Fast Trips: A scalable Insertion Operator Approach for Ridesharing over Time-Dependent Road Networks

by

Aaditya Mukherjee
B.Sc., University of Massachusetts Lowell, 2021

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

University of Victoria

We acknowledge and respect the Lək ʷəŋən (Songhees and Xʷsepsəm/Esquimalt) Peoples on whose territory the university stands, and the Lək ʷəŋən and W̱SÁNEĆ Peoples whose historical relationships with the land continue to this day.

Fast Trips: A scalable Insertion Operator Approach for Ridesharing over Time-Dependent Road Networks

by

Aaditya Mukherjee
B.Sc., University of Massachusetts Lowell, 2021

Supervisory Committee

Dr. Sean Chester, Supervisor
(Department of Computer Science, University of Victoria)

Dr. Mario A. Nascimento, Co-supervisor
(Khoury College of Computer Science, Northeastern University)

Abstract

Effective Route planning for shared mobility (RPSM) is crucial for optimizing the goals of transportation services such as ridesharing, logistics, and food delivery. Route planning requires online integration of new transportation requests into existing routes of transportation workers while accounting for real-world conditions such as traffic congestion, variable travel speeds, and changing demand patterns. A core component of route-planning systems is the insertion operator, a state-of-the-art method that integrates new transportation requests into existing worker routes with minimal additional travel time. Although effective and fast for route-planning simulations on static road networks, route-planning simulations experience significant performance degradation when applied to real-world, time-dependent road networks (TDRNs), where travel times between roads fluctuate due to varying traffic conditions.

This thesis addresses this scalability challenge by introducing an informed approach to partitioning the data used by the insertion operator into separate, disjoint batches. I propose a partioning method utilizing K-means clustering complemented by an opportunistic allocation of workers to clusters. This method reduces the large number of shortest path query invocations inherent to time-dependent insertions, significantly decreasing RPSM simulation speeds without sacrificing, and in some cases even improving the quality of the solutions.

Through extensive experimental evaluations using large-scale, real-world datasets from major Chinese cities, the proposed method is compared against the sequential time-dependent insertion operator. The results indicate a minimum of 7X acceleration in RPSM simulation times and maximum speedups up to 24X, while consistently matching or surpassing the original insertion operator in terms of solution quality. Furthermore, the flexibility of the clustering approach allows for customizable trade-offs between simulation speed and service quality, ensuring adaptability to diverse operational goals of transportation services.

Ultimately, this thesis offers a scalable, adaptable, and computationally efficient insertion operator framework capable of handling realistic scenarios in dynamic shared mobility environments, providing valuable tools for transportation and logistics companies seeking operational optimization.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgments

I would like to thank my mother and father for their endless support and belief in me. I would like to thank my grandmother as well, who has been a constant source of strength throughout my life. Without them, I could not have completed my Master's degree. I am sincerely grateful to my supervisors, Dr. Sean Chester and Dr. Mario Nascimento, for seeing the potential in me, for working alongside me, and for giving me skills and guidance that I will carry forward. To my close friends and colleagues at UVic—thank you for encouraging me, motivating me, and reminding me to keep pushing through even when things felt difficult.

Most of all, I want to thank my wonderful partner, Noella, who gave me the space to finish writing, stood by me through moments of great stress, and supported me with love and patience. I love you.

Each of you, in your own way, helped make this thesis a reality. I am forever grateful.

Aaditya Mukherjee,
Victoria, Sunday 27th April, 2025.

Chapter 1

Introduction

Route planning is fundamentally challenging. Whether it involves an individual commuting to and from work, a food delivery driver picking up orders from various restaurants and dropping them off at different homes, or a logistics company such as Canada Post delivering a vast number of packages on the same day, there is always the critical question: What is the most efficient route? Addressing this problem has been a noteworthy concern for computer scientists considering how many of them have brought significant contributions towards solving this problem ([Rob49] with Traveling Salesman, [Dij59] with the Shortest Path Problem, [CL03] with the Dial-A-Ride problem, etc.)

Route-planning for shared mobility (RPSM) is an efficient way of route planning where transportation services (such as taxis or delivery trucks) are shared among multiple users (multiple passengers in an Uber pool, Amazon last-mile deliveries, etc.) [Ton+22]. RPSM involves dynamically planning routes for workers (such as taxi or delivery drivers) who must accommodate incoming requests specifying precise pickup and drop-off locations. RPSM finds every worker a route, i.e., a sequence of locations for picking up and dropping off passengers and parcels. Effective route planning is crucial for transportation services so that they can maximize profits, mitigate customer dissatisfaction, etc. The insertion operator is the core, state-of-the-art operation for dynamically inserting the origin and destination of new requests into a worker's current route [GZC24b]. It thrives in computing the optimal route for the worker to serve requests (optimal here refers to the minimum increased travel time to serve a newly appeared request).

1.1 Insertion Operator: the fundamental solution to route-planning in shared mobility

Consider the following example: A ride-sharing taxi driver has been assigned a single passenger request R_1 which has origin and drop-off locations o_1 and d_1. The driver has just picked up the passenger o_1 and is planning on dropping them off at d_1. This is depicted in the leftmost figure of Figure 1.1. The worker's current route is (o_1, d_1) It is evident that the shortest path for this request is $\langle A, D, E, F \rangle$ (the edge weights represent travel time units). A few seconds after R_1 gets picked up, a second passenger request R_2 appears on the system, with o_2 and d_2 as the origin and destination locations (middle figure in Figure 1.1). The insertion operator will now find the best way to "insert" R_2's locations into the worker's current route so that the increased travel time for the worker is minimized. There are 3 scenarios in which this R_2's origin and destination locations can be inserted in the worker's route:

- Scenario 1: pick up R_2 after picking up R_1, then drop R_2 off **after** dropping off R_1. The worker's route would be: (o_1, o_2, d_1, d_2)
- Scenario 2: pick up R_2 after picking up R_1, then drop R_2 off **before** dropping off R_1. The worker's route would be: (o_1, o_2, d_2, d_1)
- Scenario 3: pick up and drop off R_1 first. Then pick up and drop off R_2. The worker's route would be: (o_1, d_1, o_2, d_2)

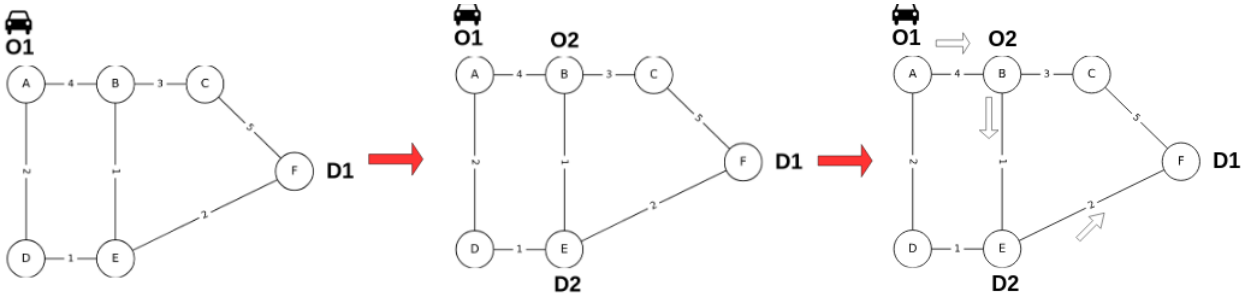


Figure 1.1: Insertion Operator workflow with best insertion route: $\langle o_1, o_2, d_2, d_1 \rangle$

If you calculate the shortest arrival times for each scenario by adding the detour times of the full worker's paths for these scenarios, you can see that scenario 2 would yield the least increase in travel time for the worker (rightmost figure in Figure 1.1). This is seen in Table 1.1.

Worker Route	Path	Shortest Path Travel Times
(o_1, o_2, d_1, d_2)	$\langle A, B, C, F, E \rangle$	14
(o_1, o_2, d_2, d_1)	$\langle A, B, E, F \rangle$	7
(o_1, d_1, o_2, d_2)	$\langle A, D, E, F, E, B, E \rangle$	9

Table 1.1: Potential Worker Routes, associated Shortest Paths, and Shortest Travel Times for the Insertion Operator to insert (o_2, d_2)

The insertion operator logically enumerates all scenarios, or more specifically, all potential insertion positions, and finds the best position to insert a new request into a worker's route. A wide variety of existing solutions have enhanced, and optimized the insertion operator to make it scalable for a large number of requests and for dense road networks of cities like New York City, Beijing City, and Chengdu [Ton+22][Ton+18][Hua+14]. But, all these solutions assume a static road network. Static road networks are not ideal when it comes to representing real road networks of cities.

It is simpler to calculate the shortest travel times by simply adding the additional detour times after insertion for different scenarios in Table 1.1. For instance, after inserting d_2 in scenario 2 into the third position of the worker's route, directly adding up the shortest distance times, i.e, $dis(o_1, o_2) + dis(o_2, d_2) = 4 + 1 = 5$ gives you the static shortest travel time to get to d_2.

One way to represent real road networks is by making the edges of a graph have multiple numerical values associated with them, suggesting that the travel cost between nodes of this edge varies over time. These variations are closer to resembling real world scenarios like rush hour, speed-limits during daytime hours in school zones, etc. Such a road network is known as a **Time-Dependent Road Network**. A simple illustration of this is seen in Figure 1.2, which is a re-iteration of the static road network of Figure 1.1 but with edge weights varying based on time units, as seen in Table 1.2. Here, the edge weights are represented by the set of $\{(Time, Weight)\}$ pairs, with the constraint that Time values for every pair has to be unique. For the sake of simplicity in this example, this can be seen as a piecewise constant function (or a step function) with x = time, and y = weight pair associated with time.

1.1.1 Time-Dependent Insertion Example

Consider the same example from Figure 1.1 on a time-dependent road network now, where the taxi arrives at o_1 at time $t = 40$. The 3 possible insertion scenarios from that example still hold true but the shortest travel time calculations change drastically. The route (o_1, o_2, d_2, d_1), whose shortest time was calculated by

the simple addition of shortest travel time distances $dis(o_1, o_2) + dis(o_2, d_2) + dis(d_2, d_1)$ does not hold true as the edge weights on a TDRN can change depending on the arrival time of the taxi to that location. The path that this route takes is $\langle A, B, E, F \rangle$. To compute the shortest arrival time for this path for a TDRN, we need to now:

- Keep track of the time of departure for the taxi at the current node.
- Look at the weight associated with this time of departure of the subsequent node that you are trying to visit.
- Calculate the arrival time at that node by adding this weight to the departure time.

For example, to calculate the shortest arrival time for the route (o_1, o_2, d_2, d_1): To go from $A \rightarrow B$ at $t = 40$, the weight at $t = 40$ for edge $E_{A,B}$ is 10 (from table 1.2). So arrival at B happens at t $= 50$. For going from $B \rightarrow E$ at $t = 50$, the weight associated with edge $E_{B,E}$ for that time is 30. Therefore, arrival at E happens at 80. $E \rightarrow F$ at $t = 80$, the weight is 3 (from $E_{E,F} : (20, 5)$), arrival at F happens at 82. The shortest travel time for this scenario is 82! It is computed by the nested function: $query(d_2, d_1, (query(o_2, d_2, query(o_1, o_2, departure\_time(o_1)))))$ Similarly, we find that for scenario (o_1, o_2, d_1, d_2), the shortest arrival time is 75 (we disregard scenario 3 because it involves backtracking). Thus, in the time-dependent case, scenario 1 is better than scenario 2!

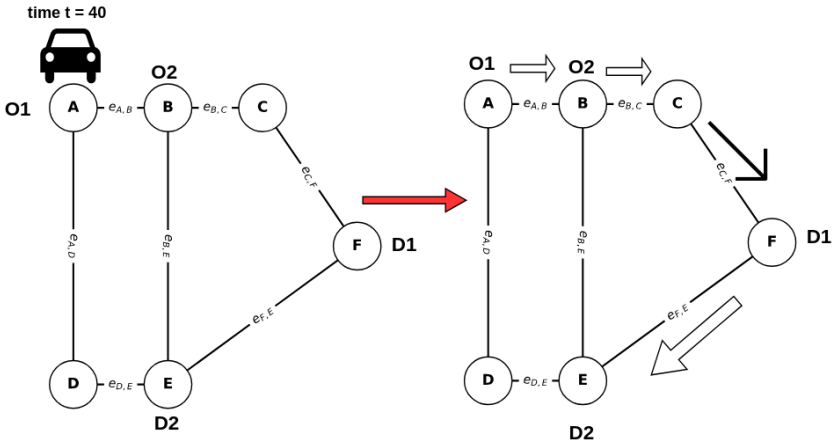


Figure 1.2: Insertion operation over a Time-Dependent graph with varying edge weights depending on the time

The additive property of simply adding the travel times of detours can only be established in static road networks. As seen from a TDRN insertion scenario,

Edge	Weights {(Time, Weight)}
$e_{A,B}$	{(0,4), (20,4), (35,10), (65,10)}
$e_{A,D}$	{(0,2), (10,3), (40,20), (50,14), (65,2)}
$e_{B,C}$	{(0,3), (65,3)}
$e_{B,E}$	{(0,6), (10,1), (65,2)}
$e_{C,F}$	{(0,5), (40,20)}
$e_{E,F}$	{(0,2), (20,5), (60,2)}
$e_{D,E}$	{(0,1), (65,1)}

Table 1.2: Time-dependent edge weights for the graph.

sequentially querying the delay time at each location based on the new route gives the accurate shortest travel time in a time-dependent road network.

Key limitation: As seen in the above example, you have to keep track of the departure time at each node, and look up the edge weight associated with this departure time for every subsequent node. You do this three times, From this small example, denoted by the 3 nested query() invocations, with which we calculate the accurate travel times of the worker between nodes. You cannot simply add the additional detour times like you could with static road networks. Additionally, you also have to store the set of (edge weight, time) pairs for every edge in a TDRN. This TDRN only had 6 vertices and 7 edges. If scaled to real-world road networks of dense cities like Chengdu (China), we would be dealing with over 400K vertices and 900k edges! The number of query invocations would therefore increase dramatically.

1.1.2 Fastest Insertion Operator over Time-Dependent Road Networks

Optimized insertion operators [Xu+19][Ton+18][Hua+14] that finish route-planning simulations in linear-time complexity w.r.t the number of requests deteriorate into cubic time complexity when dealing with TDRNs due to the sheer number of query invocations. [GZC24b] propose a **linear-time Insertion Operator over Time-Dependent Road Networks** and are able to show that their solution gives an asymptotically faster algorithm that completes in linear time with respect to the number of requests assigned to the worker. The problem is, although their insertion time per request is 44.5X faster than the state-of-the-art static road network algorithms (due to an average 90% reduction in shortest arrival time query invocations), the total running time for an RPSM simulation over a large set of requests is significantly slow. For example, it takes 4.5 hours to complete a simulation of 15,000 ride-sharing requests with 125 workers over

the road network of Chengdu City. The number of times the shortest arrival time query gets invoked is 6.654428×10^6.

Data collected by ride-hailing company, DiDi Chuxing, indicates that the average daily number of ride-hailing trips in Chengdu exceeds 240,000[WN21]. This highlights the impracticality of the time-dependent insertion operator. It would take at least a day to complete a route-planning simulation with the time-dependent insertion operator over dense cities. It is **unscalable** with real-world worker forces and trip requests. Any company that would like to use insertion operator software would ideally like to simulate a large number of requests at a faster rate to make observations on worker placement, request served hotspots, the average time of requests per day, etc to maximize their profits. This current solution does not help to achieve that within a reasonable time.

1.2 Proposed Solution

In this thesis, I propose a scalable, time-dependent Insertion Operator approach that can run fast RPSM simulations over dense road networks. These quick simulations can be used by logistics, ridesharing, food delivery companies to maximize their route-planning goals. It achieves that through:

- Speeding up the total running times of simulations by at least a factor of 5X through finding an informed way to reduce query invocations for the time-dependent insertion operator.
- Preserving the solution quality of the simulations (even after lowering query invocations) by maintaining or in some cases, exceeding the number of requests served by the sequential time-dependent insertion operator

The effectiveness of this balance between solution quality and simulation running time is thoroughly tested and demonstrated through extensive experimentation across Ridesharing and Logistics route-planning scenarios.

Essentially, my aim is to provide a scalable insertion operator that can be used to achieve any RPSM goal over real world data. These goals could be minimizing travel time[GZC24b], could be maximizing served rate [Ton+22], maximizing revenue, or a combination of those three [Ton+18]. Regardless of the goal, my approach speeds up simulation times.

How do we achieve this kind of scalability? We partition the trip requests and workforce data. We do this by using a partitioning algorithm called K-means clustering, where we specify the number of partitions we want to divide the requests into. Through experimentation on real-world road networks and requests datasets, we observe how the solution quality and execution times change for

different number of partitions. Through this we try find a sweet spot for maintaining the solution quality of the time-dependent insertion operator while gaining potential execution speed reductions. We also look for the optimal criteria of dividing the road network. We try partitioning based on the pickup locations and dropoff locations of requests to see which type would result in the most requests served. Each set of drivers can only serve requests in their respective zones and cannot attempt to serve other zones' requests. After that, we simply assign these 'zones' to independent CPU cores, taking advantage of parallel computing concepts. The major key of our partitioning is that no data is shared among threads. This results in the insertion operator running concurrently on all cores with no locks or synchronization issues.

Studies have been conducted to parallelize the insertion operator or use batch processing before to bring scalability to RPSM but it has only been performed on static road networks[Ota+17] [Zuo+21]. In the parallelization study, they use locks, atomic operations, and have a synchronization step due to shared data of the taxi trajectories. As mentioned, there have also been numerous optimizations to the static insertion operator[Ton+22]. It is also observed by [GZC24b], that these optimized solutions degrade when TDRNs get introduced. Our approach works on TDRNs and aims to produce a wait-free parallel approach with CPU cores working on independent data of the road-network which means that we ensure that there is no synchronization overhead.

In the experimental setting we treat the full request log as historical (offline) data. To approximate how the system would be configured in practice, we bootstrap the clustering with the first batch of requests that arrive in the trace—that is, the subset released during an initial warm-up window (e.g., the first 5–10 minutes of the day). This window provides a representative sample of the spatial–temporal demand pattern without presuming knowledge of the entire future stream. The resulting partitions remain fixed while the remainder of the requests are replayed, mirroring the way [Ton+22] pre-computes shortest-path tables from an early snapshot of demand before serving subsequent queries.

We subject our partitioned and parallel insertion operator to a vast experiment suite. We use large, time-dependent road network data used by [GZC24b] from major cities in China like Shanghai, Haikou, and Chengdu for different RPSM domains like logistics and ridesharing and compare our insertion operator to their linear-time insertion operator in terms of percentage of requests served and execution time of the algorithm. We use spatial and spatiotemporal criteria for k-means clustering and find which one results in a higher served rate. We allocate workers to request partitions in an opportunistic manner that maximizes the requests served rate. By introducing this partitioning, we introduce flexibility and customization for the user to determine the right k number for them. We introduce a trade-off between the requests served rate and the simulation.

