Effective Parallelization of the Vehicle Routing Problem

Rajesh Pandian M*
Indian Institute of Technology Madras, India
mrprajesh@cse.iitm.ac.in

Somesh Singh
CNRS and LIP (UMR5668), France
somesh.singh@ens-lyon.fr

Rupesh Nasre†
Indian Institute of Technology Madras, India
rupesh@cse.iitm.ac.in

N.S. Narayanaswamy
Indian Institute of Technology Madras, India
swamy@cse.iitm.ac.in

ABSTRACT

Capacitated Vehicle Routing Problem (CVRP) is an important combinatorial optimization problem, which is also NP-hard. A wide array of heuristics have been proposed in the literature to obtain an approximate solution to CVRP. To improve the execution time, parallel methods have been developed for accelerating metaheuristics-based algorithms, genetic algorithms, and evolutionary algorithms for CVRP. Despite these advances, our experiments with the state-of-the-art parallel solutions indicate that their run times are too high to be practically useful. The combinatorial explosion is so high that the execution time is prohibitively large even on mid-sized CVRP instances having a few hundred customers. In this work, we propose a novel technique which combines *local search* and *randomization* for solving CVRP faster with reasonable accuracy, even on large problem instances. Our usage of randomization enables searching a large space of candidate solutions. We experimentally compare our proposed method with the state-of-the-art GPU implementations on diverse input instances and demonstrate the efficacy of our approach. Our sequential and shared-memory parallel implementations are on an average 36-1189× faster than the state-of-the-art GPU-parallel genetic algorithms while also achieving a superior solution quality. Furthermore, our reported solutions are close to the current best-known solutions from CVRPLIB.

CCS CONCEPTS

• **Computing methodologies** → *Massively parallel algorithms*; **Shared memory algorithms**; • **Theory of computation** → Approximation algorithms analysis; • **Applied computing** → *Operations research.*

KEYWORDS

capacitated vehicle routing problem, minimum spanning tree, travelling salesman problem, operations research

*Corresponding author
†Partially supported by the NSM grant CS19201123MEIT008606.

GECCO '23, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0119-1/23/07…$15.00
https://doi.org/10.1145/3583131.3590458

ACM Reference Format:
Rajesh Pandian M, Somesh Singh, Rupesh Nasre, and N.S. Narayanaswamy. 2023. Effective Parallelization of the Vehicle Routing Problem. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590458

1 INTRODUCTION

The capacitated vehicle routing problem (CVRP) [12] is an important and extensively studied combinatorial optimization problem. Given a fleet of vehicles from a starting depot, CVRP determines the least-cost round-trip delivery routes for the vehicles to a set of geographically scattered customers while respecting the constraints on the maximum freight loadable on each vehicle (formal definition in Section 2.1). It plays a pivotal role in logistics, delivery management, and transportation [17, 22, 29].

CVRP is known to be *strongly* NP-hard [23], having the well-known *traveling salesman problem* (TSP) [14] and the *bin packing problem* [16] as special cases. Despite the rich literature on CVRP, solving the problem for moderate or large instances (having hundreds of customers) often encountered in practice [15, 33] with the desired accuracy continues to take excessively long time. The execution time of these approximate solutions can be improved with parallelization. Recently, parallel genetic algorithms have been developed on GPUs with this motivation. The state-of-the-art GPU implementations are due to Yelmewad and Talawar [35], and Abdelatti and Sodhi [1]. While quite effective on smaller instances, these GPU implementations are time-consuming (taking several tens of minutes) to produce a reasonable solution for large instances. Table 1 shows their running times (in seconds) on three large input instances from CVRPLIB [31].

Instance	Number of customers	Time (s) [35]	Time (s) [1]
Flanders2	30,000	8,355	2,534
Flanders1	20,000	7,768	2,031
Brussels1	15,000	7,164	871

Table 1: State-of-the-art GPU methods are time-consuming.

Therefore, there is a need for faster techniques which are capable of obtaining a usable, nearly-optimal solution in a reasonable time especially on large instances. Furthermore, a quickly computed reasonable solution may serve as a good starting point for metaheuristic-based methods, which aim for a near-optimal solution. With this motivation, we present our tool named `ParMDS` for

computing a good CVRP solution fast. It combines carefully crafted heuristics and multi-core parallelization to achieve this goal.

ParMDS employs a minimum spanning tree (MST) and depth-first traversal (DFS)-based lightweight technique. Theoretically, a MST-based solution provides a 2-approximate solution to TSP [34]. In practice, however, we establish that it is able to obtain a much better solution to CVRP. ParMDS uses a two-pronged approach: it exploits *local search* and *randomization* to improve the solution quality, and uses OpenMP parallelization to reduce execution time. This paper makes the following contributions:

- We devise a MST and DFS-based combinatorial heuristic combining iterative local search and randomization to compute a *good* solution to CVRP. The randomization is over the different traversal orders of a MST, each of which provides a valid candidate solution.
- We present efficient shared-memory parallel and sequential implementations of our proposed technique.
- ParMDS achieves significant speedup (between 36-1189×) over the GPU baseline [1, 35] and better solution quality (geomean deviation of 11.85%) against the current best-known solutions listed in CVRPLIB.

2 BACKGROUND AND RELATED WORK

We describe the problem formally and discuss related work.

2.1 Problem Description

Capacitated Vehicle Routing Problem (CVRP). We are given a depot and $(n-1)$ customers with coordinates (x_i, y_i) for all $i = 0, \ldots, (n-1)$. Note that the depot is at node 0. There is a fleet of *identical* vehicles, each having an integral capacity $Q > 0$, which are initially located at the depot. We assume the fleet to comprise as many vehicles as required to serve the customers' demands. A customer i has demand d_i such that $0 < d_i \leq Q$; $i = 1, \ldots, (n-1)$. The depot has zero demand. The travel cost between any pair of customers, or between customers and the depot, is taken as the Euclidean distance between them. The Euclidean distance between any two points j and k in a 2-D plane is computed as $\sqrt{(x_j - x_k)^2 + (y_j - y_k)^2}$, where the coordinates of points j and k are (x_j, y_j) and (x_k, y_k), respectively. The goal of CVRP is to find a set of routes having the minimum total distance to serve all the customers such that each route starts and ends at the depot, each customer is served exactly once, and total customer demands on any route does not exceed the vehicle capacity Q.

CVRP instance as a graph. A CVRP instance can be modelled as a graph [21]. Let, in an input instance, there be a depot and $(n-1)$ customers denoted by nodes having ids $\{0, \cdots, (n-1)\}$. The depot is represented by a node with id 0, whose demand equals 0. We define a complete graph $G = (V, E)$ such that $V = \{0, \ldots, (n-1)\}$ is the vertex set. V represents the depot and the customers. The edge-weight associated with an edge $(i, j) \in E$ is the Euclidean distance between vertices i and j; the edge-weight is symmetric.

