On the Practical Viability of Competitive Algorithms for the Homing Online Traveling Salesperson Problem: Empirical Evaluation and Practical Recommendations

by

Su Dong

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2025

Thesis advisors

Dr. Parimala Thulasiraman and Dr. Ben Li

Author

Su Dong

On the Practical Viability of Competitive Algorithms for the Homing Online Traveling Salesperson Problem: Empirical Evaluation and Practical Recommendations

Abstract

The Homing Online Traveling Salesperson Problem (H-OLTSP) models a dynamic routing scenario in which a server must respond to requests revealed over time while always returning to a fixed origin. Here, the server represents a moving agent, such as a robot or vehicle, that travels through the metric space to serve requests. Although competitive algorithms such as Plan At Home (PAH) and Greedily Traveling between Requests (GTR) have been proposed under competitive analysis, their behavior under practical conditions remains largely untested. Competitive analysis reflects worst-case performance but may not capture how these algorithms perform in realistic environments where response speed and solution efficiency are critical.

This thesis bridges that gap by empirically evaluating PAH and GTR through simulation. Each algorithm is implemented with multiple scheduling strategies and tested under controlled experimental settings that vary the number of requests and the distribution of their release times. The evaluation focuses on two core aspects:

responsiveness, which reflects how quickly the server reacts to incoming requests, and efficiency, which reflects how effectively the algorithm solves each H-OLTSP instance.

The study compares GTR and PAH directly and evaluates how each performs when combined with different scheduling algorithms under varying conditions. The results provide practical insight into the trade-offs between response speed and instance-solving efficiency, revealing performance differences that are not visible through theoretical analysis alone. This work offers a grounded perspective on the real-world suitability of competitive online algorithms. It guides their application in dynamic routing applications that require timely decisions and low completion time.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to begin by thanking my thesis advisors, my parents, and all the people who have helped me.

"Order is merely the possibility of repetition."
— Arthur Schopenhauer

Chapter 1

Introduction

In many modern systems, such as real-time services, delivery platforms, and automated robots, knowing all the information about a task in advance is impossible. For example, a ride-hailing application does not know all future ride requests, and a network router does not know all future data packets. These situations require decisions to be made as new information becomes available. Problems like these are called *online problems*, and the methods used to solve them are called *online algorithms*.

In an online problem, the input is not provided all at once. That is, the input is not known a *priori*. Instead, it is provided at a given timestep. As new inputs arrive, the algorithm makes decisions based on the acquired knowledge. Once a decision is made, it cannot be changed later. This differs from an offline problem, where the algorithm sees all the input at the beginning and can make the best decision based

on the entire input. An online algorithm solves an online problem. It reads the input piece by piece and reacts accordingly. Since it cannot see the future, it must make smart guesses or follow good rules to handle a new piece of input. (See Borodin et al. [2] for more details regarding online algorithms)

Offline algorithms are a subclass of online algorithms, assuming that input data is finite and the online algorithm can wait until all input data becomes available. In this setting, an online algorithm can delay its decisions and collect the entire input before solving the problem. Once all data is known, the online algorithm can process the instance in the same way as an offline algorithm. This means the offline algorithm is simply a special case where the online algorithm chooses to act only after seeing the full input.

This thesis focuses on the Online Traveling Salesperson Problem (OLTSP), a variation of the classic Traveling Salesperson Problem (TSP) in the online setting. In OLTSP, new locations are given one at a time while the salesperson is already moving. The goal is then to visit every location as quickly as possible, even though the list of locations is unknown.

Section 1.1 provides a formal description of the OLTSP. Section 1.2 explains the homing version of OLTSP. Section 1.3 describes the practical viability of the homing version of OLTSP, highlighting the practical relevance and research gaps addressed. The goal and motivation behind this work are discussed in Section 1.4. Section 1.5 explains the contributions of this thesis. Finally, Section 1.6 provides an overview of

the thesis structure.