1.3 Contributions

This thesis focuses on making dynamic ridesharing problem simulations scalable for dense, time-dependent road networks with a large arsenal of trip requests and worker base while maintaining the served rate. It also studies and reports the potential bottlenecks that make existing solutions inefficient for real-world data. It achieves the following:

- An informed spatiotemporal partitioning of trip requests made within large cities using K-means clustering and ensuring strict captivity of request and worker data to their respective "categories", ensuring no shared data/memory when doing dynamic ridesharing concurrently, **resulting in consistent decrease in RPSM simulation running time**.
- A **wait-free, task parallel** approach to the Insertion Operator that works concurrently on partitioned data and runs with no locks or synchronization requirements.
- An evaluation of various clustering criteria like clustering based on spatial features (pickup locations, dropoff locations), on spatiotemporal features (time-dependent shortest path travel times for requests), and picking the best one based on these evaluations **for preserving the original served rate**.
- A small case study of the state-of-the-art Data Structure for storing TDRNs: **Time-Dependent G-Tree** and its bottlenecks to tackle the scalability issue for RPSM over TDRNs.

Chapter 2

Background

This thesis studies batch insertions of new requests in route planning for shared mobility (RPSM) over time-dependent road networks (TDRNs). This chapter reviews background concepts and formally introduces the batch insertion operation over time-dependent road networks problem.

2.1 Time-Dependent Road Networks

Time-dependent road Networks are represented as a directed graph $G(V, E, F)$. V is a set of vertices and each vertex $v \in V$ represents a geo-location. $E \subseteq V \times V$ is the set of edges . Each directed edge $(u, v) \in E$ represents a road segment between u and v [GZC24b][WLT19].

Each edge is associated with a non-negative weight function $f_{u,v}(t) \in F$, where t denotes the current time, and $f_{u,v}(t)$ denotes the travel time from vertex u to vertex v at time t. The function $query(u, v, t)$ is used to find the shortest arrival time to v from u, if beginning at time t.

Piecewise Linear Functions (PLFs) are adopted to model the weight functions in a TDRN. The weight function, $f_{u,v}(t)$, is modelled as a set of interpolation points $P = \{(t_1, w_1), (t_2, w_2), ..., (t_k, w_k)\}$ where k is the number of interpolation points. Each point (t_i, w_i) denotes that it takes w_i unit time to travel from u to v at time t_i. It is used to indicate the appropriate travel time based on the edge weight assigned to it at that time of the day. For example, when a worker departs from u at time $t \in [t_1, t_2)$, their travel time is calculated with the weight function

$$f_{u,v}(t) = w_1 + (t - t_1)\frac{w_2 - w_1}{t_2 - t_1}.$$

. The formula of the weight function is given by:

$$f_{u,v}(t) = \begin{cases} w_1 + (t - t_1)\frac{w_2 - w_1}{t_2 - t_1}, & t_1 \leq t < t_2 \\ w_2 + (t - t_2)\frac{w_3 - w_2}{t_3 - t_2}, & t_2 \leq t < t_3 \\ \quad \vdots & \\ w_{k-1} + (t - t_{k-1})\frac{w_k - w_{k-1}}{t_k - t_{k-1}}, & t_{k-1} \leq t \leq t_k \end{cases} \tag{2.1}$$

where the time domain is $[t_1, t_k]$.

Property of Time-Dependent Road Networks

Existing work establishes the **first-in-first-out (FIFO)** property of time-dependent road networks[WLT19][GZC24b]. It basically says that if two people depart from the same node u to a destination node v at different times, the person who departed earlier will reach v first, implying

$$t_1 + f_{u,v}(t_1) \leq t_2 + f_{u,v}(t_2)$$

for departure times $t_1 \leq t_2$, and for any weight function. Thus, PLF functions fit this scenario for computing travel times.

2.2 Route Planning for Shared Mobility

RPSM is about matching multiple passengers or packages to shared vehicles while considering factors like time, cost, and road conditions. Ridesharing services like Uber Pool and Logistics services like Amazon last-mile deliveries are examples of this phenomenon. An important objective of RPSM is to determine the best routes that minimize the travel time of workers while maximizing the number of requests served. RPSM problems consist of the following items.

Time-Dependent Shortest Path Query

Definition: For a given source vertex u, destination vertex v, and a departure time t from u, the time-dependent shortest path query $query(u, v, t)$ computes and returns the exact minimum travel time between u and v [WLT19].

Requests

Definition: A request represents a new ridesharing trip request from a passenger (or multiple passengers) or a new food delivery order that appears on a shared mobility platform. For example, when a person sends a request on Uber to get picked up and dropped off at a certain location. A request is denoted by

$r = \langle o_r, d_r, t_r, e_r, c_r \rangle$. A request appears at time t_r at origin $o_r \in V$. Its destination is $d_r \in V$. This request needs to be served before the deadline time e_r. c_r is the size of the request. Size, in this context, refers to the number of passengers, food delivery items, parcels, etc. A request is said to **served** if it (person, food, parcel) is picked up by a worker from o_r after time t_r and dropped off at the destination d_4 before the deadline time e_r.

Worker

Definition: A worker is a person working for transportation services. This person is responsible for picking up and delivering goods from one place to another. A taxi driver, food delivery driver, or courier is an example of a worker.

A worker is denoted by $w = \langle c_w, S_R, t_0 \rangle$. c_w is the worker's capacity, which represents the capacity of the vehicle. This would be the number of passengers (ridesharing) or the amount of packages (logistics) you can fit inside the worker's vehicle. S_R is the current route of the worker at time t_0, represented by a sequence of vertices in V. $S_R = \langle v_0, v1, ..., v_n \rangle$ where v_0 is their current location and $v_1, v_2 ... v_n$ is either a destination or an origin of an unserved request in R. R is the set of unserved requests assigned to the worker. The arrival time at v_k is given by:

$$arr[v_k] = \begin{cases} t_0, & k = 0 \\ query(v_{k-1}, v_k, arr[v_{k-1}]), & k > 0 \end{cases} \tag{2.2}$$

A worker's route always remains feasible, i.e., all requests assigned to a worker will get served before the request's deadline time. A worker does not pick up any request that is not servable to them.

Insertion Operation

Definition: A newly appeared request, if not rejected due to time or capacity constraints, will be assigned to the worker who can serve the request with a minimum increase in travel time within their current route. An **insertion operation** is used to calculate the minimum increased travel time [Ton+22].

2.2.1 Time-Dependent Insertion Operator

[GZC24b]Given a worker w and a request r^+ that newly appears at time t_{r^+} , the insertion operator finds a new route S^* for this worker with the minimum increased travel time obj^* in real-time to serve all the requests $R^+ = R \cup \{r^+\}$ while satisfying the following constraints:

- **Completion Constraint:** All the requests must be served.
- **Order Constraint:** The relative orders of vertices in S_R, i.e., drop-off locations should not appear before their pickup locations.
- **Deadline Constraint:** The worker must deliver all the requests to their destinations before their deadlines.
- **Capacity Constraint:** At any time, the total size of requests that this worker has picked up but not delivered is no larger than the worker's capacity.

The insertion operator $insert(i, j)$ indicates adding the origin and destination of the new request into positions i and j of the current route S_R, directly before the vertices v_i and v_j, respectively, where $1 \leq i \leq j \leq n + 1$.

2.3 Batch Insertion Operation over Time-Dependent Road Networks

Based on the concepts above, we describe a **batch Insertion Operation problem over Time-Dependent Road Networks**. This thesis, to the best of my knowledge, is the first work to explicitly study the batch application of the insertion operator over TDRNs.The problem definition is as follows.

Given a set of requests $R = \{r_1, r_2, ..., r_n\}$, a set of workers $W = \{w_1, w_2, ..., w_m\}$, where the initial route $S_R = NULL$ for $w \in W$, each with a fixed capacity capable of serving multiple requests, and a time-dependent road network G, the Batch Insertion Operator returns the worker set $W^* = \{w_1^*, w_2^*, ..., w_m^*\}$, with the updated assignment of requests to workers. Each worker $w^* \in W^*$ contains a feasible route S_R^*, represented as an ordered list of pickup and drop-off locations of requests in R assigned to those workers.

The batch insertion operation over TDRNs consists of the following objectives:

- **Maximize Requests Served Rate:** Serve as many requests from set R as possible.
- **Minimize Total Detour Travel Time:** For each inserted request, minimize the additional travel time experienced by the workers,
- **Minimize Simulation Time:** Compute worker routes rapidly to allow practical application for large-scale RPSM simulations.

The batch insertion operation satisfies the following constraints:

- **Capacity Constraint:** At any time, the number of simultaneous requests being served by any worker $w*$ cannot exceed the vehicle capacity c_i.
- **Deadline Constraint:** Requests must be completed by their specified deadlines e_i to be considered delivered.
- **Order Constraint:** Each route sequence S_R^* of a worker must respect logical ordering of requests. Specifically, each request's origin location must be visited before its respective destination location.

Chapter 3

Related Work

Since the insertion operator is the principal operator for RPSM problems, there have been numerous attempts to improve the insertion operator for running RPSM simulations at a faster rate. This chapter goes over some papers that try to optimize the Insertion Operator, introduce parallelism to RPSM simulation algorithms, and also goes over papers that partition a road network using k-means clustering.

3.1 Optimization efforts to Insertion Operator for Shared Mobility route-planning

[Ton+22] presents a wide variety of different insertion operator algorithms, with the best one being the **prophet insertion**, which "predicts" or forecasts where new requests might appear. This paper uses metrics such as **requests served rate, total running time**, unified cost, etc. to measure the performance of their algorithm against 5 other insertion algorithms to prove that theirs outperforms the rest.

In their varied scenarios for simulations, the **total running time** does not exceed 10^4 seconds for their Chengdu simulations. In fact, the T-share algorithm takes around 10^1 to 10^2 seconds for running simulations while varying the workers from 1000 to 2000. This certainly shows that there does exist a fast insertion operator, the caveat being that these algorithms only work on **static road networks**. [GZC24b]'s paper, the one that we focus on and build upon, uses time-dependent road networks. The usage of time-dependent road networks makes it more viable to be deployed in real-world applications since static road networks do not consider the dynamic nature of roads which can be affected by traffic congestion, weather conditions, etc. Therefore, their total time is not as low as T-share, but it is still the fastest algorithm for processing time-dependent

road networks. Our clustered approaches result in an even faster running time for simulations due to the reduction of search space and parallel computing. Our approach increases the feasibility for the Insertion Operator for any ride sharing or logistics company that might want to run RPSM batch insertions for their vehicles.

More importantly, it gives us accurate parameters on how to calculate the delivery deadline times for requests. The deadline time is calculated as the sum of: the release time of a request, the travel time between its origin and destination, and a specified parameter depending on the dataset. Mathematically, it looks like:

$$e_r = \text{additional\_detour\_time} + t_r + \text{query}(u, v, t))$$

Where:

- e_r: Delivery deadline time
- query(u, v, t): Shortest arrival time query between u and v at time t
- t_r: Request release time

[GZC24b] design a linear-time insertion operator that does RPSM with Time-Dependent Road Networks (TDRNs). Although this insertion operator has linear-time complexity w.r.t the number of requests that get assigned to a worker, it is prohibitively expensive. For example, it takes **581 minutes** to run a simulation for 1000 requests with infinite deadlines, and 300 workers to fulfill logistics requests in the city of Shanghai. This design, although close to real-life, is not scalable at all. [Sch25] shows that there are over 600,000 ridesharing trips a day in a big city like New York City, and over 120,000 of yellow cab trips. A more scalable insertion operator is needed that can run simulations on these large datasets. Their algorithm is also single-threaded, whereas ours leverages multiprocessing to partition the data load onto multiple cores, leading to a faster execution time.

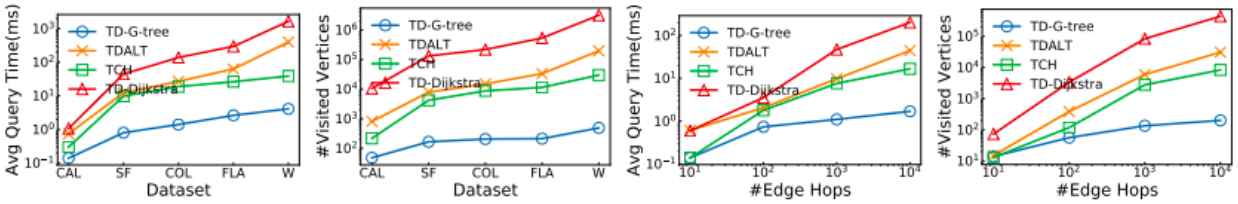
It is also important to consider factors such as the initial location of the worker. [Dai+22] demonstrates the importance of vehicle depots for autonomous taxi ride-sharing problems. It suggests that the selection of a vehicle depot location can lead to more requests served, is directly related to the operations cost of a business, and provides flexibility to taxi drivers. But in modern times, especially with ride-sharing services like Uber, Lyft, DiDi Chuxing, etc., the drivers have flexible hours and do not start at a centralized location. [Ton+18] explores this scenario in their experiments for testing their greedy dynamic programming insertion algorithm (one of the algorithms [GZC24b] compares against). Our work only focuses on applying the batch insertion operation to the depot scenario, but

it has the ability to be applied to both. The only thing that could change is the percentage of requests served. [Ton+18]'s algorithm considers three main RPSM objectives that they are trying to optimize over static road networks: Total Travel Distance (minimize), Requests Served Rate (maximize), and Total Revenue (maximize). Our solution minimizes the Total Travel Distance for workers and the total running time for simulations. We focus on studying the batch insertion operator for a singular RPSM goal over TDRNs.

3.2 TD-G-Tree

The **TD-G-Tree** data structure ([WLT19]) and its enhancements [Li+22][Dan+23][Wan+22] are novel data structures used for indexing a real-time/time-dependent road network.

It is used for its **TDSP query()** operation by [GZC24b]. As a result, we also use this shortest arrival time query *query()* in our research. [WLT19]'s experiments show that this TDSP query is the **fastest** shortest path operation over TDRNs.They conducted experiments over datasets ranging from (V, E) : $(21048, 43386)$ for CAL to $(6262104, 15248146)$ for W(Western USA). The comparison between their query() operation and others is seen in the visualization that they made (see Figure 3.1). Their TDSP query processing outperforms all the other time-dependent query algorithms.



(a) Random query (b) Random query vis# (c) Vary edge hops (FLA) (d) Edge hops (FLA) vis#

Figure 3.1: Comparison on TDSP query processing by [WLT19]

This tree has some key drawbacks. With a denser, real-world network containing more vertices, edges, and interpolation points, the tree's construction time and TDSP query time rises proportionally. For their Figure 3.1, the average interpolation points is 3 per dataset. For dense cities like New York City and Beijing City, [Li+22]'s experiments show that the TD-G-Tree index construction can take over 10^3 to 10^4 seconds. The tree sizes for both cities are **15 GB and 22 GB** respectively. The number of interpolation points per requests (p) also has an impact on the construction performance. If you increase the interpolation points from 2 to 6, the index size (in GB) increases linearly with it. The construction time (in seconds) grows quadratically.

In addition to the construction bottleneck, [Dan+23] demonstrates that the TDSP query() is prohibitively expensive in terms of space and time. [Wan+22] also mention how the TDSP query() time increases as the dataset grows. Thus, for a big dataset like Chengdu City, which contains 423434 vertices and 913718 edges, the query time is over 2.62 milliseconds.

The TD-G-tree remains a state-of-the-art solution for TDSP query processing on TDRNs but shows increase in computational cost with dense road networks with a higher average number of interpolation points per edge.

The time complexity of $query(u, v, t)$ is

$$O(\log_2^2 \kappa_f \cdot |V| \cdot \log_2^2 \alpha(T))$$

, where k_f is the fanout of the TD-G-Tree, |V| is the number of vertices, and $\alpha(T)$ is the number of linear pieces of a PLF on time domain T.

The number of vertices |V| and the number of interpolation points (in turn, |E| as well) for edges play a significant role in the time-complexity of tdsp computation. Basically, having a larger |V| and |E| will result in more TD-G-Tree nodes to form, which can (and usually does) lead to a larger traversal of tree nodes for shortest path queries, and thus, a longer processing time.

We conduct experiments and report detailed numbers for the construction and querying bottleneck in the thesis, as seen in Chapter 7.

3.3 Parallelization of RPSM algorithms

[Ota+17] runs a simulation algorithm for taxi-ridesharing and focuses on scalability. Although they simply analyze ride-sharing in taxis using their simulation framework, they do take note of the sheer number of taxi rides that occur in NYC. Their algorithm is similar to [GZC24b] in the sense that each simulation step depends on the results of the previous step. For example, a request can only be investigated after the previous request has been investigated, a driver has been assigned, and every other driver's position has been updated. This data-dependency is the reason why the insertion operator and their simulation algorithm are non-trivial to parallelize. However, the authors do distribute some workload to multiple cores. This is where we take inspiration from as well. They distribute a set of taxis across multiple cores and all workers synchronize after picking up a request. This thesis, in contrast, aims to remove synchronization completely and the cores do not need to communicate at all to finish their work.

3.4 Road Network Partitioning

Some papers work on finding the optimal partitioning of a road network, similar to our work. [LX20] uses canopy k-means clustering using the central (longitude/latitude) coordinates, average speed, and average density of a road network to divide a whole road network into four sub-areas. They then do optimal macroscopic fundamental diagram (MFD) tests on their algorithm to show that their canopy k-means partitioning algorithm is better than regular k-means partitioning. [Anw+14] studies useful partitioning strategies for road networks and uses k-means clustering to form condensed supergraphs. In contrast, our work:

1. partitions the requests on a road network using k-means clustering, based on a mixture of geospatial criteria. The road network itself remains unpartitioned.

2. We work with a time-dependent road network, while the other papers works on static ones.

3.5 Batch Processing in RPSM

[Zuo+21] uses graph clustering to pack requests together based on their edge weights. This is similar to our problem of batch insertion operation but the key differences are that they use graph clustering to partition their data while we use k-means clustering to partition ours, and we use TDRNs while they use static bipartite graphs.

Chapter 4

Naive Solutions

4.1 Chapter Outline

In this chapter, I describe the insertion operator algorithm, the data race issues that arise when I naively try to parallelize the requests by dividing it into chunks and assigning threads to each chunk, and finally introduce a partitioning of the requests data using k-means clustering with spatial data (2-dimensional coordinates representing request pick-up and drop-off locations) as the clustering feature.

4.2 The Sequential Insertion Operator Algorithm

The insertion operator algorithm takes as input the TDRN (represented by the td-g-tree data structure), the set of workers W, and the set of requests R. It iterates through individual requests one after the other. For each request, a simple radius check is done (single_search) to find nearby eligible workers that can potentially serve this request.

After these workers are identified, for each worker, it enumerates the potential positions to insert this request in the worker's current route. After passing the feasibility checks, it inserts the request in the position of the worker's route that results in the least amount of extra travel time. This optimal route $S*$ and its optimal detour time $obj*$ is noted. In the next iteration, this request is inserted (or tries to be inserted) in the next worker and the optimal route and detour time for this worker is also calculated. Then a comparison is made between this worker's optimal time and the best previous worker's optimal time to see which one's detour time is the least. This is done iteratively against all the workers

and after iterating through all of them, the worker that has the least detour time among all of them is selected as the worker that gets to serve this request.