Measure of solution quality. Since CVRP is NP-hard, we often need to settle for an approximate solution. For several problem instances in the dataset available on CVRPLIB the optimal solution is known, while for few, it is not yet known. Hence, we need to work with the best-known-solution (BKS) for an instance. As a convention, the quality of a solution is measured as: $\texttt{Gap} = \frac{Z_S - Z_{\text{BKS}}}{Z_{\text{BKS}}} \times 100$, where Z_S is the cost of the solution reported by a solver S and Z_{BKS} is the BKS. Note that Gap is expressed as a percentage. The smaller the Gap, the better is the solution. We use Gap in the quantitative evaluation of our method (Section 4).

2.2 Related Work

Since the introduction of the truck dispatching problem [12], several variants to this problem have been studied in literature under the name of Vehicle Routing Problem (VRP) — CVRP, CVRP with time windows, VRP with split deliveries, mixed fleet VRP, vehicle routing with pickups and deliveries, and many more.

The approaches for solving CVRP can be broadly categorized into exact and approximate. Exact methods find the optimal solution to CVRP. These include methods such as branch and bound strategies, and formulating CVRP as a Linear Programming problem. On the other hand, approximate methods find a sub-optimal solution to CVRP. These include set partitioning heuristics, cluster-first and route-second heuristics, genetic algorithms, evolutionary algorithms, learning-based heuristics, local-search and population-search heuristics. The last three fall in the realm of meta-heuristics.

A classical heuristic for CVRP is the savings algorithm due to Clarke-Wright [9]. It starts with as many routes as there are customers with each route containing exactly one customer. The scheme progressively merges the routes that result in the maximum savings (in terms of cost) while respecting the capacity constraints. The algorithm stops when no more merging of routes is possible. Fast iterative localized optimization [2] is a local-search based iterative improvement algorithm which uses simulated annealing. It employs a caching strategy to work on the solution area that recently produced a better solution and has a generic way of identifying promising neighboring solutions. Slack induction by string removals (SISR) [6] uses string removal subroutines, followed by greedy node insertion and finally fleet minimization. Knowledge-guided local search (KGLS) [4] uses data-mining to separate the good solutions from the others. It identifies different characteristics or metrics of the solution to make this distinction, e.g., the width and the span in radians of routes, the number of intersecting edges and the distances of connecting edges to the depot. Armed with these metrics KGLS develops a heuristic which guides the local-search process. LKH-3 [20] is an open-source implementation of Lin-Kernighan-Helsgaun [19]. LKH3 converts the CVRP problem into a symmetric TSP and handles the constraints, especially vehicle capacity, with a penalty function. The hybrid iterated local search [28] combines iterative local search heuristic and set partitioning formulation. It interleaves a metaheuristic approach with mixed integer programming to be able to solve multiple variants of VRP. HGS-CVRP [33] is a hybrid genetic search algorithm, and the current state-of-the-art sequential method, for solving VRP problems. It carefully applies a combination of well-known local search heuristics, and also proposes an efficient inter-route refinement heuristic called SWAP* to achieve better solution quality than all the alternatives. HGS-CVRP is not multi-threaded.

More recently, parallelization of CVRP on GPU has been explored [1, 25, 35]. Abdelatti and Sodhi [1] propose a hybrid algorithm combining the genetic algorithm and 2-Opt local search which runs entirely on GPU. Yelmewad and Talawar [35] propose a GPU-parallel metaheuristic algorithm which relies on local search heuristics. It employs two parallelization strategies—customer-level parallelization and route-level parallelization. It can handle large problem instances having upto several thousand customers. We use the foregoing two GPU-parallel state-of-the-art methods as baseline in the empirical evaluation of ParMDS. All the prior techniques take excessively long to arrive at a satisfactory solution to CVRP especially for *large* input instances. On the contrary, our ParMDS technique is designed to solve CVRP at scale – it can handle large input instances efficiently, computing a good solution in a practically reasonable time.

3 OUR METHOD

We propose a novel method, ParMDS, for computing an approximate yet reasonable solution to CVRP quickly, using parallelization. We also present an optimized sequential version of ParMDS, which we call SeqMDS. We begin with an overview of our proposal followed by a detailed discussion of our technique.

Algorithm 1: ParMDS: The proposed method

Input: $G = (V, E)$, Demands $D := \bigcup_{i=1}^{n} d_i$, Capacity Q
Output: R, a collection of routes as a valid CVRP solution
C_R, the cost of R

1 $T \leftarrow$ Prims_MST (G) /* Step 1 */
2 $C_R \leftarrow \infty$
3 **for** $i \leftarrow 1$ *to* ρ **do** /* Superloop */ /* Parallel */
4 $T_i \leftarrow$ Randomize (T) /* Shuffle Adjacency List */
5 $\pi_i \leftarrow$ DFS_Visit $(T_i$, Depot$)$ /* Step 2 */
6 $R_i \leftarrow$ Convert_To_Routes $(\pi_i$, Q, D$)$ /* Step 3 */
7 $C_{R_i} \leftarrow$ Calculate_Cost (R_i) /* Parallel */
8 **if** $C_{R_i} < C_R$ **then**
9 $C_R \leftarrow C_{R_i}$ /* Current Min Cost */
10 $R' \leftarrow R_i$ /* Current Min Cost Route */
11 **end**
12 **end**
13 $R \leftarrow$ Refine_Routes (R') /* Step 4 */
14 **return** R, C_R

3.1 ParMDS Overview

Algorithm 1 presents the ParMDS algorithm. It takes as input the graph representation, G, of a CVRP instance (see Section 2.1), the customer demands, D, and the vehicle capacity, Q, and outputs a valid CVRP solution. ParMDS proceeds in four super-steps. First, it constructs a minimum spanning tree (MST), T, of this edge-weighted complete graph (Line 1); this is also the first step in the well-known 2-approximation algorithm for TSP [34, pp. 45-46]. In the second super-step, ParMDS performs depth-first traversal on the MST, starting from the depot, to define an ordering on the nodes (Line 5). Importantly, this step is preceded by a random shuffle of the adjacency lists of the MST (Line 4) which enables generating a distinct DFS visit order of the MST vertices in each iteration of the superloop (Line 3). The third super-step partitions the ordered list of nodes (π_i) into separate routes to comply with the capacity, Q (Line 6). Steps two and three are repeated a fixed number, ρ, of times to explore the space of different DFS traversals in order to search for better routes. ρ is a tunable parameter and can be set to suit the desired accuracy and performance. Finally, the fourth super-step refines the ordering of the nodes within each route of the solution using the well-known *2-Opt* [11, 14] and the *nearest-neighbor* [27] heuristics for TSP (Line 13). At the end, we obtain a collection of routes which is a valid CVRP solution, along with its cost.

