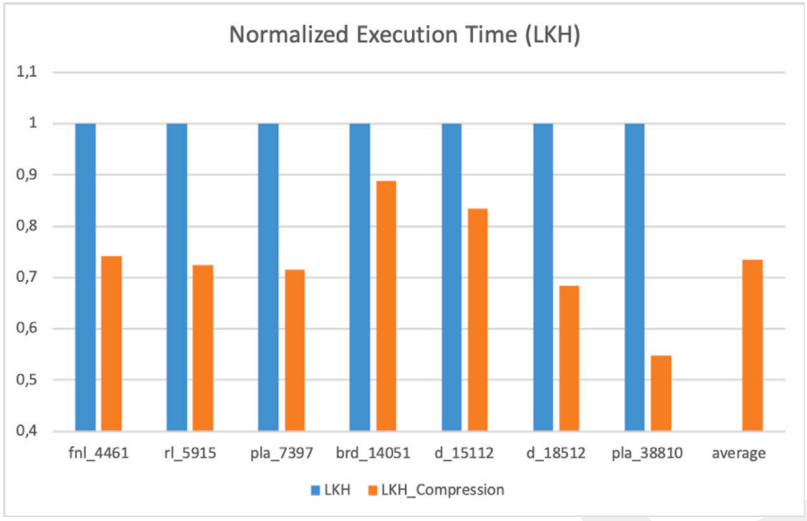


Fig. 6. Normalized execution Time of TSP with LKH algorithm. Each execution is normalized with non-compressed version.