The insertion operator workflow is displayed in Algorithm 1.

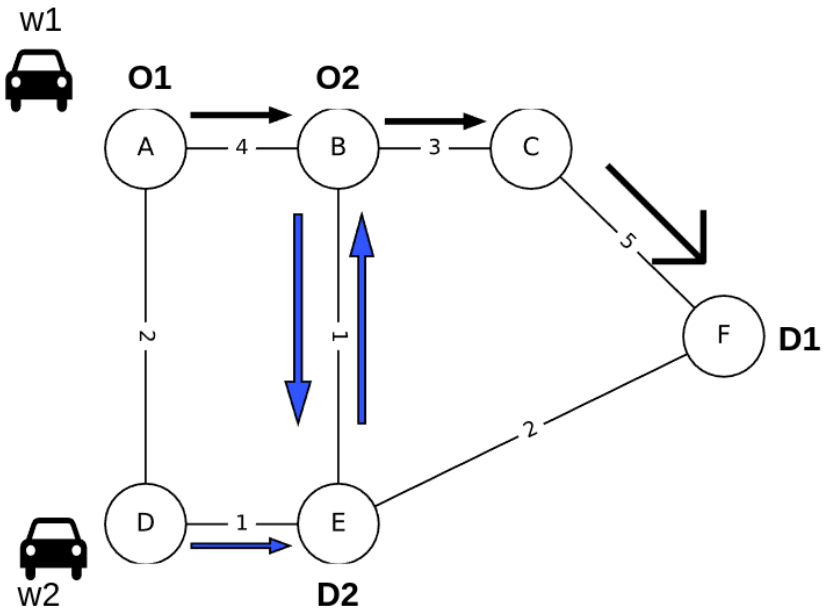


Figure 4.1: Assigning the optimal routes for workers w_1 and w_2 to serve requests r_1 and r_2. The black arrows are w_1's route. The blue arrows are w_2's route.

Example: Let's look at a variation of the example in Figure 1.1 in Figure 4.1. We can use a static road network here since the insertion operator follows the same principles made in TDRNs. Here we are given a set of workers $W = \{w_1, w_2\}$ and a set of requests $R = \{r_1, r_2\}$ with infinite deadlines. The insertion operator iterates through the requests in R one after the other.

1. Firstly, it looks at r_1. Both workers in w are eligible so it inserts r_1's origin-destination pairs into each worker's route and checks which one results in the shortest detour. For $S_1 = (o_1, d_1)$ (w_1's route) the detour time is 5. For $S_2 = (o_1, d_1)$, the detour time is 7. The detour time of w_1 is compared to w_2 and it is seen that w_1 has the least additional travel time. So worker w_1 gets selected to pick up r_1. S_2 remains empty.

2. After processing r_1, and updating the routes of the workers, we now look at r_2. Once again w_1 and w_2 are found eligible so the insertion operator iteratively inserts r_2 at the best position in the workers' current routes. For w_1, the best insertion route is found to be $S_1 = (o_1, o_2, d_2, d_1)$ with

detour time (or minimum time for completing both requests) as 7. For w_2's optimal route $S_2 = (o_2, d_2)$, the shortest time to complete the route is 3, which is better than the time taken by w_1's best route. Therefore, r_2 gets assigned to w_2.

The final routes of the workers 1 and 2 are $S_1 = (o_1, d_1)$, and $S_2 = (o_2, d_2)$, respectively. These routes are illustrated in Figure 4.1 via black (for w_1) and blue (for $w2$) arrows.

4.3 Naive Parallelization Attempt of Requests and Workers

This section describes the sequential insertion operator algorithm and illustrates how parallelizing the Time-Dependent Insertion Algorithm is non-trivial due to multiple interdependencies and shared data structures. A naive attempt to split the requests into n chunks and have multiple threads implementing the insertion operator on different chunks parallelly results in data races. Although it is true that theoretically, we could simply split the set of requests and have them be processed in parallel, the main issue is that **the sets W and R are shared data structures that are intertwined with each other and that the order in which requests are processed matters.** You can witness this in Algorithm 1.

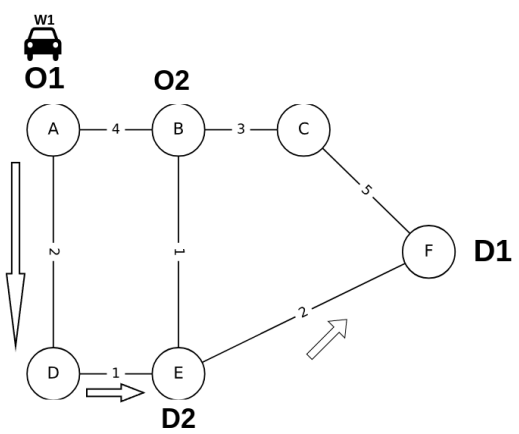
Splitting the outer loop in line 3 into parallel chunks, which iterates through the requests does not work and is explained in the following section.

Shared sets: W and R

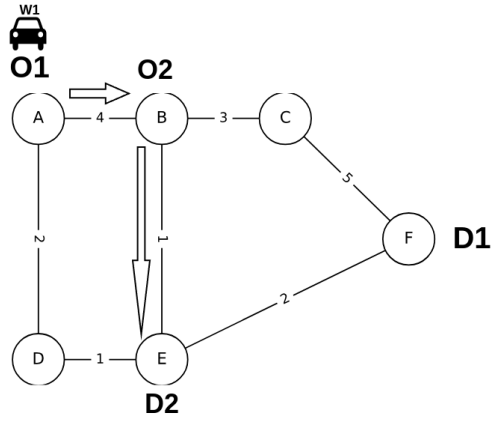
All drivers' positions are located on the map when iterating through a singular request through UpdateDriver(). This is done so eligible drivers can be found via SingleSearch(). If multiple threads are processing different requests in parallel, they may try to update the same best worker's state concurrently, which would lead to inconsistent or suboptimal results.

Algorithm 1 Time-Dependent Insertion Baseline [GZC24b]

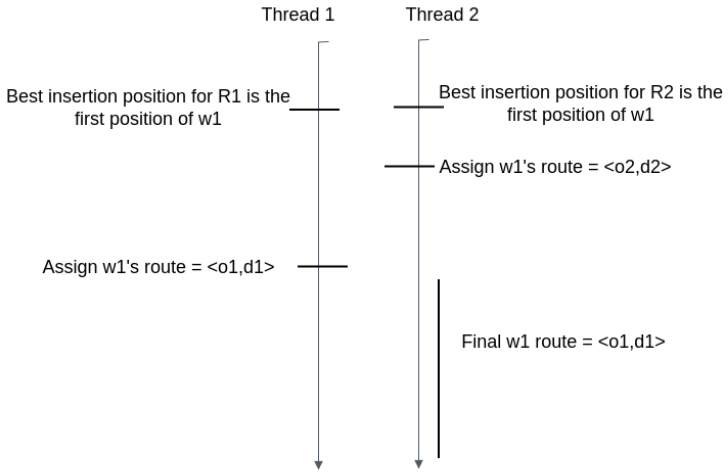
Input: Requests $R = \{r_1, r_2, ..., r_n\}$, Workers $W = \{w_1, w_2, ..., w_m\}$, TD-G-Tree
Output: Updated worker assignments and paths
for $pos \leftarrow 0$ to $n - 1$ **do**
    for $i \leftarrow 0$ to $m - 1$ **do**
        UpdateDriver$(i, R[pos].tim)$ ▷ locate all workers' current positions on the graph
    end for
    $CandiateCars \leftarrow$ SingleSearch$(R[pos].s, R[pos].ddl - R[pos].tim)$ ▷ Do a simple radius search to find potential workers for insertion of $R[pos]$
    $bestCar \leftarrow NULL$; $BestDetour \leftarrow \infty$
    for each car c **in** candidateCars **do**
        $(route,\ detour) \leftarrow$ InsertionOperator$(c,\ R[pos])$
        if $detour <$ bestDetour **then**
            bestCar $\leftarrow c$
            bestDetour $\leftarrow detour$
        end if
    end for
    Insert$\big(bestCar,\ route\big)$
        ▷ Assign request $R[pos]$ to the best worker with minimal detour and update route
end for



(a) Thread 1 serves r_1 by inserting (o_1, d_1) in w_1's empty route



(b) Thread 2 serves r_2 by inserting (o_2, d_2) in w_1's empty route



(c) Thread 1 overwrites w_1's route consisting of r_2 with r_1's route.

Figure 4.2: Threads 1 and 2 simultaneously trying to serve r_1 and r_2 by inserting both requests into w_1's route. The order in which they are inserted depends on which thread finishes first.

For example, in Figure 4.2, we have one worker $W = \{w_1\}$ and a set of requests $R = \{r_1, r_2\}$. Assume both requests can feasibly be served by w_1. The insertion operator working in two separate threads for serving r_1 and r_2 respectively, determines that w_1 is the best worker to serve both requests. It could then lead to the following issue:

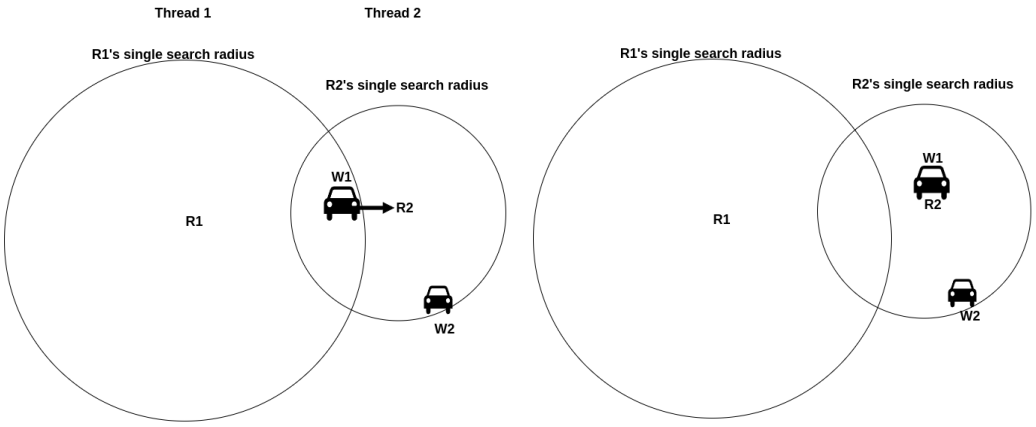
Thread 1 and thread 2 both try to insert in the empty path of the worker, at the very first position. Thread 2 finishes first and assigns $w_1 = (o_2, d_2)$ to

the worker, but thread one had already determined that r_1's locations will be inserted at the first position. Once thread 1 is done inserting, it **overwrites** r_2's locations that were finalized by thread 2 (Figure 4.2c). Thus, a race condition occurs where r_2's path is overwritten and it appears that r_2 is not served, even though it is feasible for w_1 to serve it.

SingleSearch() race condition

The $assignTaxi(car)$ function tries the sequential insertion operator on a set of potential workers in car and then modifies the best worker's path if a feasible insertion is made. If multiple threads call this function concurrently, they may be working with outdated worker states (positions and current paths of car). One way to try to ensure that two threads do not insert two different requests to the same worker at the same time is by assigning a lock to this function. This way, at least the infeasible insertion situation is averted. Nonetheless, it would still lead to inconsistent paths. $SingleSearch()$ relies on workers' positions being located at the beginning of each iteration to determine if workers are eligible. Concurrent updates (calling UpdateDriver()) might still identify available workers based on partially outdated states. For example, consider the abstract case of Figure 4.3, where we have $R = \{r_1, r_2\}$, $W = \{w_1, w_2\}$. R has been divided into two chunks containing r_1 and r_2 respectively, by parallelizing the for-loop in line 3 in Algorithm 1. Thread 1 is working on serving r_1, and thread 2 is working on serving r_2 concurrently. Both threads have access to the whole worker pool. Here's a potential race condition that might arise:

- Thread 1 runs SingleSearch() for r_1 and identifies worker w_1 as eligible.
- Simultaneously, Thread 2 finishes SingleSearch(), identifies w_1 and w_2 as eligible, and uses the insertion operator to update the state of w_1 because it is closer to r_1 than w_2 (Figure 4.3a).
- w_1 is now unavailable in r_1's search radius. This results in Thread 1 proceeding with incorrect assumptions about w_1's location (Figure 4.3b).



(a) Thread 2 finishes running insertion and assign r_2 to w_1, changing it's location to be out of r_1's search radius

(b) Thread 1 has no eligible workers around it now, and works with the outdated location of w_1.

Figure 4.3: Visualization of a race condition when Thread 1 and Thread 2 are concurrently working on serving $r1$ and $r2$. Thread 2 finishes running before thread 1, leading to thread 1 working with an outdated location of w_1

4.4 Hard Partitioning Requests

Since a naive partitioning approach does not work, we need to restructure the existing problem to enable true mutual independence. Essentially, we need the series of insertion computations to be independent from each other.

One way to ensure mutual independence is by using partitioning strategies to decompose a serial application into multiple concurrently executing parts [HAD09]. Following this strategy we can mitigate the race conditions observed above. Also, the main bottleneck of the sequential design is the large number of shortest arrival time query invocations in the Insertion Operator, which is dependent on the number of requests R, the number of eligible workers per request $CandidateCars$ from Algorithm 1, and the number of insertion possibilities. Mathematically, it could be represented as $|query\_invocations| = n \times |CandidateCars| \times |insertion\_scenarios|$

If we can hard-partition this data (specifically W and R) in an informed way, and run the insertion operator on each partition, it could lead to fewer query invocations due to the reduction of search space. On the other hand, due to the reduced search space we might miss out on optimal routes for inserting requests (since a request in one partition might have an optimal worker in another partition), the reduction in query invocation could lead us to potential speedups. And if the solution quality remains more or less the same, these speedups would

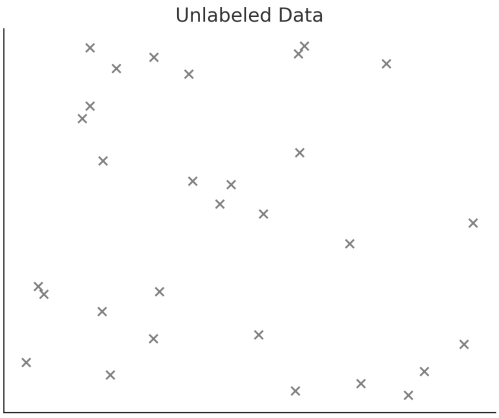
allow scalability for the insertion operator over time-dependent road networks.

An effective approach for parallelizing sequential applications is utilizing **data parallel** design patterns. Under this pattern, a sequential program is decomposed into concurrent units that execute the same instructions on distinct susbets of data. This is the design pattern we choose to follow for partitioning the sets of requests and workers. By dividing R into meaningful, exclusive clusters (or disjoint subsets) and assigning disjoint subsets of W to each of those clusters, we enable multiple instances of the insertion operator to run concurrently, achieving parallel execution. We employ this partitioning design in both the spatial and temporal dimensions of R's attributes and investigate which one fares better for our goals.

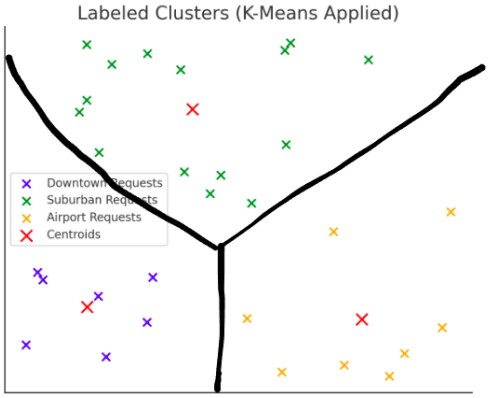
4.4.1 K-means Clustering

Clustering algorithms can be used to expose internal relationships between data in a dataset [AAS21]. K-means clustering is a hard partitioning method (i.e. the partitions are complete and exclusive from each other) that is used to organize data into groups based on their similarities. It meaningfully divides the data into k number of clusters. For example, consider a taxi company that uses K-means clustering to cluster requests based on their distances or pick-up locations (or both) with a k-value of 3. This could lead to clusters such as "Downtown Requests", "Suburban Requests", or "Airport Requests" to form, which would give them insight into which areas should be prioritized by their workforce. Figure 4.4 illustrates how k-means clustering would work on unlabeled data from the example (2-dimensional passenger pickup coordinates on a map) to form meaningful clusters. As you can see the shapes formed are voronoi cells and datapoints are assigned to clusters based on their distance to the cluster's centroids.

There are drawbacks to using k-means such as it yielding high computational complexity for large datasets[AAS21][NKR13]. But if this computation time leads to a further reduction in total simulation time, this trade off is worth it.



(a) Unlabeled pick up locations for passenger requests



(b) Clusters formed based on their pickup locations. The black lines are hard boundaries of the clusters

Figure 4.4: K-means Clustering example on passenger pickup locations for taxi requests

4.4.2 Geospatial K-means Clustering

To determine the best feature to use for partitioning R using k-means clustering, we naively iterate through a couple of R's features and analyze the results w.r.t our goals. Geospatial coordinates are a standard feature used in k-means clustering [SD21] so we investigate those features first.

We start by following the example shown in Figure 4.4 and investigate the spatial arrangement of the requests. Specifically, we look at **pickup and dropoff locations** of the requests. These will henceforth be referred to as Start Node and End Node, and their clustering analysis will be called **Start Node/End Node** clustering. We follow an assumption that given a fixed capacity c_w for a worker, they would be able to pick up multiple passengers or drop-off multiple passengers that are close to each other.

End Node Clustering

We use Python's scikit learn library [Ped+11] to deploy k-means clustering on the set of requests R. We select d_r as the feature that centroids are built upon and run the algorithm on R. The algorithm now splits R into k clusters $C_1, C_2, ..., C_k$ which are disjoint subsets of R:

$$C_1 \cup C_2 \cup ... \cup C_k = R$$

where $C_i \cap C_j = \emptyset$ for all $i \neq j$.

We then equally divide the set of workers |W| into **equal** disjoint subsets and

assign each worker subset to a request subset. After doing this we get completely disjoint request-worker pairs (C, W). We determine the optimal k by iteratively trying out different k-values and seeing what results in yields. This End Node Clustering algorithm is listed in Algorithm 2.