3.2 An Illustrative Example

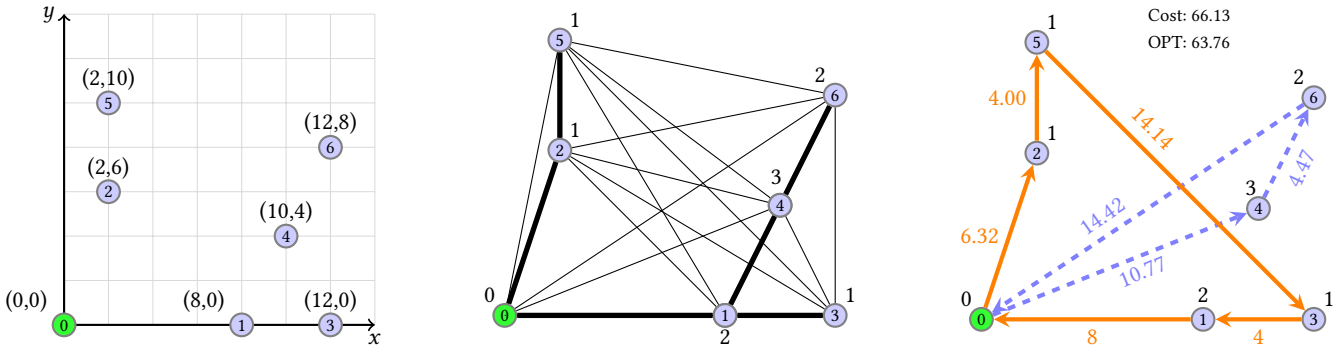
We demonstrate our method through an illustrative example. Consider a CVRP instance containing six customers and a depot, shown in Figure 1(a). Let the vehicle capacity be 5. The depot is represented as 0. The demand at the depot is zero. Figure 1b shows the complete graph for the input instance, having seven nodes, along with the demand at each of the nodes mentioned beside the respective node. The edges of the MST obtained after Step 1 of Algorithm 1 are shown with **thick** edges. Figure 1c shows the CVRP solution produced by ParMDS, after the completion of Algorithm 1. Note that the solution contains two routes, shown using orange solid-lines and blue dashed-lines. Each of the two routes starts from and ends at the depot. The cost of ParMDS's solution is 66.13; it is the total distance to be travelled in both the routes. The optimal solution for this example contains two routes with a total cost of 63.76.

We next focus on Step 2 of Algorithm 1. This step, in conjunction with random shuffle of the nodes' neighbors, enables ParMDS to generate several candidate solutions, which is critical to exploring the large solution space of CVRP. Figure 2 shows four candidate solutions generated by ParMDS, along with their respective costs. Figure 2(a) shows the MST from Figure 1b, rooted at the depot; it is redrawn for clarity. Note that in each of the Figures 2(a)–2(d), the MST is the same, with the neighbors ordered differently because of the random shuffle of the adjacency lists of the nodes.

To understand this better, consider Figures 2(a) and 2(b). The only difference in them is at node 1. For node 1, the neighbor order in Figure 2(a) is <3, 4>, whereas that in Figure 2(b) is <4, 3>. As DFS traversal is sensitive to the neighbor order, we get a different permutation π, and therefore a different route and cost (as shown in the figure below the trees). The difference between Figures 2(b) and 2(c) is at node 0 (<1, 2> versus <2, 1>). Similarly, Figures 2(c) and 2(d) only differ in node 1's neighbor order.

A distinct permutation π is produced for each of the isomorphic MSTs shown in Figure 2(a)–2(d), each having a different cost. Let the i^{th} route be represented as r_i. Convert_To_Routes (Step 3) takes the permutation π as input and outputs a set of routes. Each permutation produces up to 3 routes, which are marked at the boundary (routes at the bottom). Each of the routes r_i produced is prefixed and suffixed with the depot. However, for brevity, we show only the boundary of each of the routes.

The permutation and the set of routes in Figure 2(c) are visualized in Figure 3(a). Figure 3(a) shows the two routes <2, 5, 1, 3> and <4, 6> using orange solid-lines and blue dashed-lines respectively.



(a) Input instance I **(b) Graph for I, along with node-demands** **(c) Final routes generated by ParMDS**

Figure 1: ParMDS in action on an example input instance with $n = 7$ and $Q = 5$.

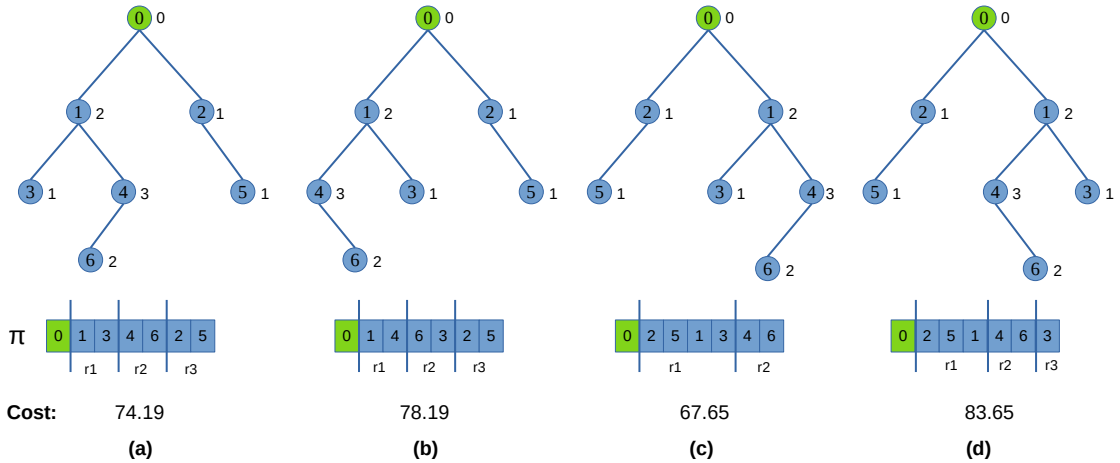


Figure 2: Candidate CVRP solutions for the example input instance, I, from Figure 1.

The original route cost is 67.65 computed using Calculate_Cost. Notice in Figure 3(a) the intersecting edges in the first route r_1: node 5 → node 1 and node 3 → node 0. The modified route-set after the removal of intersection is {<2, 5, 3, 1>, <4, 6>}. The intersection in r_1 is removed using the 2-Opt_Heuristic as shown in Figure 3(b), improving the cost from 67.65 to 66.13 (which is only 3.72% away from the optimal cost of 63.76 for this example).