1.1 Online Traveling Salesperson Problem

In the classical TSP, the goal is to find the shortest tour that visits each location exactly once and returns to the starting point. This is an offline problem where all the locations to be visited are given as input. Computing such a tour is NP-hard on general metric spaces. As a result, even the offline version of OLTSP is computationally intractable unless $P = NP$. Based on the work by Ausiello et al. [3], we describe the OLTSP more formally in Definition 1.

Definition 1. *Online Traveling Salesperson Problem (OLTSP)*

The input to the OLTSP consists of the following:

- *A metric space (M, d), where M is a finite set of locations and $d : M \times M \to \mathbb{R}_{\geq 0}$ is a distance function satisfying the standard metric properties: non-negativity, identity, symmetry, and the triangle inequality.*
- *A designated starting point $o \in M$, called the* origin*, where the server is initially located at time zero.*
- *An online sequence of requests $Q = \{(t_1, p_1), (t_2, p_2), \ldots, (t_n, p_n)\}$, where each $p_i \in M$ is a requested location and each $t_i \in \mathbb{R}_{\geq 0}$ is the release time of the request. The sequence is ordered such that $t_i \leq t_{i+1}$ for all i, and the request (t_i, p_i) is revealed to the algorithm only at time t_i.*

The server moves through the metric space at unit speed, so traveling between any two locations x and $y \in M$ requires exactly $d(x, y)$ units of time. A request (t_i, p_i) is considered served if the server visits point p_i at some time $t \geq t_i$.

The objective is to guide the server, in an online fashion, to serve all requests in Q. The goal is to minimize the ***completion time****, which is defined as the total amount of time that elapses from the start (time zero) until all requests have been served. This includes both the time spent traveling and any time the server remains idle.*

Although the server moves at unit speed, the *completion time* is not necessarily equal to the total distance traveled. This is because the completion time includes all elapsed time, including both movement and idle periods, while the total distance only accounts for the physical path the server follows.

To make this distinction clear, consider an OLTSP instance with two requests. The first request is released at time 0 for location p_1, and the second request is released at time $d(o, p_1) + x$ for location p_2, where $x > 0$. Let the server follow a simple online algorithm that behaves as follows:

- The server completes requests one by one. At any time, if there are unserved and released requests, the server selects the closest such request from its current location and begins moving toward it.
- After completing a request, the server again selects the next closest unserved and released request and continues in the same manner.

- If there are no unserved released requests available at the current time, the server remains idle at its current location until a new request is released.
- Once the server starts moving toward a selected request, it does not change its target, even if another request is released while en route.

In this scenario, the server begins at the origin o and, at time 0, receives the first request for p_1. It moves directly from o to p_1 and serves the request. After serving p_1, there are no further known requests, so the server remains idle at p_1.

At time $d(o, p_1) + x$, the second request is released at p_2. The server, now aware of this request, begins moving from p_1 to p_2 and serves it upon arrival.

In this case, the total distance traveled by the server is $d(o, p_1) + d(p_1, p_2)$, corresponding to the actual paths taken. However, the completion time is $d(o, p_1) + x + d(p_1, p_2)$, because the server must idle for x units of time between finishing the first request and starting to serve the second.

This example shows that, even under a greedy strategy that always moves toward the closest known request and avoids any intentional waiting, the completion time can still be strictly greater than the total distance traveled, due to the delayed release of information in the online setting.

1.1.1 OLTSP Versions:

Based on the definition, there are two types of OLTSP.

- Homing OLTSP (H-OLTSP): Starting at the origin, the goal is to minimize the total completion time to serve all requests presented, returning to the origin o at the end.

- Nomadic OLTSP (N-OLTSP): Starting at the origin, the goal is to minimize the total completion time to serve all the requests presented.