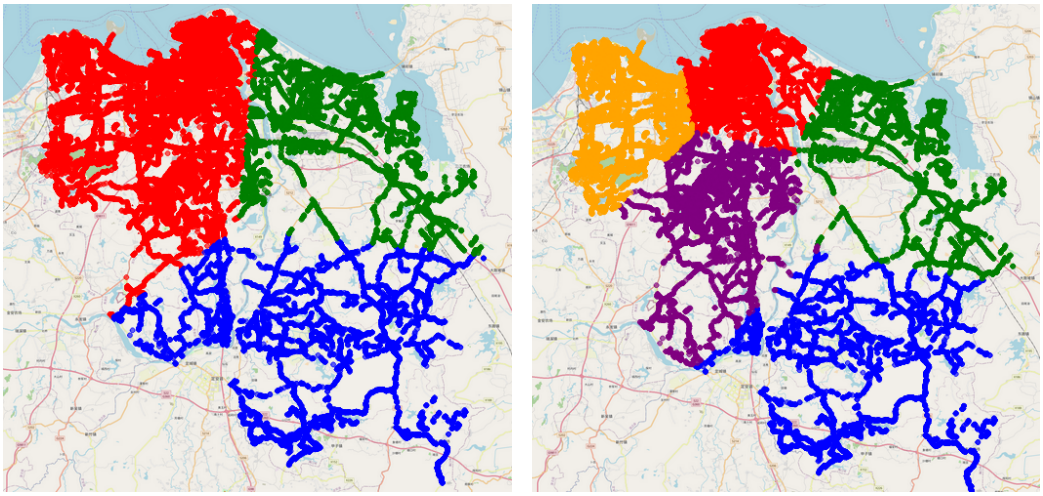
Algorithm 2 End Node Clustering

Input: Request set $R = \{r_1, r_2, ..., r_n\}$, k, worker set $W = \{w_1, w_2, ..., w_m\}$
Output: Cluster-worker subset pairs $(C_1, W_1), (C_2, W_2), ..., (C_k, W_k)$
Initialize k cluster centroids randomly from the $r.d_r$ values
repeat
    for each request $r_i \in R$ **do**
        Assign r_i to the cluster with the closest centroid (based on d_r)
    end for
    for each cluster C_j **do**
        Recompute the centroid as the mean d_r of all requests in C_j
    end for
until centroids do not change
Compute $workersPerCluster = m \div k$ (integer division)
Partition the workers set W into k equal subsets: $W_1, W_2, ..., W_k$
for each cluster C_i **do**
    Assign W_i to C_i
end for
Distribute any remaining workers ($m \mod k$) among the first few clusters
return $(C_1, W_1), (C_2, W_2), ..., (C_k, W_k)$

With this algorithm, we have successfully hard-partitioned the requests. Due to the disjointness of the (C, W) pairs, we can run the insertion operator algorithm on each pair concurrently all the way up to k-cores. In the prior section, W was a shared variable containing the set of workers available to all requests R. Now that W has been split apart and so has R, we will not encounter the data race issues of overwriting the same worker's route and working with outdated worker positions seen in the previous section.

Figure 4.5 shows a visualization of what an End Node clustering looks like for clustering 45,000 real requests over the Haikou City road network. The colored dots represent the dropoff locations of the requests. For k = 3, you can see hard, disjointed partitions based on the spatial proximity of request dropoff locations, as seen in Figure 4.5a. There are 3 subset pairs of R and W that will be formed: $(C_1, W_1), (C_2, W_2), (C_3, W_3)$. We can run the insertion operator concurrently

on each of these pairs. For k = 5, we start seeing more granularity and level of detail in the partitions. Now, the end nodes within the partition are closer to each other. For this example, 5 cluster-and-workers subsets will be formed: $(C_1, W_1), (C_2, W_2), (C_3, W_3), (C_4, W_4), (C_5, W_5)$.



(a) End Node clustering, k = 3. The colored dots represent End nodes.

(b) End Node Clustering, k = 5. The colored dots represent End nodes.

Figure 4.5: Visualization of the End Nodes of requests using k-means clustering over 45,000 Haikou City requests with EndNode coordinates as the clustering feature for different k values. The different colors represent unique clusters

Chapter 5

Informed Partitioning Method

In this chapter, I build upon **Algorithm 2** that partitions R into k clusters and equally distributes subsets of workers W to each cluster. The final result is a set of k request-cluster and worker pairs: $\{(C_1, W_1), (C_2, W_2), ..., (C_k, W_k)\}$. We then run the insertion operator on these pairs concurrently. The minimum number of concurrent jobs running is 2, to ensure the utilization of multiple cores. The max number of concurrent jobs is data dependent due to memory bottlenecks caused by the TD-G-Tree (discussed in Chapter 7). Since we effectively reduce the search space and invoke fewer query(), we should witness a significant speedup in our clustered approach compared to the sequential.

We are unsure if the spatial feature partitioning in Algorithm 2 is the best feature for maximizing the served rate. We do not get to know any more information about the requests within the clusters beyond the fact that their dropoff locations are close to each other. Two requests could have a close drop-off location to each other but might have pickup locations far away from each other. Only the workers within this cluster are allowed to interact with these requests. This might lead to larger routes for these workers which may lead to a decreased served rate. To mitigate this, and to study different clustering criteria, this chapter analyzes other features of R that we can use to get this served rate up. It also looks at opportunistically allocating workers to clusters under some assumptions.

5.1 Spatiotemporal K-means Clustering

A spatial feature that is not originally a part of R's attributes is the shortest path time between the start node and end node. Shortest path distances for ridesharing trips have been used as clustering features in prior research [Zuo+21] to maximize served rate while minimizing travel time. Additionally, platforms like Google Maps use Contraction Hierarchies to precompute some important

distances before answering their queries [Got+19]. Thus, it makes sense to have a preprocessing step that computes the shortest paths (specifically the travel time of the shortest path) of the requests. We take into account this pre-processing time in our experiments and show that it benefits our goals.

The nature of a time-dependent road network also makes shortest path travel times **temporal in nature**, since the SP travel time between two nodes fluctuate throughout the day. Thus, we calculate the time-dependent shortest path (tdsp) using query (u,v,t) for all requests as a pre-processing step.

After calculating all the tdsps for a given R (which carries negligible computational costs when compared to the overall execution time), we feed it to the k-means clustering algorithm as a single feature. The idea is that by clustering with this feature, we will receive more information on the request's nature than simple end node/start node clustering. Using this information, we can then assign the workforce to the cluster set in an effective manner.

For example, consider a set of taxis are near downtown Victoria, B.C., on a Friday night and start getting dispatch requests from downtown. The drivers would ideally like to take on short rides for drop-off at closer neighbourhoods around downtown. This way, they could potentially come back downtown and take more short-trip passengers, leading to higher revenue earned for the night. They would not prefer picking someone whom they are dropping them off at Sidney, B.C. (which is a 30+ minute drive from downtown) because they would lose the chance to serve multiple requests.

Proposition: Given a set of request clusters C and set of Workers W, if requests in a request cluster have smaller time-dependent shortest path travel times on average, they have a higher probability of getting served. Thus, a higher percentage of workers (aka a bigger subset of W) should be allocated to these clusters. This allocation should follow the constraints listed in the problem definition.

Based on this proposition, it makes even more sense to use **tdsp** as our primary cluster feature over End Node/Start Node because:

1. TDSP clusters will contain requests of similar sizes. Thus, the average tdsp will be closer to the median of the cluster, rather than the mean, which gives us informative categorization: short distance requests, medium distance requests, long-distance requests. A higher k-value would lead to higher precision of categorization in the cluster set.

2. Workers can be allocated based on our proposition of serving more requests. We inverse proportionally assign the workers based on the average tdsp of

the cluster. The formula for calculating average tdsps A_i is given by:

$$A_i = \frac{1}{|C_i|} \sum_{j=1}^{|C_i|} tdsp_j$$

where $|C_i|$ is the total number of requests in cluster C_i, and $tdsp_j$ is the time-dependent shortest path associated with the j^{th} request within C_i.

3. We are building upon the insertion operators' objective function and constraints. Its objective is to minimize the travel time for workers while serving as many feasible requests as possible. Using tdsp clustering leans into this even more since we are assigning more workers to clusters that are considered "small trips", which have low travel times.

The **opportunistic worker allocation formula is given as follows:** Given a total number of available workers m, and k clusters $C_1, C_2, ..., C_k$, the number of Workers W_i allocated to each cluster C_i is calculated by:

$$W_i = \left\lfloor \frac{\frac{1}{A_i}}{\sum_{j=1}^{k} \frac{1}{A_j}} \times m \right\rfloor$$

where,

- W_i is the number of workers assigned to cluster C_i
- A_i is the average tdsp per request within cluster C_i
- $\lfloor x \rfloor$, the floor function ensures the number of workers assigned is an integer.

TDSP Clustering

Similar to Algorithm 2, we use Python's scikit learn library to apply our k-means clustering on a set of requests R. The algorithm contains a pre-processing step and clustering implementation:

1. We calculate $tdsp$ for all the requests in R using $query(u, v, t)$ as a pre-processing step and embed them into R's tuple. For example, the i^{th} request is now denoted by $r_i = \langle o_{r_i}, d_{r_i}, t_{r_i}, e_{r_i}, c_{r_i}, tdsp_{r_i} \rangle$.

2. We select $tdsp_r$ as the feature for k-means clustering that is run on R. This splits the R into k clusters $C_1, C_2, .., C_k$ which are disjoint subsets of R.

Algorithm 3 TDSP Clustering and Opportunistic Worker Allocation

Input: Request set $R = \{r_1, r_2, ..., r_n\}$, number of clusters k, worker set $W = \{w_1, w_2, ..., w_m\}$ where $m = |W|$
Output: A set of cluster-worker subset pairs $(C_1, W_1), (C_2, W_2), ..., (C_k, W_k)$
Initialize k cluster centroids randomly based on shortest-path distances ($tdsp$)
repeat
    for each request $r_i \in R$ **do**
        Assign r_i to the closest centroid based on $tdsp$
    end for
    for each cluster C_j **do**
        Recompute the centroid as the mean $tdsp$ of all requests in C_j
    end for
until centroids do not change or max iterations reached
for each cluster C_i **do**
    Compute total $tdsp$ sum S_i for C_i:
    Compute average $tdsp$ A_i per request for C_i:
end for
for each cluster C_i **do**
    Compute worker allocation W_i:
    Adjust rounding to ensure exactly m workers are assigned
end for
return $(C_1, W_1), (C_2, W_2), ..., (C_k, W_k)$

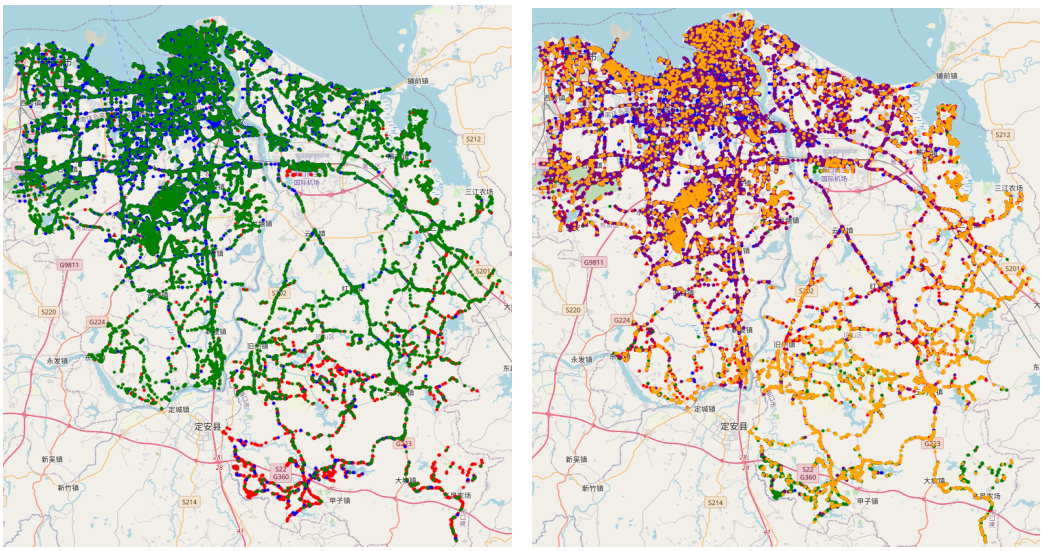
3. We then split the workers $W = \{w_1, w_2, ..., w_m\}$ among the clusters. We allocated them inverse proportional to the average tdsp per cluster given by the formulas above.

4. We then return the disjointed request-cluster and worker pairs (C, W).

Detailed algorithm is given in **Algorithm 3**.

We run the insertion operator on the output of this algorithm, i.e., (C, W) pairs, concurrently, with a dynamic schedule. For example, if there are more (C,W) pairs than cores, we run the insertion operator on k (C,W) pairs concurrently with the help of multiprocessing using GNU parallel. Whenever a pair is done running and a core is freed, we instantly feed another (C,W) job to it. This way, all available cores are always running jobs in parallel.

Figure 5.1 shows a visualization of how the requests are distributed into clusters based on tdsps. The locations on the maps (colored dots) are pickup locations for the requests (used for simplicity and comparison purposes).The shapes are in stark contrast with figure 4.5 because now the requests are not hard bound to spatial zones (the clusters are non-uniform, more than spatial-based). They are multi-layered, while still being disjointed. Thus, they offer some flexibility for dynamic ridesharing and workforce allocation. The workers are not restricted to spatial zones either and can be allocated more effectively depending on which category of trips a ridesharing or food delivery service wants them to do.



(a) TDSP clustering, k = 3 (b) TDSP Clustering, k = 5

Figure 5.1: Visualization of k-means clustering over 45,000 Haikou City requests with *tdsp* parameter as the clustering feature for different k values. The different colors represent unique clusters

Chapter 6

Empirical validation

6.1 Experiment Goals

Our goals are to demonstrate the scalability and customizability of our parallel approach by:

- Maintaining or exceeding the percentage of requests served in a simulation achieved by the sequential insertion operator.
- Consistently witnessing a lowered total running time of a RPSM simulation compared to the sequential.
- Presenting multiple potential trade-off options between requests served and execution time for our clustered approach.

We thoroughly test our parallel approach on 3 different datasets from different RPSM applications, and we use various ride-sharing scenarios, as described by [GZC24b] as input parameters, and investigate trends that help identify bottlenecks in the insertion operator.

6.2 Experiment Setup

6.2.1 Experimentation Software

For the sake of reproducibility, I will explain the software used to parallelize the code, and the specifications of the computer(s) that I ran the insertion operator experiments on. The sequential operator code is made publically available by [GZC24b] on GitHub. It can be accessed by visiting: https://github.com/gzyhkust/Insertion-Operator
The code is implemented in C++ 11 and it requires the usage of the METIS

library found in: https://github.com/KarypisLab/METIS. A a contiguous graph partitioning algorithm from this library is used to build the TD-G-Tree.

[GZC24b] provide us with real-world datasets for workers, requests, and road networks taken from [Zen25]. We run the tdsp() algorithm from TD-G-Tree to get the time-dependent shortest arrival times for every request in the dataset, as seen in Chapter 5. With these request and worker datasets as input, we use k-means clustering on Python with the features mentioned above, and vary K. After generating k disjoint (C, W) pairs, we run every pair concurrently on a separate core using GNU Parallel [Tan11].

GNU Parallel

GNU Parallel is a shell tool that executes jobs concurrently using one or more CPU cores. In our case, a job is the insertion operator executable working with a cluster-worker pair (C_i, W_i). We run up to k-jobs concurrently, dependent on the number of processors and RAM that our hardware has. We always try to run as many concurrent jobs as possible. For example, the command line prompt for running maximum concurrent jobs, for 20 (C, W) pairs, is given by:

time parallel -j 0 "./n requests_cluster_.txt LogisticWorkers_.txt > requests_.log" ::: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Here, *n* is the insertion operator executable. *parallel* invokes GNU parallel, **j** refers to jobs to be run in parallel and takes an input *0*, which denotes the maximum number (up to the max number of cores in the computer) of jobs are to be ran in parallel. The numbers $0, 1, ..., 20$ denote the cluster-worker subscripts, ensuring that each (C_i, W_i) runs the executable.

The $time$ command is used to calculate the total running time of the entire script. Thus, it will give the total running time of finishing all the cluster-worker pairs. Since the pre-processing steps do not incur much cost when compared to the execution times, only the total time to execute the insertion operator is collected and compared for both the sequential and the clustered approaches.

The three major algorithms seen in the experiment results are:

- End Node Clustering with equal worker distribution
- TDSP Clustering with opportunistic worker allocations
- Sequential Approach

Implementation Specifications

The experiments for the Haikou and Cainiao datasets are conducted on my personal computer with sixteen 12th Gen Intel(R) Core(TM) i7-12650H processors

with hyperthreading enabled and 16 GB of memory. The experiments for the Chengdu datasets are conducted on Compute Canada's Cedar server, on a compute node with sixteen Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz processors with hyperthreading enabled and 150gb of memory.

6.2.2 Datasets

This section covers the datasets that the proposed methods are evaluated on, which stem from different applications. All of them are Road Networks based in different cities of China. The applications that we focus on are **Ridesharing** and **Logistics**.

Logistics Dataset

The Logistics dataset that I work on is taken from [GZC24b] and contains logistics data of last-mile deliveries of parcels in Shanghai collected by the Cainiao Logistics Platform. It consists of parcel origins, parcel destinations, and deadlines, and is preprocessed by downloading the road network of Shanghai City, China. This road network is taken from [Zen25] and the edge weights are represented as units of travel time between vertices. Figure 6.1 is a visualization of this road network, where the blue dots represent nodes and the red line segments represent edges connecting certain nodes. We work on a subnetwork of this full Shanghai Road Network called Cainiao, in accordance to [GZC24b]'s baseline.

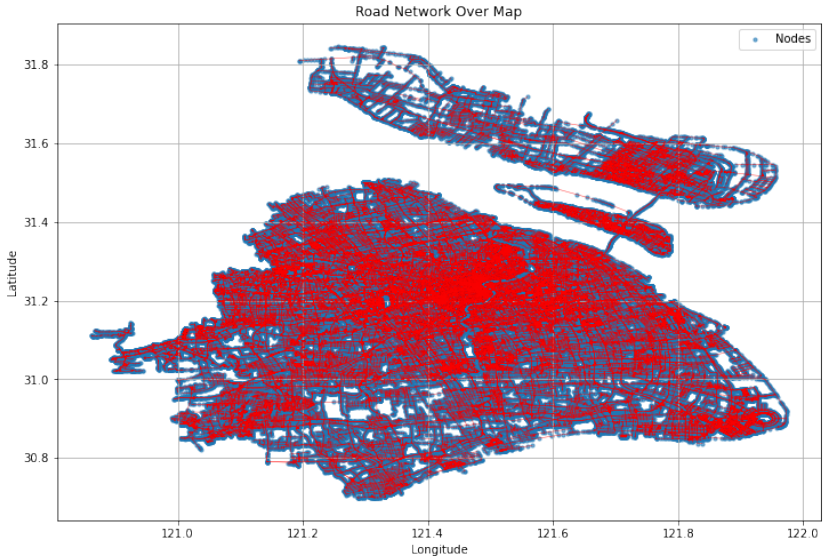


Figure 6.1: Road Network of Shanghai

The dataset also provides original deadline times, which is an essential parameter in our experiments. This dataset and a skeletal road network of Shanghai (less vertices and edges) are used in [GZC24b] for their sequential, linear-time insertion operator performance evaluations. We use their sequential operator as a baseline, and use our clustered algorithm on the same road network to demonstrate how it outperforms theirs, in terms of total running time. This dataset's importance stems from its original deadline times which enhances the Logistics simulation's closeness to the real-world. Thus, we use it extensively, with various additional parameters to test our clustered approach with scrutiny.