3.3 Algorithmic Details

We describe the four steps of our Algorithm 1 in detail now.

1) ***Constructing MST of G:*** On an undirected graph G, which is an edge-weighted complete graph on n nodes, we run Prim's MST algorithm [24]. Prim's algorithm starts from a vertex of the graph and iteratively grows the tree by selecting the least-weight edge between the vertices of the tree and a non-selected vertex of the graph. On G, Prim's algorithm (Algorithm 1, Line 1) begins from the depot, and the MST, T, contains all the nodes (since the graph is connected). In the illustrative example, Figure 1b shows the MST with **thick** edges. It is important to note that we compute the MST only once.

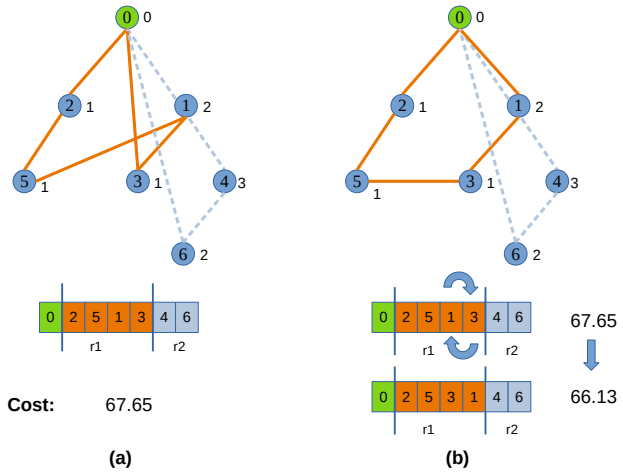


Figure 3: Intra-route optimization for Figure 2(c)

2) ***Depth-first traversal on the MST:*** On the MST T obtained from Step 1, we perform a depth-first traversal starting from the depot

Algorithm 2: CONVERT_TO_ROUTES (π, Q, D)

Input: A permutation π, Capacity Q, Demands D
Output: Routes, a set of routes

```
1  OneRoute ← φ; Routes ← φ  /* Initialize to empty */
2  ResidueCap ← Q            /* Residual capacity */
3  for v ∈ π do
4      if v = Depot then continue        /* Skip Depot */
5      if ResidueCap - D[v] ≥ 0 then     /* Same Route */
6          OneRoute.add(v)
7          ResidueCap ← ResidueCap - D[v]
8      else                              /* New Route */
           /* Add previous route to Routes set      */
9          Routes ← Routes ∪ OneRoute
10         OneRoute ← φ
11         OneRoute.add(v)
12         ResidueCap ← Q - D[v]
13     end
14 end
   /* Add the last route to Routes set           */
15 Routes = Routes ∪ OneRoute
16 return Routes
```

Algorithm 3: REFINE_ROUTES (Routes)

Input: Routes
Output: ModifiedRoutes, set of routes after refinement

```
   /* Intra-route improvement techniques          */
1  RoutesOne ← NEAREST_NEIGHBOUR_HEURISTIC (Routes)
2  RoutesTwo ← 2-OPT_HEURISTIC(RoutesOne)
3  RoutesThree ← 2-OPT_HEURISTIC(Routes)
4  ModifiedRoutes ← φ
5  for i ← 1 to Routes.size() do // i^th route /* Parallel */
       /* Choose the minimum cost route          */
6      Min-ith-Route ← MIN{RoutesTwo[i], RoutesThree[i]}
7      ModifiedRoutes ← ModifiedRoutes ∪ Min-ith-Route
8  end
9  return ModifiedRoutes
```

(Algorithm 1, Line 5). This DFS traversal leads to a permutation of the n nodes (similar to the one in the double-tour TSP processing). Figure 2(a) shows the permutation π obtained after this step, below the tree.

Random Shuffle of Adjacency Lists. The node visit-order in a DFS traversal is contingent on the order of nodes stored in the adjacency lists (of the MST T). ParMDS takes advantage of this property to generate several candidate solutions for CVRP by shuffling the adjacency lists of the MST (Algorithm 1, Line 4) before performing the depth first traversal in every superloop iteration. The random shuffle of adjacency lists is performed using C++'s `std::algorithm/shuffle()` method.

3) ***Partitioning the nodes into routes:*** The permutation of all the nodes (or customers) is represented as π. Algorithm 2 effectively slices the permutation π and makes each piece a route respecting the vehicle's capacity constraint. `OneRoute` is used to store nodes of a single route, and `Routes` stores the set of routes. Both are initially empty. CONVERT_TO_ROUTES picks a node starting from the first node of the permutation (Line 3) and keeps adding to a route until no more nodes can be added (Line 6). If adding a node violates the capacity constraint, that is, the sum of all the nodes' demands in the route exceeds the capacity, it creates a new route starting at that vertex (Line 11). It repeats the steps for all the remaining nodes in the permutation. Figures 2(a)–2(d) show the conversion of each permutation, π, into routes. The routes are numbered r_i, and the costs of the routes are mentioned at the bottom.

We repeat steps 2 and 3 (Algorithm 1 Lines 5 and 6) a fixed number ρ of times such that the random adjacency lists capture several of the possible orderings. We calculate the cost of the routes (Line 7) at the end of step 3 and update the min-cost and min-routes (Lines 8-10) accordingly. At the end of each iteration, the cost C_{R_i} is computed using CALCULATE_COST (Algorithm 4) and the current minimum cost C_R and corresponding routes R' are updated accordingly. After a fixed number of iterations, the current best solution R' is fed to the refinement stage (Algorithm 3) for final processing.

4) ***Refining the ordering within routes:*** There are multiple ways to refine routes, but they can be categorized into two: i) intra-route improvement: updating node order within a route, and ii) inter-route improvement: improving the solution by moving nodes across different routes. A crucial advantage of intra-route improvement is that multiple intra-techniques can be composed (that is, run one after another) and adding new intra-route techniques in future is easy. The REFINE_ROUTES step (Algorithm 3) of ParMDS uses two intra-routes improvement techniques, namely NEAREST_NEIGHBOUR_HEURISTIC (Line 1) and 2-OPT_HEURISTIC (Lines 2-3). Both these heuristics are very well studied in the context of TSP. 2-OPT_HEURISTIC removes intersections within a route, by considering the nodes in a route to form a TSP problem instance, as shown in Figures 3(a) and 3(b). NEAREST_NEIGHBOUR_HEURISTIC works in a different way. Initially, all the nodes in a route are unvisited. After visiting the first node in a route, we subsequently pick the next nearest node out of the remaining unvisited nodes of the route. This potentially results in a different node ordering within a route but it may create an intersection of the edges of the route. So, we run 2-OPT_HEURISTIC twice (Lines 2-3) as the last step to remove the intersections, if any.