data transfer time from CPU to GPU. We proposed using data compression techniques to represent cities and we present three schemes named: (1) Using GCD, (2) Shifting to the origin (3) CompreCity. We implement our methodology in Iterated Local Search (ILS) algorithm with 2-opt and our results indicate that our implementation presents 29% performance improvement compared to the state-of-the-art GPU implementation. As a future study, different compression methods can be applied on different algorithms (ie. k-opt algorithms or deep learning methods) in order to analyze the performance gain of proposed scheme.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] M. Grötschel, M. Padberg, On the symmetric travelling salesman problem: Theory and computation, 1978, http://dx.doi.org/10.1007/978-3-642-95322-4_12.
[2] G. Reinelt, TSPLIB–A traveling salesman problem library, ORSA J. Comput. 3 (4) (1991) 376–384.
[3] R. Matai, S. Singh, M.L. Mittal, Traveling salesman problem: an overview of applications, formulations, and solution approaches, in: D. Davendra (Ed.), Traveling Salesman Problem, 2010.
[4] K. Rocki, R. Suda, Accelerating 2-opt and 3-opt local search using GPU in the travelling salesman problem, in: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012), 2012, pp. 705–706, http://dx.doi.org/10.1109/CCGrid.2012.133.
[5] J.J. Grefenstette, R. Gopal, B.J. Rosmaita, D.V. Gucht, Genetic algorithms for the traveling salesman problem, in: International Conference on Genetic Algorithms, 1985.
[6] I.M. Oliver, D.J. Smith, J.R.C. Holland, A study of permutation crossover operators on the traveling salesman problem, in: Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and their Application, L. Erlbaum Associates Inc., USA, ISBN: 0805801588, 1987, pp. 224–230.
[7] A. Hussain, Y.s. Muhammad, N. Sajid, I. Hussain, A. Shoukry, S. Gani, Genetic algorithm for traveling salesman problem with modified cycle crossover operator, Comput. Intell. Neurosci. 2017 (2017) http://dx.doi.org/10.1155/2017/7430125.
[8] M. Dorigo, L.M. Gambardella, Ant colonies for the travelling salesman problem, Biosystems (ISSN: 0303-2647) 43 (2) (1997) 73–81, http://dx.doi.org/10.1016/S0303-2647(97)01708-5, URL https://www.sciencedirect.com/science/article/pii/S0303264797017085.
[9] B. Yu, Z.-Z. Yang, B. Yao, An improved ant colony optimization for vehicle routing problem, European J. Oper. Res. 196 (1) (2009) 171–176, URL https://ideas.repec.org/a/eee/ejores/v196y2009i1p171-176.html.
[10] Y. Wang, Z. Han, Ant colony optimization for traveling salesman problem based on parameters optimization, Appl. Soft Comput. (ISSN: 1568-4946) 107 (2021) 107439, http://dx.doi.org/10.1016/j.asoc.2021.107439, URL https://www.sciencedirect.com/science/article/pii/S1568494621003628.
[11] R. Skinderowicz, Improving ant colony optimization efficiency for solving large TSP instances, Appl. Soft Comput. 120 (2022) 108653, http://dx.doi.org/10.1016/j.asoc.2022.108653.
[12] S. Basu, Tabu search implementation on traveling salesman problem and its variations: A literature survey, Am. J. Oper. Res. 02 (2012) http://dx.doi.org/10.4236/ajor.2012.22019.
[13] H. Li, B. Alidaee, Tabu search for solving the black-and-white travelling salesman problem, J. Oper. Res. Soc. 67 (8) (2016) 1061–1079, URL https://EconPapers.repec.org/RePEc:pal:jorsoc:v:67:y:2016:i:8:d:10.1057_jors.2015.122.
[14] P. da Costa, J. Rhuggenaath, Y. Zhang, A. Akcay, U. Kaymak, Learning 2-Opt heuristics for routing problems via deep reinforcement learning, SN Comput. Sci. (ISSN: 2661-8907) 2 (5) (2021) 388, http://dx.doi.org/10.1007/s42979-021-00779-2.
[15] H.R. Lourenço, O.C. Martin, T. Stützle, Iterated local search, in: F. Glover, G.A. Kochenberger (Eds.), Handbook of Metaheuristics, Springer US, Boston, MA, ISBN: 978-0-306-48056-0, 2003, pp. 320–353, http://dx.doi.org/10.1007/0-306-48056-5_11.
[16] Nvidia, P. Vingelmann, F.H. Fitzek, CUDA, release: 10.2.89, 2020, URL https://developer.nvidia.com/cuda-toolkit.
[17] J. Huang, Nvidia, GTC sept 2022 keynote with NVIDIA CEO jensen huang, 2022, URL https://www.youtube.com/watch?v=PWcNlRI00jo.
[18] M.A. O'Neil, M. Burtscher, Rethinking the parallelization of random-restart hill climbing: A case study in optimizing a 2-Opt TSP solver for GPU execution, in: Proceedings of the 8th Workshop on General Purpose Processing using GPUs, in: GPGPU-8, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450334075, 2015, pp. 99–108, http://dx.doi.org/10.1145/2716282.2716287.
[19] M.A. O'Neil, D.E. Tamir, M. Burtscher, A parallel GPU version of the traveling salesman problem, 2011.
[20] S. Chen, S. Davis, H. Jiang, A. Novobilski, CUDA-based genetic algorithm on traveling salesman problem, in: R. Lee (Ed.), Computer and Information Science 2011, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-642-21378-6, 2011, pp. 241–252.
[21] G.A. Croes, A method for solving traveling-salesman problems, Oper. Res. 6 (6) (1958) 791–812, http://dx.doi.org/10.1287/opre.6.6.791, arXiv:https://doi.org/10.1287/opre.6.6.791.
[22] Y. Zhou, F. He, N. Hou, Y. Qiu, Parallel ant colony optimization on multi-core SIMD CPUs, Future Gener. Comput. Syst. (ISSN: 0167-739X) 79 (2018) 473–487, http://dx.doi.org/10.1016/j.future.2017.09.073, URL https://www.sciencedirect.com/science/article/pii/S0167739X16304289.
[23] L. Salal Hasan, Solving traveling salesman problem using cuckoo search and ant colony algorithms, J. Al-Qadisiyah Comput. Sci. Math. 10 (2) (2018) http://dx.doi.org/10.29304/jqcm.2018.10.2.377, pp. Comp Page 59 – 64. URL http://qu.edu.iq/journalcm/index.php/journalcm/article/view/377.
[24] D. Karaboga, B. Basturk, Artificial Bee Colony (ABC) optimization algorithm for solving constrained optimization problems, Vol. 4529, 2007, pp. 789–798. http://dx.doi.org/10.1007/978-3-540-72950-1_77.

[25] L. Li, Y. Cheng, L. Tan, B. Niu, A discrete artificial bee colony algorithm for TSP problem, in: Proceedings of the 7th International Conference on Intelligent Computing: Bio-Inspired Computing and Applications, ICIC '11, Springer-Verlag, Berlin, Heidelberg, ISBN: 9783642245527, 2011, pp. 566–573, http://dx.doi.org/10.1007/978-3-642-24553-4_75.