Ridesharing Datasets

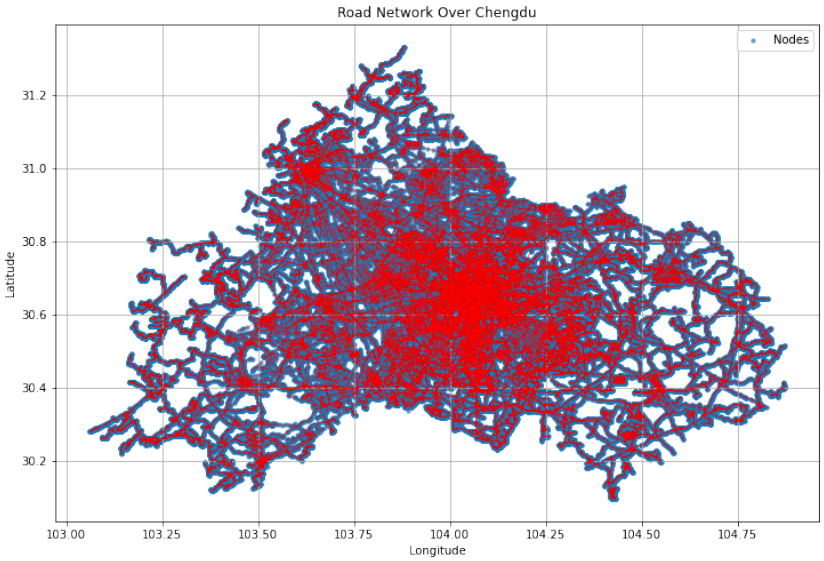


Figure 6.2: Road Network of Chengdu

We use two public **ridesharing datasets** from the cities of Chengdu (our biggest road network with over 400,000 vertices, and 900,000 edges) and Haikou City, China to evaluate our algorithm's performance, visualized in figures 6.2 and 6.3 respectively. These datasets consist of over 200,000 taxi requests during daytime hours (8:00 to 18:00). The origin and destination of these requests are mapped to the nearest node in the road network for these cities, taken from [Zen25]. Once again, the edge weights are travel time units. Since this is a popular service, it is important to see how our algorithm performance against the baseline sequential algorithm.

Table 6.1 consists of statistical characteristics of the datasets mentioned above, including the number of vertices, edges, interpolation points, and average

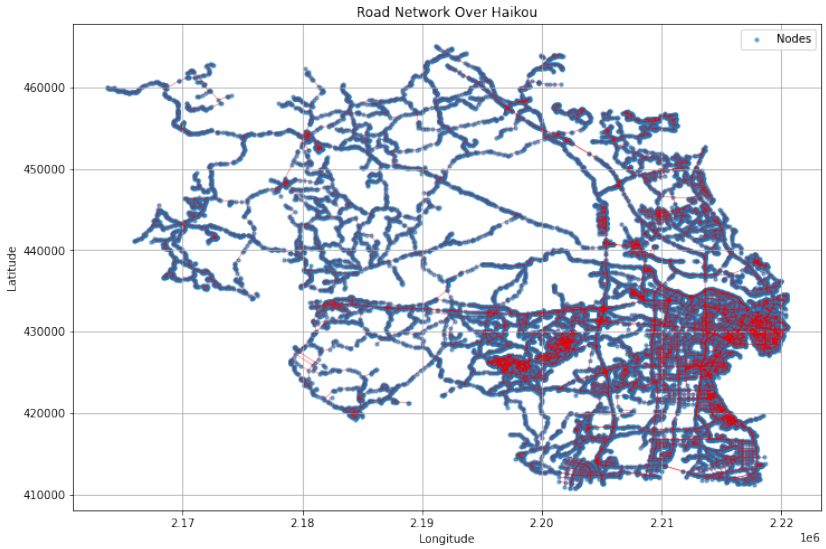


Figure 6.3: Road Network of Haikou City

interpolation points per edge. The existence of more than one interpolation point for an edge weight turns a static road network into a time-dependent one. Due to the enormous construction size and higher querying time of Chengdu, the average number of interpolation points is 2. For Haikou, a smaller dataset, the average interpolation points is 3. For Cainiao, the smallest dataset, we have 22 interpolation points on average per edge, suggesting that the time-dependent aspect of the road network is more prevalent in Cainiao, with edge weights and travel times changing around 22 times a day, on average. These interpolation points are directly taken from [GZC24b].

Table 6.1: Statistics of datasets.

Category	Dataset	#Vertices	#Edges	#Interpolation points	# (Avg. Interpolation pts per edge)
Ridesharing	Chengdu	423,434	913,718	1,827,436	2
	Haikou	41,542	89,206	267,618	3
Logistics	Cainiao	9,936	23,872	525,184	22

6.2.3 Sequential Parameters

In this section, I describe the parameters I vary, formulate a table on how we vary them, and also describe the metrics that evaluate the performance of our partitioned approach against the baseline. The table 6.2 summarizes the values for all of the parameters described above and states which ones we will keep constant in bold while looking for specific impacts. This experiment suite is identical to Sequential's experiment suite.

Parameter Settings	
Parameters	**Settings**
# of workers $\|W\|$	Ridesharing (Haikou): 100, **200**, 300, 400, 500 Ridesharing (Chengdu): 25, **50**, 75, 100, 125 Logistics: 200, **400**, 600, 800, 1000
# of Requests $\|R\|$	Ridesharing (Haikou): 20K, **40K**, 60K, 80K, 90K Ridesharing (Chengdu): 5K, **10K**, 15K, 20K, 25K Logistics: 4h, **6h**, 8h, 10h, 12h
Delivery deadline (minutes)	Ridesharing: 10, 15, **20**, 25, 30 ($+ t_r + \text{tdsp}(o_r, d_r)$) Logistics: **Original deadline information (($+ t_r +$ tdsp(o_r, d_r)))**
Capacity c_w	Ridesharing: 3, **5**, 10, 15, 20 Logistics: 80, **100**, 120, 140, 160

Table 6.2: Parameter Settings

Number of Workers |W|

[GZC24b] tests the scalability of their algorithm by varing the number of workers. Although their experiments show exceptional speeds for average insertion and response time per request, we evaluate how their total running time compares to our approach against this parameter. Our goal is to evaluate how varying |W| affects the sequential and parallel approaches in terms of served rate and simulation time and observe if our parallel approach is scalable.

Number of Requests |R|

We investigate how the requests served rate and execution time vary based on the number of available requests. For Cainiao, we possess real parcel delivery data with real deadlines from 8:00 am to 8:00 pm. Therefore, for Cainiao, we vary the requests based on the **Request Release Duration**. The requests are released at different times of the day. We vary the time of day by varying the delivery hours, ranging from 4 to 12 hours. For example, 4 hours indicates the number of requests between the intervals [8:00 am - 12:00 pm], 6 hours indicates the number of requests between [8:00 am - 2:00 pm], etc.

For the ridesharing datasets, we vary the number of requests in accordance to [GZC23].

Delivery Deadline: $e_r - t_r$

We investigate if our algorithm can serve more requests given a higher deadline time for Ridesharing datasets. We use original deadline times for Logistics and those remain fixed. The deadline calculation is taken from [Ton+22]

Capacity of the worker c_w

We vary the capacity of the worker and evaluate if having a higher capacity can serve lead to more requests served in both algorithms and see which one fairs better. We also investigate if increasing the capacity leads to more requests served when we order by pickup node hotspots.

6.2.4 K-means Clustering Parameters

These parameters are specific to our clustered approach. When varying a 'cluster' parameter, we keep sequential parameters fixed(unless specified).

Number of Clusters

Finding an optimal amount of clusters that **serves our goals** is necessary because we do not want to miss out on requests served or a faster simulation. We try to find the **'sweet spot'** K number through extensive experimentation, and looking for the elbow in the graph. This sweet spot refers to assigning 50/50 importance to both served rate and execution time. We observe graph trends such as the number of requests served plateauing or the decrease in execution time becoming moderate/low instead of steep.

Table 6.3 summarizes the various cluster parameter settings.

Cluster Parameter Settings	
Parameters	**Settings**
# of Clusters K	1,3, 5, 8, 10, 20, 25

Table 6.3: Cluster Parameter Settings

6.3 Experiment Results

In this section, I will present the results of my experiments that I conducted by assessing the better clustering criteria and varying the number of clusters to determine the optimal criteria and value for both. This is followed by varying

the parameters declared above with the optimal clustering method and insights are derived from the results. I evaluate at the Requests Served Rate and Execution Time for the two algorithms: Sequential Insertion Operator and Clustered Insertion Operator, and demonstrate how our clustered approach fares better for scalability while maintaining solution quality.

6.3.1 Determining the better Clustering Criteria:

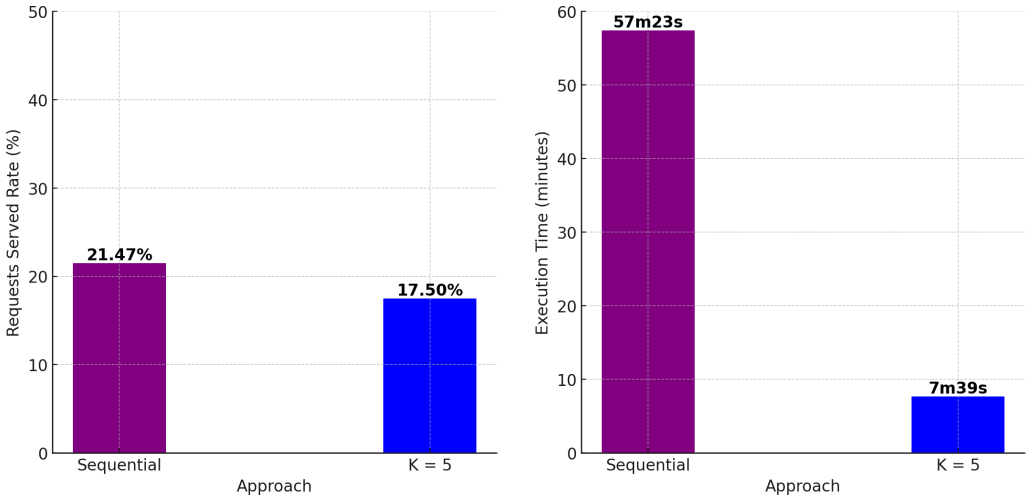
End Node Clustering and Even Distribution of Workers

As mentioned before, the motivation behind clustering the data is reducing the search space so that the bottleneck $query()$, will be invoked less due to less number of eligible workers and total number of requests.

This hypothesis is supported by the results given below in Figure 6.4. We call the act of running the Insertion operator once over the complete sets of W and R as the **sequential approach** and we call running the insertion operator concurrently on k (C, W) pairs as the **clustered/parallel approach**. Here are the key observations we make from the results of these two approaches:

Pros:

Figure 6.4b supports our hypothesis. The total running time of the simulation speeds up by **8.14X**. This is a significant decrease in the execution time. This demonstrates the **speedup potential that k-means partitioning and data-level parallelism bring to the insertion operator**. Because of a reduced number of requests and workers within the (C, W) pairs, there are fewer query invocations than the sequential, which leads to the clustered approach finishing faster.



(a) Served Rate: Sequential vs End Node Clustering (lower is better)

(b) Execution Time: Sequential vs. End Node Clustering (higher is better)

Figure 6.4: Comparison between the sequential Insertion Operator ran once on a single Requests set vs. the sequential Insertion Operator ran concurrently on 3 (C, W) pairs

Cons:

As seen in Figure 6.4a, the requests served rate decreases compared to the sequential. Potential reasons this could be the case are as follows:

1. We cluster based on the End Node but we do not take into account any Start Node information which leads to lesser-informed clustering. For example, if two passengers are going to the airport but their starting locations are far away from each other, a worker might end up going extra distances/carry unnecessary loads to serve both, or not serve them at all. This leads to a reduction of feasible insertions in a worker's path. A worker from another cluster that might be closer to one of those airport requests cannot pickup this passenger due to strict zone constraints.

2. Due to equal distribution of m workers among k clusters, the worker-to-request ratio becomes disproportionate. For example, the worker-to-request cluster distribution is given in table 6.4. For request clusters C_4 and C_5, there are significantly less workers available for the number of requests that they contain. This would lead to fewer requests served due to fewer potential workers in those clusters.

Requests-per-cluster	Worker	Ratio
5732	40	0.0069
7595	40	0.0052
4233	40	0.0094
12386	40	0.0032
10054	40	0.0039

Table 6.4: Worker to number of requests per cluster ratio

3. The clustering does not take into account the **distances of the requests themselves.** For example, on the surface it might look like C_3 has the potential to serve the most requests due to its higher workers-per-request ratio, but we do not know how long those requests might take. Their distances are unknown to us within End Node clustering, which leads to uninformed distribution of workers and ultimately, missed opportunities to serve potential requests that the sequential approach serves.

To summarize, we are able to support our hypothesis, which is that a reduction in the sizes of |W| and |R| leads to a faster execution time. We observe that the requests served rate decreases by 4% (with a potential to decrease even more with increasing k values) due to the relative lack of informed partitioning of both requests and workers. This motivates us to expand upon the End Node clustering algorithm, focusing on opportunistic worker allocation and better clustering criteria instead. We assume that we will run into the same issues that we experienced with End Node Clustering in Start Node Clustering as well. Chapter 6 dives into this more informed, opportunistic k-means partitioning and worker allocation algorithm.

TDSP Clustering and Adaptive Worker Allocation

Clusters C_i	Requests	Total tdsp S_i	Average tdsp A_i	Workers W_i
C0	7807	22911020.08	2934.68	63
C1	19640	15594932.26	794.04	231
C2	17553	30538697.07	1739.80	106

Table 6.5: Opportunistic Worker allocations for TDSP clustering, k = 3

As an example, the opportunistic worker allocations for $|W| = 400$ workers amongst 3 and 5 request clusters from Figure 5.1 are given in tables 6.5 and 6.6 respectively. The details are listen in the table for k = 3 and k = 5. Here are some key observations made from this:

Clusters C_i	**Requests**	**Total tdsp S_i**	**Average tdsp A_i**	**Workers W_i**
C0	10872	19816649.16	1822.72	58
C1	10979	6302975.05556	574.09	183
C2	2725	9542359.88	3501.78	30
C3	13654	16208660.24	1187.10	88
C4	6770	17174005.08	2536.78	41

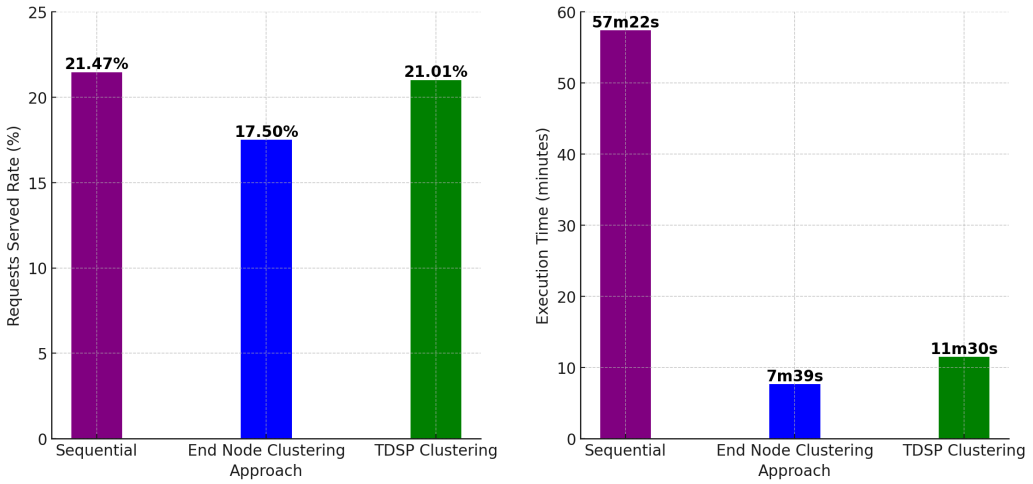
Table 6.6: Opportunistic Worker allocations for TDSP clustering, k = 5

- **We have more information about the requests and can categorize them based on their average tdsps**. In table 6.5, we can categorize C_0 as **"long distance requests"**, C_1 as **"short distance requests"**, and C_3 as **"medium distance"** requests.
- A higher percentage of workers are allocated to the short distance requests C_1, in line with our proposition and Algorithm 3. This ensures more requests in this cluster will get served and will bring up the overall served rate. **Short distance requests are a priority, according to our proposition.**
- For k = 5, we now have the opportunity to allocate the majority of workers to C_1 and C_3, aka the "short distance" requests. This potentially could heighten the served rate as well.
- **Worker allocation balances between efficiency and fairness.** While our algorithm prioritizes short-distance requests (C_1 and C_3) in table 6.6 to maximize overall served rate, long-distance requests (C_0, C_4) are still allocated workers too. This ensures that while shorter trips receive priority, longer trips are not entirely neglected, providing a fair access of taxis to all types of requests.

Superiority to End Node Clustering

Our **proposition is confirmed** in Figure 6.5a, where we indeed serve more requests than End Node clustering, even with a smaller workforce per request cluster. It ends up matching the sequential's request serve with an unnoticeable decrease (21.4% vs. 21.01%). And since we have lesser query(u,v,t) invocations, we also see a similar amount of decrease in execution time for the TDSP clustering. We witness a 5X speedup! Our partitioning + multiprocessing approach is effectively matching the requests served rate for a significant decrease in execution time. It is also able to do with a much smaller search space for (C, W) pairs.

Therefore, this extended partitioning method is superior to End Node clustering in terms of informedness, detailed categorization, and achieving the goals outlined in this thesis. This **TDSP clustering now becomes the primary parallel approach with which we compare against the baseline (sequential approach).**



(a) Served Rate: Sequential vs End Node Clustering vs. TDSP Clustering

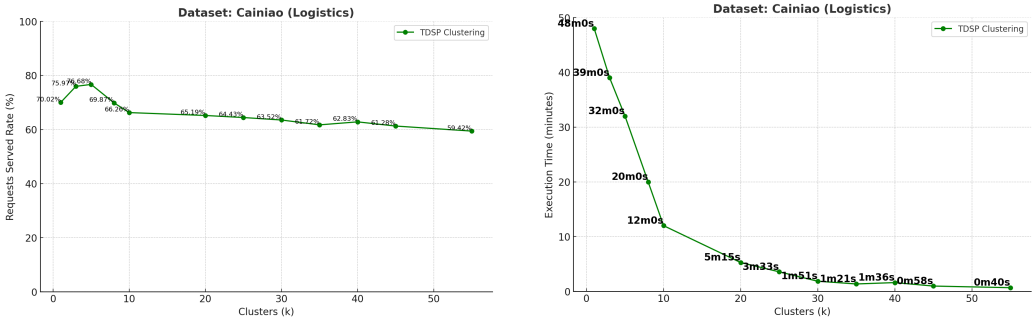
(b) Execution Time: Sequential vs. TDSP clustering

Figure 6.5: Comparison between the sequential Insertion Operator ran once on a single Requests set vs. the sequential Insertion Operator ran concurrently on 3 (C, W) pairs. K = 5

6.3.2 Selecting a K Value

As mentioned, it is important to vary the K value and choose a K before running our insertion operator simulations on varying parameters. We decided to prioritize both the served rate and decreasing the simulation time equally. This is the 'optimal' K value, or 'sweet spot'.