After running both (in different combinations), we go over all the routes (Line 5) and pick the lower cost alternative for every route. The result of the REFINE_ROUTES step is ParMDS's final solution.

Reducing the Search Space. In CVRP, the solution space is exponential in the size of the input. Thus, to solve a problem satisfactorily in a reasonable time, it is imperative to prune the solution search space effectively and quickly — avoiding unfruitful searches to the extent possible. ParMDS accomplishes search space reduction by performing a DFS traversal of the MST. This simple technique helps in avoiding unfruitful permutation of nodes, thus substantially reducing the search space to explore. Furthermore, generating

Algorithm 4: CALCULATE_COST (Routes)

Input: Routes, a collection of valid CVRP routes
Output: Cost, the cost of Routes

```
1 Cost ← 0
2 for Route ∈ Routes do                          /* Parallel */
      /* Route is a sequence of one or more nodes */
      /* Distance(i,j): 2D Euclidean distance
         between nodes i and j                    */
3     Cost ← Cost + Distance(depot, first node in Route)
4     for adjacent nodes (p, q) ∈ Route do /* Parallel */
5         Cost ← Cost + Distance(p,q)
6     end
7     Cost ← Cost + Distance(depot, last node in Route)
8 end
9 return Cost
```

a random permutation of the adjacency lists followed by a DFS traversal, for a fixed number ρ of iterations of the superloop enables ParMDS to explore the reduced, yet massive solution space faster. Thus, ParMDS arrives at a good solution to CVRP in a reduced time. We quantitatively discuss the effect of ρ on the performance and solution quality of ParMDS in Section 4.2.

Time complexity. The running time of SeqMDS is the sum of the time taken by the MST computation and time spent on RANDOMIZE, DFS_VISIT and CONVERT_TO_ROUTES for the fixed number of iterations and, finally, the REFINE_ROUTES step. As we use the adjacency list representation, the time complexity of MST is $O(m \log n)$ where n and m represent the numbers of nodes and edges in G, respectively. As the initial graph is complete, $m = O(n^2)$. So, the MST computation takes $O(n^2 \log n)$ time. The MST contains only n nodes and $n-1$ edges. DFS is linear in the size of the input: $O(n+n-1) = O(n)$. CONVERT_TO_ROUTES is again linear time as it runs over all the nodes in a permutation once. Similarly, randomization runs linearly on the number of edges of MST. RANDOMIZE, DFS_VISIT and CONVERT_TO_ROUTES run a fixed number of times ρ; to be exact, $\rho \times O(3n) = \rho \times O(n)$. Refinement step includes NEAREST_NEIGHBOUR_HEURISTIC and 2-OPT_HEURISTIC, each of which takes quadratic time: $O(3n^2)$ to be precise (Algorithm 3 Lines 1-3). At the end of refinement, the processing iterates through all the routes to pick the minimum valued i^{th} route, which is $O(n)$. So, the refinement step is $O(n^2 + n)$. Putting everything together, we get a total running time $O(n^2 \log n + 3n^2 + (\rho \times 3n) + n)$. Omitting the constants finally results in $O(n^2 \log n + n^2 + n)$.

3.4 ParMDS: Parallel and Random

Of the total time taken by SeqMDS Figure 4 describes the percentage of time spent at each step of our method. As we observe from Figure 4, over 99% of the time is spent in the superloop (Algorithm 1, Line 3) in general (except Belgium-type which consumes 80-98%); the other two steps 1 and 4 take very little time. Each iteration of the superloop takes a copy of the original adjacency list information, performs Steps 2 and 3, and updates C_R and R'. There is no dependency across iterations, which makes the loop embarrassingly parallel. Further, we also parallelize CALCULATE_COST (Algorithm 4)

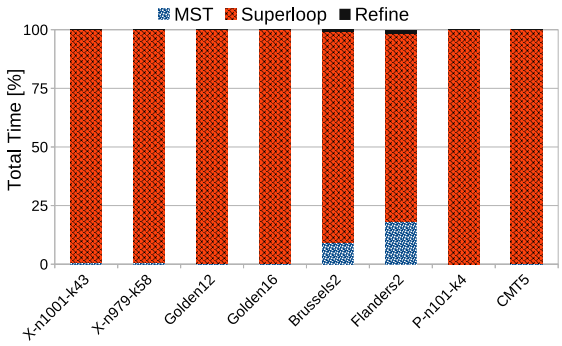


Figure 4: SeqMDS—time spent at each step.

Algorithm 5: ParMDS: Finer details of Algorithm 1

```
   /* Standard: stride = 1;                          */
   /* Strided : stride = #CPU cores                  */
   /* Parallel for loop: Standard/Strided            */
1  for i ← 1; i ≤ ρ; i = i + stride do
2      for v ∈ V do
           /* seed ← constant or i or rand()         */
3          SHUFFLE-NEIGHBORS(AdjList(v), seed);
4      end
5      ...
6  end
7  ...
```

and REFINE_ROUTES (Algorithm 3). Importantly, randomization removes any bias due to pre-exisiting node order in the adjacency lists. Algorithm 1 highlights the functions with parallel tasks. The parallel tasks are marked with Parallel.

Standard vs. Strided Parallelization. Algorithm 5 zooms in on the parallel superloop in Algorithm 1 (Line 3). The loop can have a stride of 1 (which we call Standard) or more than 1 (which we call Strided). We use the Strided approach — setting the stride to the number of CPU cores; stride-many iterations are run in parallel. We observe the Strided approach to have better performance (due to reduced *false sharing* in the shared C_R and R').

Seed Variation. The seed for random shuffle of adjacency lists can be set in different ways: **1)** Constant, **2)** Variable, or **3)** Random. The random shuffle of the adjacency list is effected using a for-loop which runs over all vertices. For each iteration of the superloop, the shuffle is invoked |V| times. With Constant, the same seed value is used throughout. With Variable, the iteration number is set as the seed, whereas in Random, the seed is also generated at random. So, each invocation of SHUFFLE-NEIGHBORS gets a different seed. Using different seeds helps generate different node orders within the adjacency lists, thus diversifying the solution.

All the above three types can be combined with the Standard and the Strided parallelization approaches. We quantitatively compare the three methods in Section 4 and use the best-performing combination in ParMDS.

4 EXPERIMENTS

We quantitatively evaluate the performance and solution quality of our proposed method and compare it with the state-of-the-art.