[26] B. Akay, 2-Opt based artificial bee colony algorithm for solving traveling salesman problem, 2011, pp. 666–672.

[27] Jiang, Solving traveling salesman problem using artificial bee colony algorithm, in: Computer Science and Technology, 2017, pp. 989–995, http://dx.doi.org/10.1142/9789813146426_0110, arXiv:https://www.worldscientific.com/doi/pdf/10.1142/9789813146426_0110 URL https://www.worldscientific.com/doi/abs/10.1142/9789813146426_0110.

[28] X. Dong, Q. Lin, M. Xu, Y. Cai, Artificial bee colony algorithm with generating neighbourhood solution for large scale coloured traveling salesman problem, IET Intell. Transp. Syst. 13 (10) (2019) 1483–1491, http://dx.doi.org/10.1049/iet-its.2018.5359, arXiv:https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-its.2018.5359 URL https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-its.2018.5359.

[29] X.-S. Yang, S. Deb, Cuckoo search via Levy flights, 2010, arXiv:1003.1594.

[30] Y. Zhou, X. Ouyang, J. Xie, A discrete cuckoo search algorithm for travelling salesman problem, Int. J. Collab. Intell. 1 (1) (2014) 68–84, http://dx.doi.org/10.1504/IJCI.2014.064853, arXiv:https://www.inderscienceonline.com/doi/pdf/10.1504/IJCI.2014.064853 URL https://www.inderscienceonline.com/doi/abs/10.1504/IJCI.2014.064853, PMID: 64853.

[31] Z. Zhang, J. Yang, A discrete cuckoo search algorithm for traveling salesman problem and its application in cutting path optimization, Comput. Ind. Eng. (ISSN: 0360-8352) 169 (2022) 108157, http://dx.doi.org/10.1016/j.cie.2022.108157, URL https://www.sciencedirect.com/science/article/pii/S0360835222002273.

[32] I. Zelinka, R. Senkerik, M. Bialic-Davendra, D. Davendra, Chaos driven evolutionary algorithm for the Traveling Salesman Problem, in: D. Davendra (Ed.), Traveling Salesman Problem, IntechOpen, Rijeka, 2010, http://dx.doi.org/10.5772/13107.

[33] X.-S. Yang, Firefly algorithms for multimodal optimization, 2010, arXiv:1003.1466.

[34] M. bo Wang, Q. Fu, N. Tong, M. Li, Y. Zhao, An improved firefly algorithm for traveling salesman problems, in: Proceedings of the 2015 4th National Conference on Electrical, Electronics and Computer Engineering, Atlantis Press, ISBN: 978-94-6252-150-6, 2015/12, pp. 1085–1092, http://dx.doi.org/10.2991/nceece-15.2016.193.

[35] A. Homaifar, S. Guan, G.E. Liepins, A new approach on the traveling salesman problem by genetic algorithms, in: International Conference on Genetic Algorithms, 1993.

[36] G. Sena, G. Isern, D. Megherbi, Implementation of a parallel genetic algorithm on a cluster of workstations: The travelling salesman problem, a case study, in: J. Rolim, F. Mueller, A.Y. Zomaya, F. Ercal, S. Olariu, B. Ravindran, J. Gustafsson, H. Takada, R. Olsson, L.V. Kale, P. Beckman, M. Haines, H. ElGindy, D. Caromel, S. Chaumette, G. Fox, Y. Pan, K. Li, T. Yang, G. Chiola, G. Conte, L.V. Mancini, D. Méry, B. Sanders, D. Bhatt, V. Prasanna (Eds.), Parallel and Distributed Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-540-48932-0, 1999, pp. 266–274.

[37] L.-Z. Tan, Y.-Y. Tan, G.-X. Yun, C. Zhang, An improved genetic algorithm based on k-means clustering for solving traveling salesman problem, in: Computer Science, Technology and Application, 2016, pp. 334–343, http://dx.doi.org/10.1142/9789813200449_0042, arXiv:https://www.worldscientific.com/doi/pdf/10.1142/9789813200449_0042 URL https://www.worldscientific.com/doi/abs/10.1142/9789813200449_0042.