We keep all sequential parameters fixed while varying K across the 3 datasets. Remember, K = 1 here implies the sequential approach. K > 1 implies the parallel approach. The results are as follows:



(a) Served rate vs. K

(b) Execution time vs. K

Figure 6.6: Comparison between sequential (k=1) and TDSP Clustered (k $>$ 1) approaches for Served Rate and Execution Time for **Cainiao**. Workers are inverse proportionally divided by average TDSP

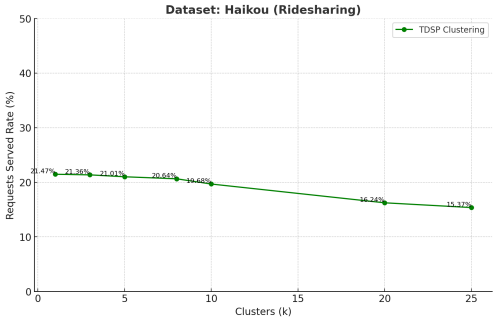
For Cainiao's **fixed parameters**, we see that 70.02% of requests get served for the sequential approach in 48 minutes [Figure 6.6]. The served rate is relatively high but we need to keep in mind that we are only using 10% of the |W| and |R| values. If we were to scale those parameters to larger numbers (|W| = 4000, |R| = 120,000), we would witness a severe increase in execution time. It would take at least half a day to finish a simulation with those numbers. Perhaps a company would prefer if they could rapidly perform simulations to explore different scenarios and dictate how to maximize requests through those varied simulations and their current workforce numbers.

Using our k-means clustering method with opportunistic worker allocations (k $>$ 1), we find out that for some K values, we end up **achieving a higher served rate (Figure 6.6a) compared to the sequential for a lower execution time (Figure 6.6b)**. For example, for **K = 3,5** we serve **5.95%** and **6.66%** more requests served than the sequential. We also see speedups of 1.2X to 1.5X for those values respectively, which is marginal but still an improvement.

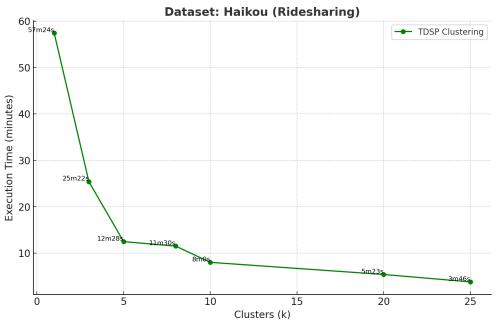
In both the served rate and execution time figures, the elbow is seen between **K = 10, 20**. From observing this, we claim that **K = 20** is the optimal k-value for Cainiao. For a minor **4.83%** decrease in served rate, we see a speedup of **9.6X** (k = 20). There is less than 1% decrease in served rate between K = 10 and K = 20, while the speedups for k = 20 being way higher.

For the sake of extreme analysis, we vary K all the way to 55 and witness a **72X** speedup for a maximum of 10.6% decrease in serve rate.

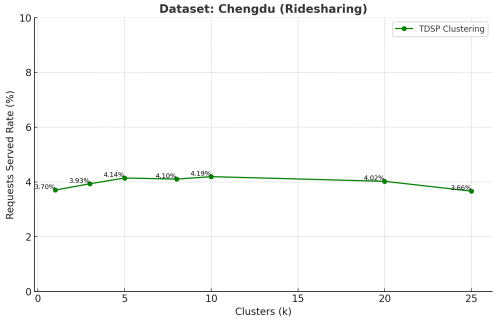
To determine the optimal k-value for **Haikou City**, we fix all the sequential Haikou parameters and test our TDSP clustering + Worker Allocation (parallel approach) via increasing k. In Figure 6.7a, we witness a decrease of 0.62% **to** 6.1% in the served rate. We see a speed up of **1.57X to 15X**, as seen in Figure



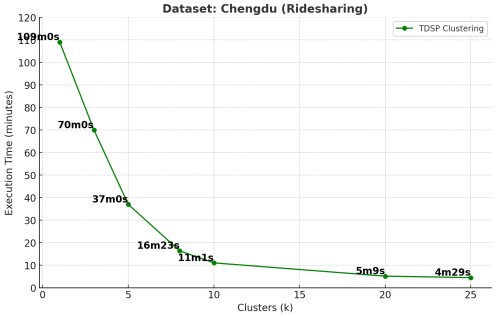
(a) Served Rate(%) vs. Increasing K



(b) Execution Time(mins) vs. Increasing K



(c) Served Rate (%) vs. Increasing K



(d) Execution Time(mins) vs. Increasing K

Figure 6.7: Comparison between sequential (k=1) and TDSP Clustered (k > 1) approaches for Served Rate and Execution Time for **Haikou (top row) and Chengdu (bottom row)**. Workers are inverse proportionally divided by average TDSP

6.7b. The elbow of the curve for the execution time seems to be K = 5 or K = 8. The served rate elbow is less evident for Haikou. Thus, we select the optimal K value for an equal trade-off between Served Rate and Execution Time to be **K = 8**. For that value, we lose only 0.83% of the served rate. For that loss, we gain a speedup of **4.6X to 5X**.

For **Chengdu**, we see a consistent **increase in served rate from K = 3 to K = 20**, as seen in Figure 6.7c, which demonstrates the potential of our approach for maximizing requests and the verification of our proposition for prioritizing lower tdsp clusters. We see speedups of 1.5X to 24X for varying K (Figure 6.7d). As per the elbow, we see that K = 8 or K = 10 could once again be optimal. We decide to go with **K = 10** to show some variety in our optimal K-values for each dataset.

Summary

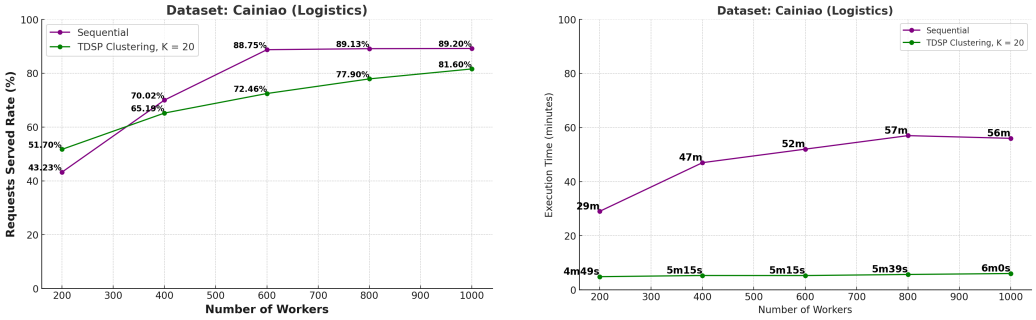
In summary, we determine the optimal k value to be anywhere between **K = 8 to K = 20** across 3 datasets when it comes for 50/50 importance. There is only a negligible decrease in served rate while we garner immense speedups! Also, the served rate inevitably gets lower for large number of K values (Figure 6.6a) so we keep K <= 20. We use the best k-values for each respective dataset in table 6.7 when comparing the baseline to our parallel approach against the experiment parameters.

Table 6.7: Optimal K-value of datasets

Category	Dataset	Optimal K-value
Ridesharing	Chengdu	10
	Haikou	8
Logistics	Cainiao	20

6.3.3 Varying the number of workers |W|

Logistics



(a) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 20)

(b) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 20)

Figure 6.8: [**Logistics**] Impact of Varying |W| on Served Rate and Execution Time and comparing sequential and TDSP clustering approaches.

The impact of varying the number of workers in the sequential algorithm is notable. In figure 6.8 (purple line), for the Logistics Dataset, we see that increasing the number of workers for a fixed number of requests results in an increase in execution time. The served rate increase is steep from 200 to 600 workers but it levels out after that, around 89.20%.

For the clustered approach (green line), here are some key observations:

- For $|W| = 200$, we see an **increase in the served rate** by **8.47%** for a speedup of **5.8X** for the k-means clustered approach. This is a notable improvement from the sequential algorithm and is in agreement with our goals.
- Increasing $|W|$ results in a steep increase in served rate for the sequential up until 600, but then begins to plateau for further $|W|$ increase (Figure 6.8a). While the served rate for the clustered approach keeps increasing linearly with $|W|$.
- The served rate for clustered approach remains lower than the sequential for varying $|W|$ from 400 to 1000 but seems to catch up to the sequential for $|W| = 1000$. There's only a 7.6% deterioration of served rate while we achieve a **9.3X** speed-up.
- The execution time also plateaus with the served rate beyond $|W| = 600$ for the sequential approach. In contrast, **the clustered approach's execution time remains mostly constant through the varying of $|W|$**. Thus, we end up seeing a speedup of 5.8X to 9.3X.

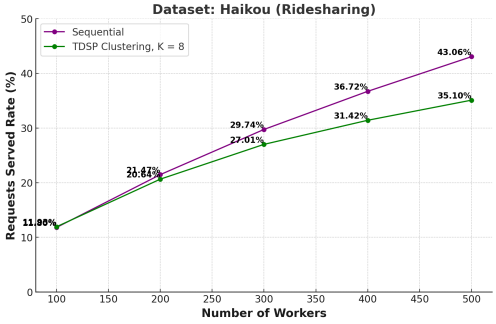
These points show how robust our clustering algorithm is with the scaling of workers.

Ridesharing

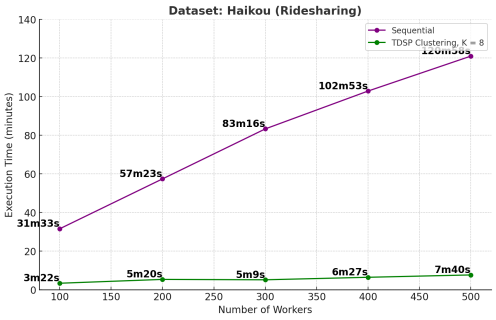
The key observations for the Ridseharing datasets made from Figure 6.9 are as follows:

For **Haikou**, in general, the sequential demonstrates superiority when it comes to the served rate with varying $|W|$, except for when the worker base is small ($|W| = 100$). Even then, for a large worker base, the deterioration of the parallel approach remains less than 8%. For this trade off, we see a speedup of 9X to 15X strengthening the scalability aspect of our parallel approach. The simulation time for our parallel approach appears to remain relatively constant to the linear increase of the sequential, which is a notable drop in time complexity.

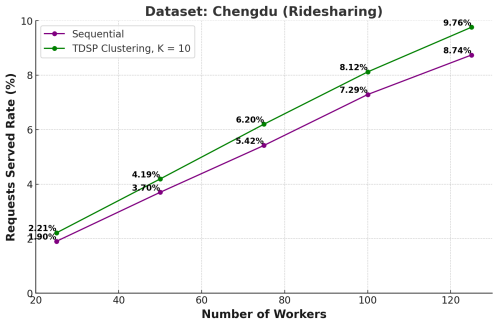
For **Chengdu**, the parallel approach (green line) consistently achieves a higher served rate than the sequential approach (purple line) as the worker base increases. Moreover, the gap in served rates steadily widens in our favor, increasing from $+0.31\%$ to $+1.02\%$, aligning with our served rate objectives. The speedups achieved by the parallel approach are also significant. The time increases linearly both for the sequential and parallel approaches but slope is much higher for the sequential (purple line) than the parallel (green line). The speedup achieved is an average of 8.5X.



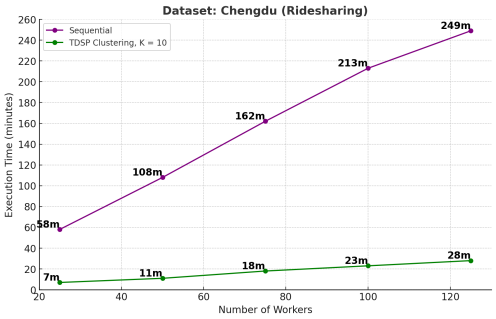
(a) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 8)



(b) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 8)



(c) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 10)



(d) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 10)

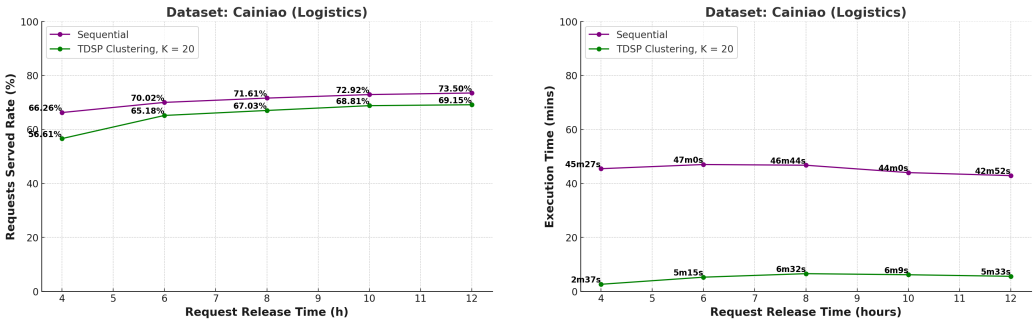
Figure 6.9: [**Ridesharing**] Impact of Varying |W| on Served Rate and Execution Time, comparing sequential and TDSP clustering approaches.

Summary

In summary, our sweet spot K-values for the respective datasets work well in accomplishing our objectives while increasing the worker base. Our parallel approach manages to match (and in some cases exceed) the served rate of the sequential while consistently speeding up simulation times by at least a factor of 5.

6.3.4 Varying the number of requests |R|

Increasing the number of requests results in more requests served due to more potential requests becoming available to the workers, but this might also reduce the overall percentage of requests served. This is because the number of requests served does not proportionally scale with the number of available requests [Figure 6.11]. [Cainiao experiments]



(a) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 20)

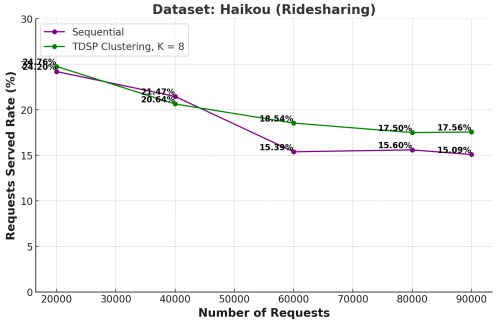
(b) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 20)

Figure 6.10: [**Logistics**] Impact of Varying |**R**| on Served Rate and Execution Time and comparing sequential and TDSP clustering approaches.

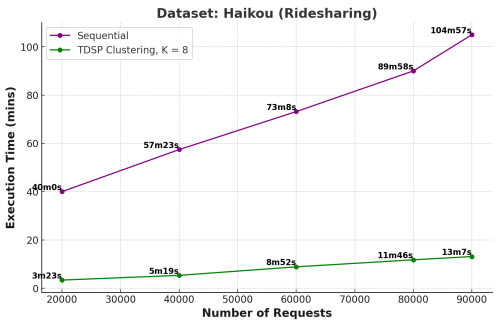
For **Cainiao**, the number of requests is a function of the **Requests Release Time duration** because the logistics orders are only delivered during delivery hours (8:00 am to 8:00 pm). We see a trend of negligible increase in the served rate for both the sequential and parallel approaches (Figure 6.10a). Although the parallel approach loses 10% of the served rate for the morning hours (8:00 am to 12:00 pm) it does catch up to the sequential when all the delivery hour requests are available (8:00 am to 8:00 pm). The average loss of served rate from parallel to sequential is -5.51%, which is a worthy tradeoff for the speedups achieved. We get a high speedup of 15X for daytime hours. Although the speedup factor decreases to 8.6X for the full delivery hours, it is still a pretty significant drop in simulation time. This demonstrates that our parallel approach is scalable for real-world, per-day requests over a dense road network.

For **Haikou**, we see a negligible dip in served rate around 40K requests but we overall witness a higher served rate than the sequential (Figure 6.11a). On average, the parallel approach serves +1.45% more requests than the sequential for varying |R|. We again see consistent speed ups in the range of 8X to 11.4X for varying |R| (Figure 6.11b), while achieving a superior served rate.

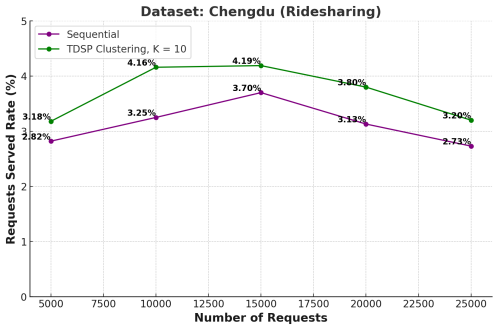
For **Chengdu**, there's an overall higher served rate for our parallel approach as well (Figure 6.11c). We achieve a +0.5% increase in served rate (do remember that the served rate for Chengdu, regardless of approach is very low, the maximum of 4.19% of requests served by approach (green line)). We witness a speedup factor of 7.8X to 10.7X (Figure 6.11d) for varying |R|, demonstrating that our parallel approach is fast and scalable for a large number of available requests, which is exactly what any ridesaring company would want.



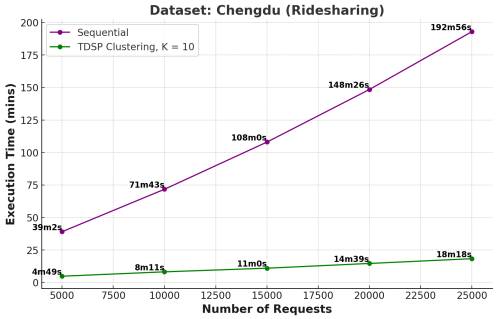
(a) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 8)



(b) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 8)



(c) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 10)



(d) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 10)

Figure 6.11: [**Ridesharing**] Impact of **Varying |R|** on Served Rate and Execution Time, comparing sequential and TDSP clustering approaches.

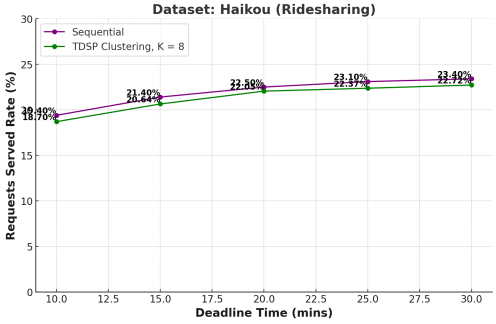
6.3.5 Varying the Delivery Deadline DDL_r

We do not vary the requests' Deadlines (aka waiting times) for the Cainiao experiments because it contains original deadline information and only vary the deadlines for the ridesharing datasets.

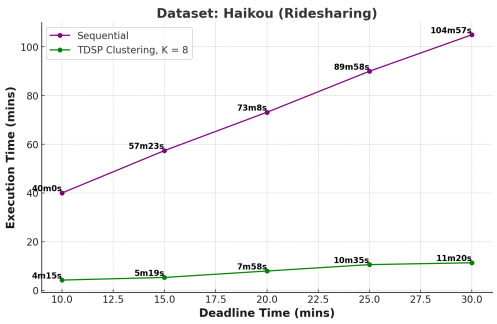
We see similar results to the previous subsections when we vary DDL across datasets.