4.1 Setup

All the experiments are carried out on a machine having Intel Xeon CPU E5-2640 v4 with 40 cores, clock-speed of 2.4 GHz, 25 MB L3-cache, and 64 GB memory. It runs CentOS Linux 7 (64-bit). GPU implementations are run on NVIDIA's Tesla P100 GPU and 12GB global memory. P100 has 3584 cores, each with a clock-speed of 1.33 GHz, spread across 54 streaming multiprocessors. We use CUDA 11.5 to compile and execute the methods on the GPU. SeqMDS is compiled with gcc 9.3.1 with flag `-std=c++14`. ParMDS is complied using `nvc++` compiler from NVIDIA's HPC SDK 22.11 with flag `-acc=multicore`; it uses OpenMP for parallelization. All codes are available at *https://github.com/mrprajesh/parMDS*.

Input Instances. We use 130 diverse CVRP instances from CVRPLIB [31]. These instances are due to several contributors [3, 5, 7, 8, 13, 18, 26, 32] and are also part of the 12^{th} DIMACS Programming Challenge on the vehicle routing problem [30]. We categorize the 130 instances into four types namely: X, Golden, Belgium (which comprises Antwerp*, Brussels*, Flanders*, Ghent*, Leuven*), and Others (which comprises CMT5, E-n*, F-n135-k7, P-n101-k4, tai385). X, Golden, Belgium and Others contain 100, 12, 10, and 8 instances respectively. The size n of inputs ranges from 76 to 30,001 nodes, and the vehicle capacity Q is in the range of 3–2210. CVRPLIB maintains the best-known solution (BKS) for all the CVRP instances (*http://vrp.galgos.inf.puc-rio.br/index.php* accessed on July 19, 2022) which we utilize to compute the Gap.

Comparisons. We compare our tools, ParMDS and SeqMDS, against two recent state-of-the-art GPU-parallel implementations of genetic algorithms for CVRP: **Base1** which implements a metaheuristic algorithm for VRP on GPU in CUDA [35] and **Base2** which implements a genetic algorithm for VRP on GPU using Numba, CuPy and NumPy [1]. Four significant metrics for measuring the efficacy of a VRP heuristic are speed, accuracy, simplicity and flexibility [10]. The focus of ParMDS is on speed, simplicity and accuracy (or the solution quality). The accuracy is measured using Gap (Section 2.1). To increase statistical significance, we report the execution time and the Gap of ParMDS averaged over 5 independent runs. We present results for ParMDS with 40 threads.

4.2 Determining the Superloop Iteration Bound

As discussed in Section 3.3, the performance and the solution quality of ParMDS depend on the number of iterations, ρ, of the superloop (Line 3, Algorithm 1). We study the impact of increasing ρ on the solution quality (measured using Gap) and the execution time of ParMDS. Figure 5 shows the effect of increasing ρ on four representative input instances from our test-suite; these include the largest input of each instance-type. As we observe from Figure 5, Gap decreases monotonically with increase in ρ, for all four input instances. We further observe that the execution time increases superlinearly and monotonically with increase in ρ, for all four input instances. In order to manage this trade-off between performance and solution quality well, based on the empirical evidence presented in Figure 5, we identify $\rho = 10^5$ as a sweet-spot for ParMDS.

Instance	BKS	Q	n	Execution Time (s) Base1	Base2	SeqMDS	ParMDS
X-n1001-k43	72,355	131	1,001	0.98	4,741.34	7.43	0.20
X-n979-k58	118,976	998	979	2.34	4,881.31	7.47	0.22
Golden12	1,101	1,000	484	0.38	1,513.46	3.21	0.09
Golden16	1,611	1,000	481	0.27	1,280.89	3.51	0.08
Brussels2	345,481	150	16,001	326.53	7,033.72	173.42	20.42
Flanders2	4,373,320	200	30,001	2,534.29	8,354.91	373.58	79.37
P-n101-k4	681	400	101	-	286.68	0.68	0.03
CMT5	1,291	200	200	-	214.80	1.44	0.04

Table 2: Execution time of the baselines, SeqMDS and ParMDS

Method	Gap Constant	Variable	Random	Execution Time (s) using Random
SeqMDS	22.15	21.72	17.56	1,722.44
ParMDS-Standard	14.14	13.49	10.96	1,522.26
ParMDS-Strided	14.14	13.48	11.85	186.50

Table 3: Gap for Standard vs. Strided, and execution time

4.3 Comparison of Execution Time

We evaluate the two baselines and our method on the 130 input instances. Compared to Base1 and Base2, ParMDS runs faster; Base2 takes substantially longer time than ours. ParMDS takes 187 seconds and SeqMDS takes 28 minutes (Table 3) to complete on all the instances, whereas Base1 takes 107 minutes and Base2 2.5 days. Base2 performs poorly, especially on large instances.

Tables 2 and 4 show the absolute run times and costs for the baselines and our tool on two largest instances per type. Base1 did not run on Others-type instances (denoted by '-' in Tables 2 and 4). Over all 130 instances, the average speedup of ParMDS is 9×, 36×, 1189× over SeqMDS, Base1 and Base2 respectively. Figure 6 shows the average speedup of ParMDS over the two baselines. We observe a similar trend for all instance-types (see Figure 6).

Effect of Randomization. Table 3 shows the benefits of the Strided approach over the Standard-parallelization approach (discussed in Section 3.4). For each type of seeding (Constant, Variable and Random), we list the average Gap corresponding to it and the overall time for all 130 instances. Parallelization helps reduce the Gap over sequential execution. Although there is a slight increase in Gap when using a random seed, the execution time rules in our favour. We use Strided with Random seed in all our experiments for ParMDS.

4.4 Comparison of Solution Quality

We now evaluate the quality of the solution produced by our proposed method. The smaller the Gap, the better is the solution quality. We use Gap as defined in Section 2.1. Table 4 tabulates the absolute cost of baselines and ParMDS along with the BKS. We present the absolute numbers for the two largest input instances per type. We observe that ParMDS's solution indeed has a smaller cost compared to the baselines.