[38] K.-P. Wang, L. Huang, C.-G. Zhou, W. Pang, Particle swarm optimization for traveling salesman problem, in: Proceedings of the 2003 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.03EX693), Vol. 3, 2003, pp. 1583–1585, http://dx.doi.org/10.1109/ICMLC.2003.1259748, Vol.3.

[39] X. Shi, Y. Liang, H. Lee, C. Lu, Q. Wang, Particle swarm optimization-based algorithms for TSP and generalized TSP, Inform. Process. Lett. (ISSN: 0020-0190) 103 (5) (2007) 169–176, http://dx.doi.org/10.1016/j.ipl.2007.03.010, URL https://www.sciencedirect.com/science/article/pii/S0020019007000804.

[40] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680, http://dx.doi.org/10.1126/science.220.4598.671, arXiv:https://www.science.org/doi/pdf/10.1126/science.220.4598.671 URL https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[41] L. Xiong, S. Li, Solving TSP based on the improved simulated annealing algorithm with sequential access restrictions, in: Proceedings of the 2016 6th International Conference on Mechatronics, Computer and Education Informationization (MCEI 2016), Atlantis Press, ISBN: 978-94-6252-282-4, 2016/12, pp. 610–616, http://dx.doi.org/10.2991/mcei-16.2016.127.

[42] E. Osaba, J.D. Ser, A. Sadollah, M.N. Bilbao, D. Camacho, A discrete water cycle algorithm for solving the symmetric and asymmetric traveling salesman problem, Appl. Soft Comput. (ISSN: 1568-4946) 71 (2018) 277–290, http://dx.doi.org/10.1016/j.asoc.2018.06.047, URL https://www.sciencedirect.com/science/article/pii/S1568494618303818.

[43] X. Ren, X. Wang, Z. Wang, T. Wu, Parallel DNA algorithms of generalized traveling salesman problem-based bioinspired computing model, Int. J. Comput. Intell. Syst. 14 (2020) 228–237.

[44] K. Helsgaun, An effective implementation of the Lin–Kernighan traveling salesman heuristic, European J. Oper. Res. (ISSN: 0377-2217) 126 (1) (2000) 106–130, http://dx.doi.org/10.1016/S0377-2217(99)00284-2.

[45] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, Parallel Programming in OpenMP, Morgan kaufmann, 2001.

[46] B. Nichols, D. Buttlar, J.P. Farrell, Pthreads Programming - A POSIX Standard for Better Multiprocessing, O'Reilly, ISBN: 978-1-56592-115-3, 1996, I–XVI, 1–267.

[47] D. Tullsen, S. Eggers, H. Levy, Simultaneous multithreading: Maximizing on-chip parallelism, in: Proceedings 22nd Annual International Symposium on Computer Architecture, 1995, pp. 392–403.

[48] L. Spracklen, S. Abraham, Chip multithreading: opportunities and challenges, in: 11th International Symposium on High-Performance Computer Architecture, 2005, pp. 248–252, http://dx.doi.org/10.1109/HPCA.2005.10.

[49] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, in: AFIPS '67 (Spring), Association for Computing Machinery, New York, NY, USA, ISBN: 9781450378956, 1967, pp. 483–485, http://dx.doi.org/10.1145/1465482.1465560.

[50] R. Saxena, M. Jain, S. Bhadri, S. Khemka, Parallelizing GA based heuristic approach for TSP over CUDA and OPENMP, in: 2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI, 2017, pp. 1934–1940, http://dx.doi.org/10.1109/ICACCI.2017.8126128.

[51] N. Tzy-Luen, Y.T. Keat, R. Abdullah, Parallel cuckoo search algorithm on OpenMP for traveling salesman problem, in: 2016 3rd International Conference on Computer and Information Sciences, ICCOINS, 2016, pp. 380–385, http://dx.doi.org/10.1109/ICCOINS.2016.7783245.

[52] I. Skliarova, A.d. Ferrari, FPGA-based implementation of genetic algorithm for the traveling salesman problem and its industrial application, in: International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, 2002.

[53] I. Mavroidis, I. Papaefstathiou, D. Pnevmatikatos, A fast FPGA-based 2-Opt solver for small-scale Euclidean Traveling Salesman Problem, in: 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007), 2007, pp. 13–22, http://dx.doi.org/10.1109/FCCM.2007.40.