For **Haikou**, we see a general trend of mild increase in the served rate for a disproportionate increase in execution time, as seen in Figure 6.12. This suggests that customers agreeing to a longer detour time does not impact the served rate for Haikou that much. The sequential approach seems to fare better compared to the parallel approach, which witnesses a negligible -0.67% average decrease in served rate Then again, our parallel approach achieves an average speedup factor of 10X compared to the original.

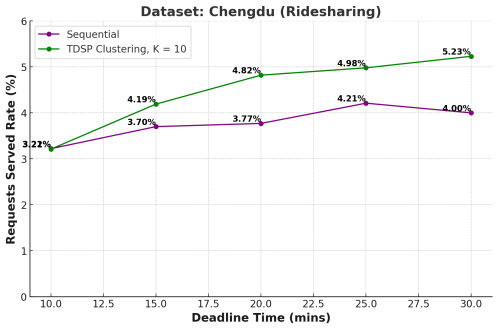
For **Chengdu**, we once again see an increase in served rate for our parallel



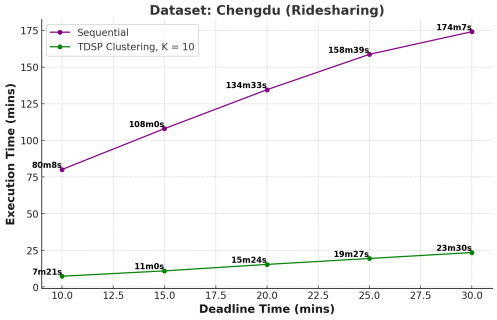
(a) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 8)



(b) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 8)



(c) Impact on Served Rate: Sequential vs. TDSP Clustering (k = 10)



(d) Impact on Execution Time: Sequential vs. TDSP Clustering (k = 10)

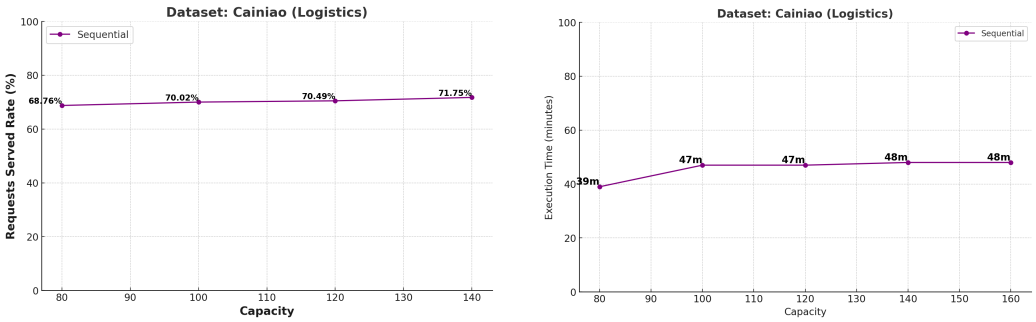
Figure 6.12: [**Ridesharing**] Impact of **Varying** *DDL* on Served Rate and Execution Time, comparing sequential and TDSP clustering approaches.

approach across all deadlines (green line). An average of +0.77% requests served rate is achieved by the parallel approach, which is pretty big considering the absolute highest requests served rate is 5.23% (also achieved by our approach). The parallel approach also shows an upward trend of the served rate while the sequential trends less upward, and for 30 minutes, even goes down. We see a speedup of 11.4X to 7.6X with increasing DDLs. The execution time graph slope of the sequential approach is much higher than the parallel, although both trend linearly.

Varying Capacity

Increasing the capacity for Cainiao barely has an effect on the number of requests served for the baseline algorithm, with an increase of 2.2% for varying the capacity from minimum to maximum (Figure 6.13a). The execution time also remains mostly constant (Figure 6.13b).

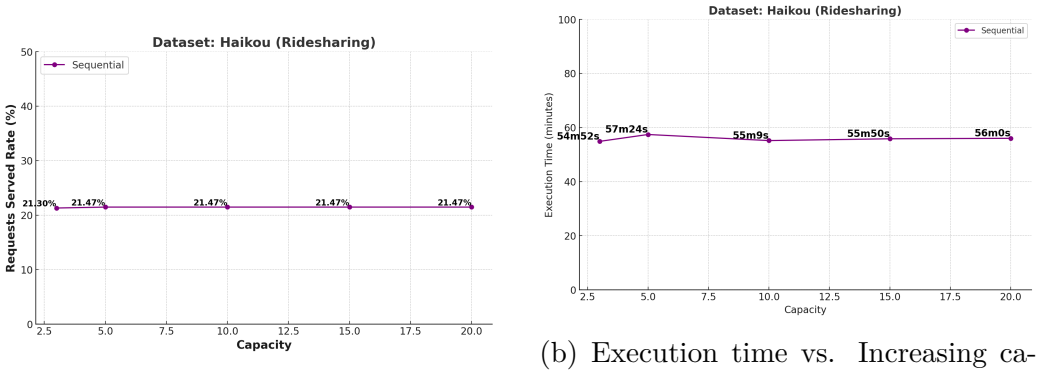
Similarly, varying the capacity seems to have the least effect on the serve rate and execution time (Figure 6.14) for Haikou. We only see a 0.17% increase in the served rate from 3 to 5 and then the served rate plateaus for higher capacities. The execution time also remains mostly constant with the capacity increase. This suggests there isn't much gain to be had by studying this parameter further (except for potential speedups). Therefore, we elected to not conduct further experiments on it using our parallel method.



(a) Served rate vs. Capacity

(b) Execution time vs. Capacity

Figure 6.13: Impact of varying Capacity on Served Rate and Execution Time on the Cainiao Dataset



(a) Served rate vs. Increasing capacity

(b) Execution time vs. Increasing capacity

Figure 6.14: Impact of Varying Capacity on Served Rate and Execution Time for the Haikou Dataset

6.3.6 Multiprocessing Vs. One-by-One Run

We leverage multiprocessing (bound to 16 cores) and assign each (C, W) pair to a specific core in our system. We then launch N number of sequential operator

executables where N = number of clusters. The execution times reported for the parallel approaches are the result of this multiprocessing, made possible with the help of GNU parallel. In this subsection, we try to answer the question: **where the speedups are exactly coming from?** How much of the speedups can be attributed to our partitioning method, and how much can be attributed to parallel computing?

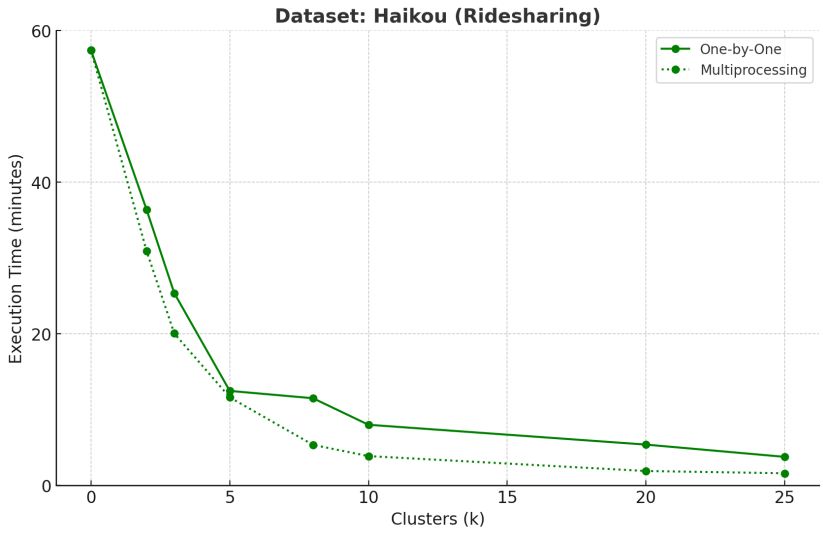


Figure 6.15: Execution Time: One-by-One vs. Multiprocessing on the Haikou Dataset for TDSP clustering. Maximum jobs run in parallel = 15

As seen in Figure 6.15, there is a higher speedup achieved when partitioning + multiprocessing is performed on the Haikou Dataset for fixed parameters (time taken from Figure 6.7b). This is seen by the dotted line in the figure. Performing partitioning ONLY on the requests dataset and then running the sequential algorithm on each of the clustered requests one after the other still results in a much higher speedup than the sequential algorithm. This shows that although multiprocessing does help in gaining additional speedups, **the main speedup contributor is the partitioning itself**.

A reason for multiprocessing not providing substantial speedups could be that we try to load the Td-g-TREE into RAM for all datasets N number of times when we run the executables in parallel. For example, the size of the td-g-tree for Haikou is **900MB**. When we run the insertion operator on 15 Haikou requests clusters in parallel on our personal computer, we observe a huge increase in usage of RAM and Swap memory (Figure 6.16). If we try to run more than 15 jobs in parallel, the processes get killed because we run out of memory. Thus, we are constrained to 15 jobs max in parallel for Haikou. For Chengdu, the td-g-

Table 6.8: Speedup Factor w.r.t Multiprocessing

k-value	One-by-One	Multiprocessing	Speedup
1	57m24s	57m24s	1.0
2	36m24s	30m57s	1.18
3	25m22s	20m3s	1.27
5	12m28s	11m37s	1.07
8	11m30s	5m21s	2.15
10	8m0s	3m52s	2.07
20	5m23s	1m54s	2.83
25	3m46s	1m36s	2.35

tree's size is 7.8GB. Trying to run more than 2 jobs concurrently results in killed processes for it. This suggests that we either need a larger memory size for the computer, or that we should find a way to load the tree only once and be read by all the processes (through mmap() or other shared memory resources), or we need to break down the huge chengdu graph into disconnected subgraphs so the td-g-tree structures are smaller for each of the requests clusters.

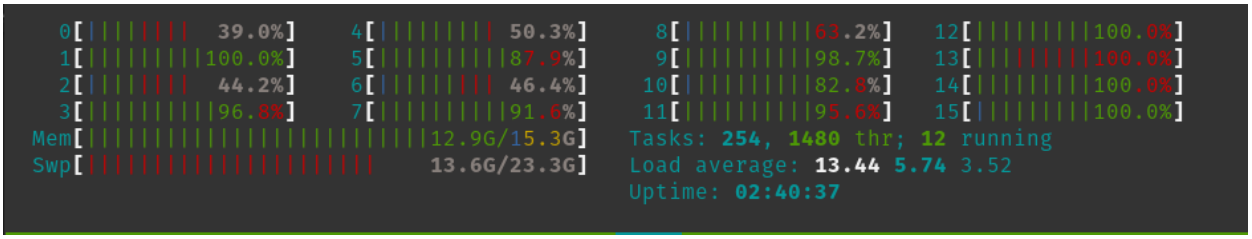


Figure 6.16: Htop Process Viewer showing that both Memory and SWAP is near full when running 15 Haikou jobs in parallel

Chapter 7

Bottleneck Analysis

7.1 Chapter Outline

In this chapter, I investigate the primary bottlenecks that cause the Insertion Operator over TDRNs to have huge computational costs, making it unfeasible for real-world usage: the **TD-G-tree** data structure that is used to store the TDRN, and its primary operation **query (u,v,t)** with which the time-dependent shortest paths between vertices are calculated. I also explain the reason behind our parallel approach being faster than the sequential.

7.2 Need for Bottleneck Analysis

Identifying bottlenecks acts as the first step for making computationally expensive programs efficient without upgrading hardware (which is cost-intensive) or changing your entire program structure. [DCG25] use the Intel VTune Profiler [Int25] to identify performance bottlenecks in MRI pre-processing programs, demonstrating the need for profiling in real-world applications. They identify which functions take up the most CPU time which is what we aim to do in this chapter. The Intel Vtune Profiler is a well known tool for detecting performance bottlenecks caused by high computational spots in the code (hotspots). It provides the option for Micro-architectural analysis (whether the program's pipeline is front-end bound or back-end bound), and gives tips on possible parallelization of tasks. We use it identify **the hotspots and memory-bound issues** in the insertion operator.

7.3 Observation 1: TD-G-Tree

As mentioned before, the TD-G-Tree remains as one of the state-of-the-art data structures for representing the time-dependency of real-world road networks [WLT19], but it doesn't come without its issues. Firstly, let's look at the index size:

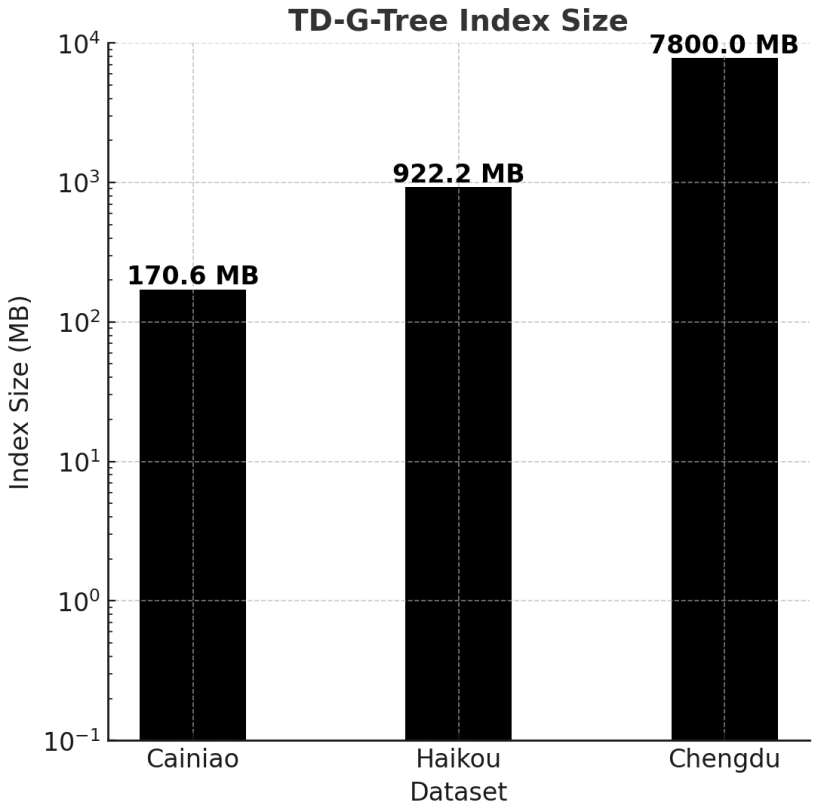


Figure 7.1: Index Size (MB) vs. Datasets

Table 7.1: Dataset Statistics

Category	**Dataset**	**(Vertices, Edges)**
Ridesharing	Chengdu	(423434, 913718)
	Haikou	(41542, 89206))
Logistics	Cainiao	(9936, 23872)

This index size was measured by putting the TD-G-Tree data structure into a text file and measuring its size by looking at its properties. In Figure 7.1,

it is evident that the size of the TD-G-Tree is linearly proportional to the size of the road network (sizes are given in Table 7.1). For a larger dataset like Chengdu containing over 900,000 edges and 400,000 vertices, the size of the data structure is 7.8 GB. Even for a smaller dataset like Cainiao (which is a subnetwork of Shanghai), the size is 170 MB. The tree is getting loaded into RAM every single time we want to run the insertion operator program. This could be a very big issue for hardware with smaller RAMs where even loading the tree would be enough to prematurely kill the process.

It is also noted by [GZC24a] that as the number of interpolation points increases, the index sizes of all datasets will increase too. Chengdu's td-g-tree takes up 7.8GB of memory for only two interpolation points per edge. Adding more interpolation points (which might be necessary to represent complex road network time-dependencies) would result in substantial memory costs. Time-Dependent Road Networks larger than Chengdu might not be able to use the td-g-tree more than once (since one copy of the tree is loaded for each multiprocessing job) unless they are used on larger RAM devices.

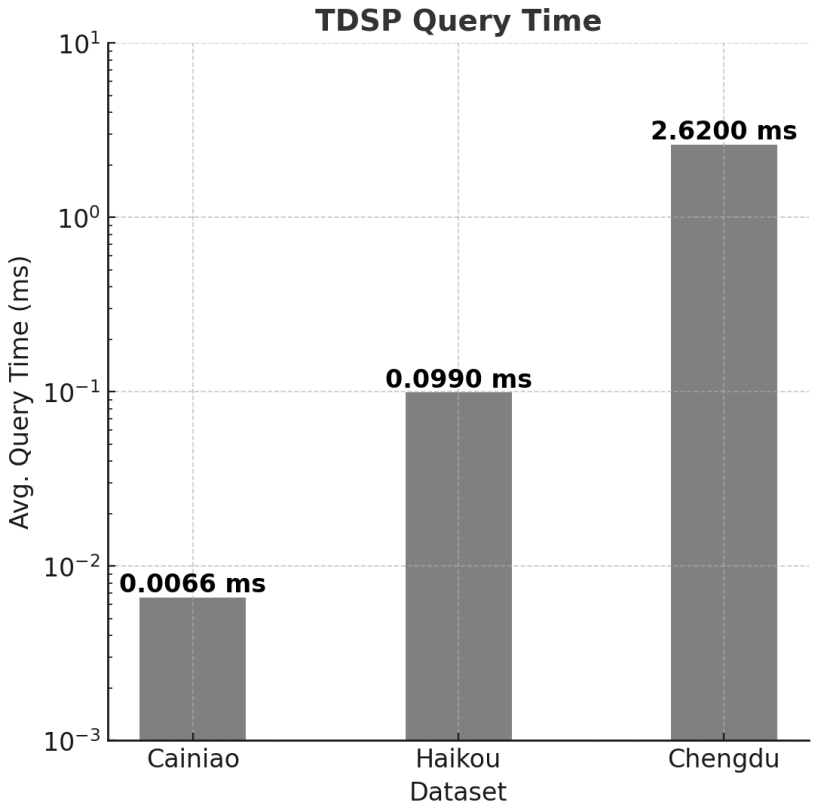


Figure 7.2: Average Query time (milliseconds) vs. Datasets

7.4 Intel VTune Profile Run

The TDSP query operation from the td-g-tree outperforms most shortest-path query algorithms, as seen in [WLT19]. We can observe from Figure 3.1 that the TDSP query time is directly related to the size of the dataset. For example, it takes an average of 2.62 milliseconds to finish a query(u,v,t) computation for Chengdu. The sheer number of query invocations present in an insertion operator simulation, this query ends up becoming a huge bottleneck. This is verified by the Intel VTune Hotspot analysis that I performed in the upcoming subsection.

Using a fixed set of parameters from chapter 6 for every dataset, I ran VTune's hotspot analysis to identify which part of the code takes the most amount of CPU time. The tdsp query takes **97.5%** of CPU time for the whole program run (Figure 7.3), making it the primary bottleneck that we need to mitigate to make the insertion operator scalable.