Figure 7(a) shows the average Gap across all 130 instances and per type. The Gap is plotted along the y-axis. Each bar in a set represents the Gap of the baselines and our tool. Base1 has a smaller Gap compared to Base2. The Gap of Base1 and SeqMDS has only a marginal difference. ParMDS's Gap is smaller than both the baselines. The average Gap of Base1, Base2, SeqMDS and ParMDS are 17.56%, 175.82%, 17.88% and 11.85% respectively. Our tool consistently has

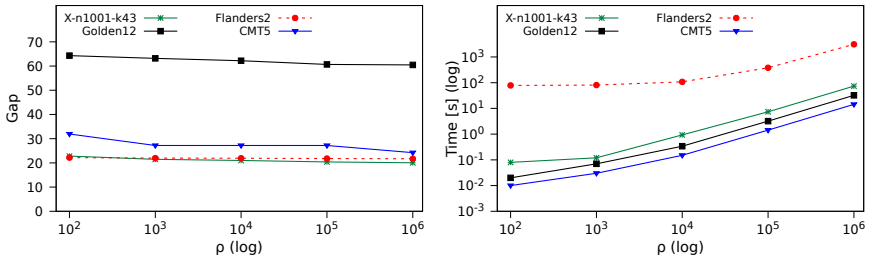


Figure 5: Effect of increasing ρ on (a) Gap and (b) run time of ParMDS

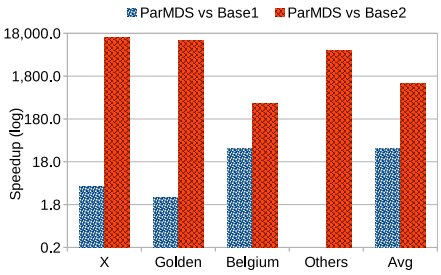


Figure 6: Speedup of ParMDS vs. baselines

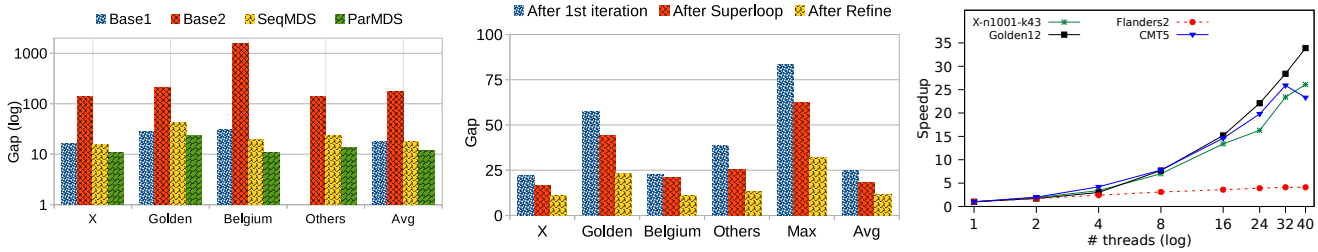


Figure 7: (a) Gap comparison [lower is better] (b) Gap at the end of: 1st iteration, Superloop and Refine step (c) ParMDS Scalability

Instance	BKS	Base1	Base2	ParMDS
X-n1001-k43	72,355	91,042	415,387	80,217
X-n979-k58	118,976	137,072	455,823	126,529
Golden12	1,100.67	1,504	4,296	1,461.14
Golden16	1,611.28	2,128	6,069	2,098.10
Brussels2	345,481	457,430	9,406,251	394,292
Flanders2	4,373,320	6,164,809	202,483,288	4,882,650
P-n101-k4	681	-	3,864	755
CMT5	1,291.29	-	1,793	1,486.16

Table 4: Absolute cost of baselines and ParMDS

a smaller Gap than Base2 on all the input instances, and a smaller Gap than Base1 on most of the instances (102 out of 122). The improvement in the Gap of ParMDS over SeqMDS is due to the effect of randomization and parallelization (in Section 3.4).

To quantify the gap per step, in Figure 7(b) we plot the Gap at the end of: the 1st iteration of the superloop, all the iterations of the superloop, and the Refine_Routes step. We observe that overall the local-search iteration and Refine_Routes steps help improve the solution quality. Note that maximum Gap of any instance for ParMDS is 81% in the worst case, while it is much higher for the baselines. ParMDS's solution is well within the 2-approximate solution.

4.5 Scalability Study of ParMDS

We plot Figure 7(c) to study the scalability of ParMDS. We vary the thread size {2,4,8,16,24,32,40} in the Strided and Random configuration to record the speedup of ParMDS over single-threaded ParMDS on the largest instance of each instance-type. We observe that for X-n1001-k43, Golden12 and CMT5 the speedup grows steadily. However, for Flanders2 of Belgium-type, we only observe marginal speedup; the speedup tends to flatten after 8 threads. This is because the solution space for Flanders2 is much larger compared to other instances. This is due to the customers being densely packed (the number of customers per unit area is high) and the high capacity-to-customer demand ratio.

5 CONCLUSION

We proposed an efficient shared-memory parallel method, called ParMDS, for obtaining a satisfactory approximate solution to the capacitated vehicle routing problem faster. ParMDS employs a novel MST- and DFS-based method in conjunction with randomization which enables it to explore a large space of candidate solutions quickly, even for large input instances. We compared ParMDS with the state-of-the-art methods using GPU-parallel genetic algorithms. Experiments on diverse input instances from CVRPLIB showed that ParMDS is on average 36-1189× faster than the baselines and also achieves a superior solution quality over the baselines; average Gap of ParMDS is 11.85% compared to the best-known solution.

We have three directions for future work. We plan to develop a GPU-parallel version of the proposed method to further enhance performance. On the algorithmic front, we plan to build *direction-awareness* into the current scheme, and add inter-route refinement strategies to better the solution quality of ParMDS. Furthermore, we observed empirically that in ParMDS randomization is instrumental in improving the solution quality. It would be interesting to study the effect of randomization on the solution quality theoretically.

ACKNOWLEDGMENTS

We thank Eduardo Uchoa and Eduardo Queiroga for the discussions and their insightful inputs which improved our understanding of the current CVRP landscape. We are grateful to Pramod Yelmewad and Marwan Abdelatti for their help in running their code.

REFERENCES

[1] Marwan F. Abdelatti and Manbir Singh Sodhi. 2020. An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem. In *GECCO '20: Genetic and Evolutionary Computation Conference, Cancún Mexico, July 8-12, 2020*, Carlos Artemio Coello Coello (Ed.). ACM, 663–671. https://doi.org/10.1145/3377930.3390159

[2] Luca Accorsi and Daniele Vigo. 2021. A Fast and Scalable Heuristic for the Solution of Large-Scale Capacitated Vehicle Routing Problems. *Transportation Science* 55, 4 (2021), 832–856. https://doi.org/10.1287/trsc.2021.1059

[3] Florian Arnold, Michel Gendreau, and Kenneth Sörensen. 2019. Efficiently solving very large-scale routing problems. *Computers and Operations Research* 107 (2019), 32–42. https://doi.org/10.1016/j.cor.2019.03.006

[4] Florian Arnold and Kenneth Sörensen. 2019. What makes a VRP solution good? The generation of problem-specific knowledge for heuristics. *Computers & Operations Research* 106 (2019), 280–288. https://doi.org/10.1016/j.cor.2018.02.007

[5] Philippe Augerat, José Manuel Belenguer, Enrique Benavent, Angel Corberán, D. Naddef, and Giovanni Rinaldi. 1995. Computational results with a branch and cut code for the capacitated vehicle routing problem. *Technical Report 949-M* (01 1995).