The difference between the versions is whether the server must return to the origin after finishing all the requests. All other parts of the problem are the same. Taking an example, we assume a metric space $< M, d >$ where $M = \{0, 1, 2, 3\}$, $d(x, y) = |x-y|$. An OLTSP instance can be $I = \{< M, d >, o = 0, Q = \{(0, 1), (10, 2)\}\}$. To explain this example in more detail, the metric space of this example is the positive real line, and all possible points are $0, 1, 2, 3$, and 4 on that real line. It requires the server to access two requests starting from point 0. The first request is released at time 0 and asks the server to visit point 1. The second request is released at time 10 and asks the server to visit point 2. After completing the request, if the server is solving N-OLTSP, then the goal is achieved. Accordingly, if the server is solving H-OLTSP, it has to return to the origin point 0.

In this thesis, we focus on the homing version of OLTSP (H-OLTSP) on metric spaces. In contrast to a real line model, the metric space model encompasses many real-world applications, including mobile robotics, automated delivery systems, real-time logistics, and dynamic task scheduling.

1.2 H-OLTSP

To our knowledge, Ausiello et al. [3] proposed two theoretical algorithms, Greedily Traveling between Requests (GTR) and Plan At Home (PAH). These algorithms are designed with provable worst-case guarantees based on competitive analysis (described in Appendix B.1), which evaluates how closely an online algorithm approximates the performance of an optimal offline algorithm that has full knowledge of the request sequence in advance.

In addition to their worst-case guarantees, GTR and PAH algorithms represent two fundamentally different approaches to online decision-making in the presence of dynamically released requests. The PAH algorithm constructs a complete route that *starts and ends at the origin* and includes all currently unserved requests. While the server follows this planned route, the algorithm continuously monitors for newly released requests. When a new request arrives, the PAH algorithm evaluates whether the server should return to the origin to initiate a rescheduling step. This decision is based on the relative positions of the server, the origin, and the new request. If returning is deemed beneficial, the server proceeds back to the origin, and a new route is computed that includes all requests released up to that point.

In contrast, the GTR algorithm adopts a more *reactive* strategy: it recomputes a new path once a new request is released, even while the server is in motion. This allows GTR to adapt rapidly to changes, but may also introduce instability or inefficiencies in specific scenarios.

The PAH and GTR algorithms will be discussed in detail in Chapter 3 and Chapter 4, the focus of this thesis.

1.3 Practical Viability of the H-OLTSP Algorithms

The PAH and GTR algorithms have been theoretically well-studied in the literature. However, the viability and performance of these algorithms in a practical setting remains largely unexplored. Existing analyses are confined to competitive-ratio guarantees under worst-case conditions, which do not account for the complexities in real-time environments. In practice, systems implementing online routing must respond to requests quickly, often under limited computational budgets. They may be unable to compute optimal or near-optimal tours at every update. As a result, the gap between theoretical assumptions and practical feasibility becomes significant.

Another significant limitation is that the performance of GTR and PAH algorithms has not been evaluated when combined with scheduling algorithms that are designed for practical efficiency rather than worst-case guarantees. While the original studies analyze these algorithms assuming that each scheduling step uses an exact or approximate solution with known performance bounds, they do not examine how the algorithms behave when paired with more lightweight, heuristic-based procedures typically used in real systems.

Since our goal is not to derive new theoretical results, we are free to test GTR and PAH using scheduling strategies that are efficient in practice, even if they lack formal

analytical guarantees. This approach allows us to assess how well these algorithms function under realistic constraints and whether their theoretical promise translates into practical viability.

1.4 Motivation and Goal of the Thesis

Most existing algorithms for the H-OLTSP have been evaluated primarily through competitive analysis (Appdix B.1), focusing on worst-case guarantees. However, little is known about how these algorithms perform in practice, especially under realistic conditions with varying request loads (Definition 5). This gap between theoretical analysis and practical behavior creates uncertainty when choosing algorithms for real-time applications. There is a need to study how algorithms like PAH and GTR behave in concrete scenarios, not just in theory. This thesis is motivated by the lack of empirical evaluation and aims to provide practical insight into the strengths and limitations of these algorithms when used with different scheduling strategies.