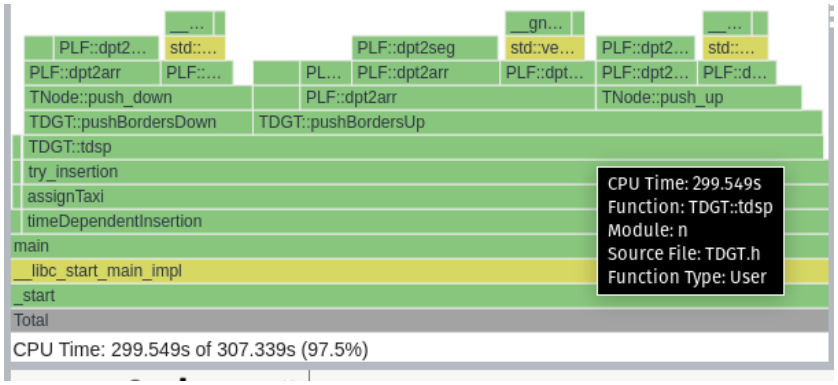


Figure 7.3: Intel VTune Flame Graph showing TDSP query and its components to be the main bottleneck

The Insertion Operator simulations are DRAM (Dynamic RAM) bound (Figure 7.4). This means that the performance of the program is primarily limited by the speed of accessing and transferring data from the system's DRAM. **37.5%** is the amount of time that the CPU was stalled due to pending memory requests during the program run. It is possible that the td-g-tree is the primary reason for these stalls.

7.5 Query Invocation Benchmark

In Figures 7.5 and 7.6, we benchmark the number of query invocations achieved in both the sequential and parallel approaches for both ridesharing (Haikou) and the logistics (Cainiao) scenarios. The setup is for fixed parameters for the Shanghai

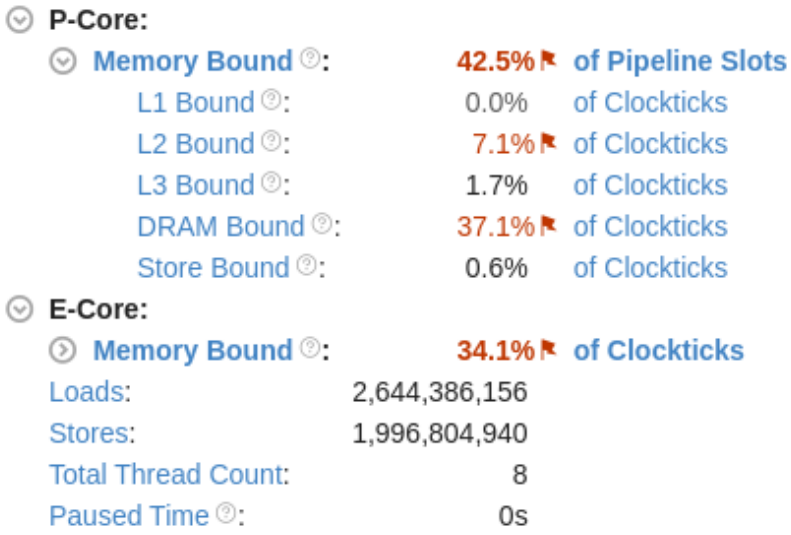


Figure 7.4: Intel VTune Memory Access Profile showing that the Insertion Operator program run is DRAM bound

and Haikou datasets and we run the sequential insertion operator and the TDSP clustering approach with k = 20 for Shanghai, and k = 8 for Haikou which are the optimal k numbers for each, respectively. We observe that the clustered approach results in a 100x reduction in the number of query invocations. This clearly demonstrates why our clustered approach is faster than the sequential.

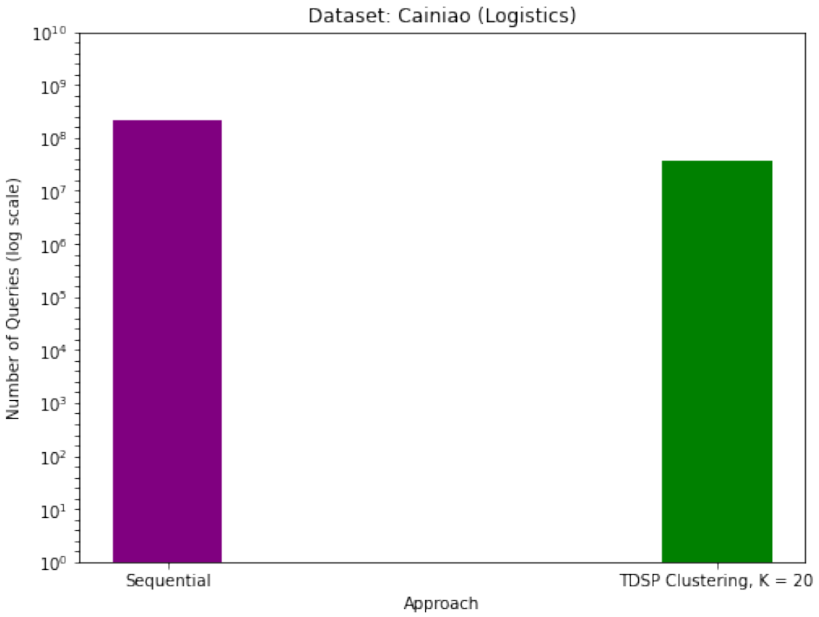


Figure 7.5: Query invocations comparison between the sequential and the TDSP clustering approach for Logistics.

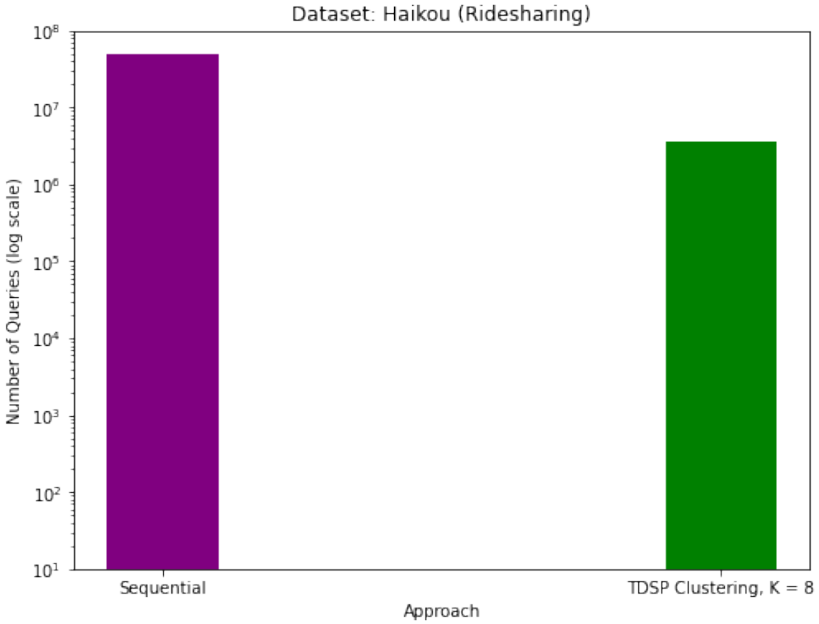


Figure 7.6: Query invocations comparison between the sequential and the TDSP clustering approach for Ridesharing.

Chapter 8

Conclusion and Future Work

This chapter summarizes the motivations and goals of the thesis, the steps taken to achieve said goals, the experiments conducted to evaluate the steps, and the results obtained. We also provide potential future work avenues with the algorithms I designed.

8.1 Conclusion

The insertion operator is the fundamental operator used in route planning for shared mobility. It's a state-of-the-art operation that minimizes the detour travel time for a worker when a request gets added to their route in real-time. This, in combination with an objective function like maximizing requests served, increasing customer satisfaction, or minimizing total travel time, is a game-changer for optimal route planning. The adoption of the insertion operator over time-dependent road networks makes it more applicable to real-life scenarios since the travel time between two locations can change drastically over the day. [GZC24b] achieve modeling this real-world scenario by using a TD-G-Tree (graph partioning, height balanced tree data structure) to store a Time-Dependent Road Network and designing a linear-time insertion operator, which indeed turns out to be fastest amongst its competitors over TDRNs specifically. It has a linear time complexity w.r.t the number of requests that get assigned to a worker. But, when their operator is ran over dense, real-world datasets of cities like Chengdu, Haikou, and Cainiao, it's seen that the total running time is high. Its performance deteriorates proportionally to the number of tree nodes in the td-g-tree and the interpolation points in the TDRN. This performance degradation makes the insertion operator non-scalable for real-world data and non-profitable for any company that wants to churn out simulations at a rapid rate for their workforce and requests and derive insights from it. In this thesis, I created two partitioning

methods that partition the requests and workers data that the sequential insertion operator works with. The partitioning methods create disjoint clusters of the requests data and pair them with disjoint subsets of the requests data. I did this to reduce the search space of every cluster-workers pair since it is simply just a small instance of the larger requests and workers sets available to the sequential operator. By making these cluster-workers pairs, I also expose data-level parallelism within the code, which means that I can run the sequential operator on these pairs concurrently, with minimum of at least two jobs running at the same time (maximum subject to hardware limitations). I use k-means clustering as the clustering method for making these partitions.

The first partitioning method was End Node K-means Clustering, based on the geospatial location of the destination of requests. This partition itself leads to a minimum of **2X** speedup in simulation time compared to the sequential and the speedup factor keeps increasing with increasing k. This method also sees a slight degradation in the requests served rate, which is inversely proportional to increasing k.

The second, and superior method that I created was based on shortest-path clustering via k-means and opportunistic worker allocation. Clustering on TDSPs provides us with more information about the requests themselves and helps us effectively categorize them into short-distance, medium-distance, long-distance, etc., requests. Under the assumption that shorter distance requests have more probability of getting served, we assign more workers to requests clusters that on average, have shorter distance requests. This results in more requests served than the End Node clustering and matches (and in some cases exceeds) the sequential's served rate, while providing a minimum of 2X speedups.

We effectively develop a data partitioning system that provides customizability and flexibility to the user. If a user wants rapid simulations, we provide the option of increasing k up to 25 and witnessing a minimum of **8.5X** speedups with a $<$10% served rate degradation. Higher speedups can be achieved by increasing k with diminishing served rates. We found out via the elbow method that the sweet spot lies between k $= 8$ and k $= 20$. Thus, if they want to prioritize more requests, they can have a low k-value and still witness a noticeable decrease in execution time. The experiments conducted are exhaustive and done across 3 different datasets to examine the TDSP partitioning method with scrutiny and enhance the credibility of my partitioning method.

I also provide a bottleneck analysis of the insertion operator and the TD-G-Tree structure. Essentially, if a road network is large, and has many interpolation points (to signify time-dependence), the TD-G-Tree data structure is large (takes up a lot of RAM and SWAP memory), and the tdsp query time increases. This query remains the to be the bottleneck in both the sequential and the parallel approaches. The parallel approach results in lower total query invocations overall,

but the large size of the td-g-tree means that there's an upper limit to which you can run the insertion operator in parallel.

8.2 Future work

The work presented in this thesis opens up lots of avenues for experimental analysis. We assume there's a fixed depot (could be a taxi depot, amazon delivery van depot, etc.) that the workers start at before beginning to serve requests. Future work could look at exploring ridesharing scenarios where workers start at random positions on the road network. Mini depot scenarios could be set up where workers start waiting at ferry terminals, airports, etc., and observe how the RPSM simulations react. Our algorithms provide the opportunity to explore such scenarios at fast rates.

Another potential future work is eliminating loading the td-g-tree multiple times. Although the multiprocessing approach gets the job done in terms of doing parallel computing, the whole program is effectively DRAM bound (by VTune analysis). Multiple loads of the tree give unnecessary load to a computers ram and limit the number of jobs that can be run in parallel. Going beyond the maximum amount leads to hardware crashes or termination of the processes due to the memory being overfull. Loading the tree once and giving the partitions access to this global tree when working concurrently. Therefore, a multithreaded approach could be beneficial to extending the work done in this thesis and effectively mitigating a large bottleneck of the insertion operator, resulting in even faster simulations.

Bibliography

[AAS21] Alguliyev, R. M., Aliguliyev, R. M., AND Sukhostat, L. V. Parallel batch k-means for Big data clustering. *Computers & Industrial Engineering* 152 (2021), 107023. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2020.107023. URL: https://www.sciencedirect.com/science/article/pii/S0360835220306938.

[Anw+14] Anwar, T., ET AL. Spatial partitioning of large urban road networks (Jan. 2014). DOI: 10.25916/sut.26224499.v1. URL: https://figshare.swinburne.edu.au/articles/conference_contribution/Spatial_partitioning_of_large_urban_road_networks/26224499.

[CL03] Cordeau, J.-F., AND Laporte, G. The dial-a-ride problem (DARP): Variants, modeling issues and algorithms. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies* 1 (2003), 89–101.

[Dai+22] Dai, R., ET AL. Optimization and evaluation for autonomous taxi ride-sharing schedule and depot location from the perspective of energy consumption. *Applied Energy* 308 (2022), 118388. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2021.118388. URL: https://www.sciencedirect.com/science/article/pii/S0306261921016263.

[Dan+23] Dan, T., ET AL. Double Hierarchical Labeling Shortest Distance Querying in Time-dependent Road Networks. In: *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2023, 2077–2089. DOI: 10.1109/ICDE55515.2023.00161.

[DCG25] Dugré, M., Chatelain, Y., AND Glatard, T. An analysis of performance bottlenecks in MRI preprocessing. *GigaScience* 14 (Mar. 2025), giae098. ISSN: 2047-217X. DOI: 10.1093/gigascience/giae098. eprint: https://academic.oup.com/gigascience/article-pdf/doi/10.1093/gigascience/giae098/62337828/

giae098.pdf. URL: https://doi.org/10.1093/gigascience/giae098.

[Dij59] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[Got+19] Gottesbüren, L., ET AL. Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies. *Algorithms* 12, 9 (2019). ISSN: 1999-4893. DOI: 10.3390/a12090196. URL: https://www.mdpi.com/1999-4893/12/9/196.

[GZC23] Gong, Z., Zeng, Y., AND Chen, L. *A Fast Insertion Operator for Ridesharing over Time-Dependent Road Networks*. 2023. arXiv: 2303.03614 [cs.DB]. URL: https://arxiv.org/abs/2303.03614.

[GZC24a] Gong, Z., Zeng, Y., AND Chen, L. Querying Shortest Path on Large Time-Dependent Road Networks with Shortcuts. In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2024, 4532–4544. DOI: 10.1109/ICDE60146.2024.00345.

[GZC24b] Gong, Z., Zeng, Y., AND Chen, L. Real-Time Insertion Operator for Shared Mobility on Time-Dependent Road Networks. *Proc. VLDB Endow.* 17, 7 (May 2024), 1669–1682. ISSN: 2150-8097. DOI: 10.14778/3654621.3654633. URL: https://doi.org/10.14778/3654621.3654633.

[HAD09] Hoffmann, H. C., Agarwal, A., AND Devadas, S. Partitioning strategies for concurrent programming (2009).

[Hua+14] Huang, Y., ET AL. Large scale real-time ridesharing with service guarantee on road networks. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 2017–2028. ISSN: 2150-8097. DOI: 10.14778/2733085.2733106. URL: https://doi.org/10.14778/2733085.2733106.

[Int25] Intel Corporation *Intel® VTune™ Profiler*. Accessed 26 Apr 2025. 2025. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.i6xhgk.

[Li+22] Li, J., ET AL. Efficient kNN query for moving objects on time-dependent road networks. *The VLDB Journal* 32, 3 (July 2022), 575–594. ISSN: 1066-8888. DOI: 10.1007/s00778-022-00758-w. URL: https://doi.org/10.1007/s00778-022-00758-w.

[LX20] Lin, X., AND Xu, J. Road network partitioning method based on Canopy-Kmeans clustering algorithm. *Archives of Transport* 54, 2 (June 2020), 95–105. DOI: 10.5604/01.3001.0014.2970. URL: https://www.archivesoftransport.com/index.php/aot/article/view/199.

[NKR13] Naik, D. S. B., Kumar, S. D., AND Ramakrishna, S. V. Parallel processing of enhanced K-means using OpenMP. In: *2013 IEEE International Conference on Computational Intelligence and Computing Research*. 2013, 1–4. DOI: 10.1109/ICCIC.2013.6724291.

[Ota+17] Ota, M., ET AL. STaRS: Simulating Taxi Ride Sharing at Scale. *IEEE Transactions on Big Data* 3, 3 (2017), 349–361. DOI: 10.1109/TBDATA.2016.2627223.

[Ped+11] Pedregosa, F., ET AL. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[Rob49] Robinson, J. J. On the Hamiltonian Game (A Traveling Salesman Problem). In: 1949. URL: https://api.semanticscholar.org/CorpusID:122464330.

[Sch25] Schneider, T. W. *NYC Taxi and Ridehailing Data (Uber, Lyft, etc.)* Accessed: 2025-01-08. 2025. URL: https://toddwschneider.com/dashboards/nyc-taxi-ridehailing-uber-lyft-data/.

[SD21] Sinclair, C., AND Das, S. Traffic Accidents Analytics in UK Urban Areas using k-means Clustering for Geospatial Mapping. In: *2021 International Conference on Sustainable Energy and Future Electric Transportation (SEFET)*. 2021, 1–7. DOI: 10.1109/SeFet48154.2021.9375817.

[Tan11] Tange, O. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb. 2011), 42–47. DOI: 10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[Ton+18] Tong, Y., ET AL. A unified approach to route planning for shared mobility. *Proc. VLDB Endow.* 11, 11 (July 2018), 1633–1646. ISSN: 2150-8097. DOI: 10.14778/3236187.3236211. URL: https://doi-org.ezproxy.library.uvic.ca/10.14778/3236187.3236211.

[Ton+22] Tong, Y., ET AL. Unified Route Planning for Shared Mobility: An Insertion-based Framework. *ACM Trans. Database Syst.* 47, 1 (May 2022). ISSN: 0362-5915. DOI: 10.1145/3488723. URL: https://doi.org/10.1145/3488723.

[Wan+22] Wang, Y., et al. Reachability-Driven Influence Maximization in Time-dependent Road-social Networks. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2022, 367–379. DOI: 10.1109/ICDE53745.2022.00032.

[WLT19] Wang, Y., Li, G., and Tang, N. Querying shortest paths on time dependent road networks. *Proc. VLDB Endow.* 12, 11 (July 2019), 1249–1261. ISSN: 2150-8097. DOI: 10.14778/3342263.3342265. URL: https://doi.org/10.14778/3342263.3342265.

[WN21] Wang, S., and Noland, R. B. Variation in ride-hailing trips in Chengdu, China. *Transportation Research Part D: Transport and Environment* 90 (2021), 102596. ISSN: 1361-9209. DOI: https://doi.org/10.1016/j.trd.2020.102596. URL: https://www.sciencedirect.com/science/article/pii/S1361920920307823.

[Xu+19] Xu, Y., et al. An Efficient Insertion Operator in Dynamic Ridesharing Services. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, 1022–1033. DOI: 10.1109/ICDE.2019.00095.

[Zen25] Zeng, Y. *Shanghai Dataset*. Accessed: 2025-01-28. 2025. URL: https://yzengal.github.io/datasets/.

[Zuo+21] Zuo, H., et al. High-capacity ride-sharing via shortest path clustering on large road networks. *J. Supercomput.* 77, 4 (Apr. 2021), 4081–4106. ISSN: 0920-8542. DOI: 10.1007/s11227-020-03424-6. URL: https://doi.org/10.1007/s11227-020-03424-6.