[6] Jan Christiaens and Greet Vanden Berghe. 2020. Slack Induction by String Removals for Vehicle Routing Problems. *Transportation Science* 54, 2 (2020), 417–433. https://doi.org/10.1287/trsc.2019.0914

[7] N. Christofides and S. Eilon. 1969. An Algorithm for the Vehicle-dispatching Problem. *Journal of the Operational Research Society* 20, 3 (1969), 309–318. https://doi.org/10.1057/jors.1969.75 arXiv:https://doi.org/10.1057/jors.1969.75

[8] N. Christofides, A. Mingozzi, and P. Toth. 1979. The vehicle routing problem. *Combinatorial Optimization* 1 (1979), 315–338.

[9] G. Clarke and J. W. Wright. 1964. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research* 12, 4 (1964), 568–581. http://www.jstor.org/stable/167703

[10] Jean-François Cordeau, Michel Gendreau, Gilbert Laporte, Jean-Yves Potvin, and Frédéric Semet. 2002. A guide to vehicle routing heuristics. *J. Oper. Res. Soc.* 53, 5 (2002), 512–522. https://doi.org/10.1057/palgrave.jors.2601319

[11] Georges A Croes. 1958. A method for solving traveling-salesman problems. *Operations research* 6, 6 (1958), 791–812.

[12] G. B. Dantzig and J. H. Ramser. 1959. The Truck Dispatching Problem. *Management Science* 6(1) (1959), 80–91. https://doi.org/10.1287/mnsc.6.1.80

[13] Marshall L. Fisher. 1994. Optimal Solution of Vehicle Routing Problems Using Minimum K-Trees. *Operations Research* 42, 4 (1994), 626–642. https://doi.org/10.1287/opre.42.4.626

[14] Merrill M. Flood. 1956. The Traveling-Salesman Problem. *Operations Research* 4, 1 (1956), 61–75. http://www.jstor.org/stable/167517

[15] MIT Center for Transportation Logistics. 2021. Amazon Last Mile Routing Research Challenge. https://routingchallenge.mit.edu/about-the-challenge/. [Online; accessed 01-Feb-2023].

[16] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[17] Bruce L Golden, Subramanian Raghavan, Edward A Wasil, et al. 2008. *The vehicle routing problem: latest advances and new challenges.* Vol. 43. Springer.

[18] Bruce L. Golden, Edward A. Wasil, James P. Kelly, and I-Ming Chao. 1998. The Impact of Metaheuristics on Solving the Vehicle Routing Problem: Algorithms, Problem Sets, and Computational Results. *Fleet Management and Logistics* (1998), 33–56. https://doi.org/10.1007/978-1-4615-5755-5_2

[19] Keld Helsgaun. 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *Eur. J. Oper. Res.* 126, 1 (2000), 106–130. https://doi.org/10.1016/S0377-2217(99)00284-2

[20] Keld Helsgaun. 2017. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University* (2017), 24–50. http://akira.ruc.dk/~keld/research/LKH-3/LKH-3_REPORT.pdf

[21] Gilbert Laporte. 1992. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* 59, 3 (1992), 345–358. https://doi.org/10.1016/0377-2217(92)90192-C

[22] Gilbert Laporte. 2009. Fifty Years of Vehicle Routing. *Transportation Science* 43, 4 (2009), 408–416. https://doi.org/10.1287/trsc.1090.0301

[23] Adam N. Letchford and Juan José Salazar González. 2019. The Capacitated Vehicle Routing Problem: Stronger bounds in pseudo-polynomial time. *European Journal of Operational Research* 272, 1 (2019), 24–31. https://doi.org/10.1016/j.ejor.2018.06.002

[24] R. C. Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

[25] Antón Rey, Manuel Prieto, José Ignacio Gómez, Christian Tenllado, and José Ignacio Hidalgo. 2018. A CPU-GPU Parallel Ant Colony Optimization Solver for the Vehicle Routing Problem. In *Applications of Evolutionary Computation - 21st International Conference, EvoApplications 2018, Parma, Italy, April 4-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10784)*, Kevin Sim and Paul Kaufmann (Eds.). Springer, 653–667. https://doi.org/10.1007/978-3-319-77538-8_44

[26] Yves Rochat and Éric D. Taillard. 1995. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics* 1, 1 (1995), 147–167. https://doi.org/10.1007/BF02430370

[27] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis. 1977. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM J. Comput.* 6 (1977), 563–581.

[28] Anand Subramanian, Eduardo Uchoa, and Luiz Satoru Ochi. 2013. A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research* 40, 10 (2013), 2519–2531. https://doi.org/10.1016/j.cor.2013.01.013

[29] Paolo Toth and Daniele Vigo. 2014. *Vehicle routing: problems, methods, and applications.* SIAM.

[30] Eduardo Uchoa. 2022. 12th Implementation Challenge: Track - Capacitated Vehicle Routing Problem. http://dimacs.rutgers.edu/programs/challenge/vrp/cvrp/. [Online; accessed 21-Jul-2022].

[31] Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, Anand Subramanian, and Ivan Lima. 2014. CVRPLIB: Capacitated Vehicle Routing Problem Library. http://vrp.galgos.inf.puc-rio.br/index.php/en/. [Online; accessed 19-Jul-2022].

[32] Eduardo Uchoa, Diego Pecin, Artur Alves Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. 2017. New benchmark instances for the Capacitated Vehicle Routing Problem. *European Journal of Operational Research* 257, 3 (2017), 845–858. https://doi.org/10.1016/j.ejor.2016.08.012

[33] Thibaut Vidal. 2022. Hybrid genetic search for the CVRP: Open-source implementation and SWAP* neighborhood. *Computers & Operations Research* 140 (2022), 105643. https://doi.org/10.1016/j.cor.2021.105643

[34] David P. Williamson and David B. Shmoys. 2011. *The Design of Approximation Algorithms.* Cambridge University Press. http://www.cambridge.org/de/knowledge/isbn/item5759340/?site_locale=de_DE

[35] Pramod Yelmewad and Basavaraj Talawar. 2021. Parallel Version of Local Search Heuristic Algorithm to Solve Capacitated Vehicle Routing Problem. *Clust. Comput.* 24, 4 (2021), 3671–3692. https://doi.org/10.1007/s10586-021-03354-9