The goal of this thesis is to understand the practical effectiveness of the PAH and GTR algorithms when applied to H-OLTSP. It focuses on evaluating how quickly each method responds to newly released requests and how efficiently it solves problem instances. The study compares different scheduling strategies applied to both algorithms across various request load (Definition 5) conditions. It also directly compares the overall performance of PAH and GTR to identify their practical strengths and weaknesses. The aim is to provide insight into how these algorithms perform

in realistic settings where quick decisions and low completion time (objective value) are essential.

1.5 Contributions

The thesis:

1. presents an empirical evaluation of the PAH and GTR algorithms in practical settings, to examine whether the behaviors and advantages suggested by their theoretical analysis still hold when the algorithms are applied in realistic environments.

2. evaluates the impact of different scheduling strategies, in PAH and GTR algorithms, to examine how these algorithms perform in settings that reflect real-time routing applications.

3. introduces an evaluation framework under two aspects. *Responsiveness* reflects how quickly the algorithm reacts to newly released requests. *Efficiency* refers to the completion time (objective value) required to serve all requests and return to the origin.

1.6 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 reviews related work on the Online Traveling Salesperson Problem (OLTSP). It summarizes existing theoretical results, competitive algorithms, and related scheduling approaches. Chapter 3 and 4 describes the two online algorithms that are the focus of this study: Plan At Home (PAH) and Greedy Traveling between Requests (GTR). Chapter 5 introduces the experimental framework used to evaluate the practical performance of PAH and GTR. Chapter 6 and Chapter 7 present the empirical results for PAH and GTR when combined with different scheduling algorithms across a range of experimental conditions, respectively. Chapter 8 presents a comparison of PAH and GTR through experimental evaluations. Finally, Chapter 9 concludes the thesis.

Chapter 2

Literature Review

This chapter reviews existing work on the OLTSP and its connection to scheduling strategies. Section 2.1 begins with a summary of major algorithmic results, focusing on known competitive algorithms and the models they address. Section 2.2 explain a discussion of limitations and gaps in the literature. Section 2.3 then explains relevant route scheduling strategies from the literature pertinent to this work.

2.1 OLTSP Algorithmic Results

The OLTSP algorithms are analyzed using competitive analysis, where performance is measured by the competitive ratio. An algorithm is called x-competitive if its cost is at most x times the cost of the optimal offline solution in the worst case (See Appendix B.2 for details).

Ausiello et al. [3] defined two versions of the OLTSP problem, N-OLTSP, and H-OLTSP, defined in Section 1.1. They proposed the Greedily Travelling between Requests (GTR) and Plan at Home (PAH) algorithms for solving the H-OLTSP on a general metric space, using route scheduling strategies based on the Hamiltonian Path and the Traveling Salesperson Problem, respectively.

2.1.1 GTR algorithm with Hamiltonian Path Scheduling

The authors [3] first proposed the GTR algorithm for N-OLTSP. The GTR algorithm uses the Hamiltonian path problem (Appendix B, Definition 10) as a scheduling problem. In short, the GTR algorithm schedules a new route each time a new request is released. This is done by solving a Hamiltonian path problem on all unserved requests. The server, for example, may be stationary at a specific point (location, say a) or moving between two requested points (b and c). In the first instance, a path is scheduled from a. In the second instance, the algorithm computes two possible Hamiltonian paths. One path starts from the current location and visits point b first. The other path starts from the server's current location and visits point c first. Each path then continues to cover all the requests that have not yet been served. This includes any requests that were part of the old scheduled route but have not yet been visited, and the newly released requests. The server determines the shorter of the two paths and moves along this path until another request is released, and the process is repeated.

Several special cases may arise during GTR algorithm's execution, which are described in more detail later in Section 4.1. Assuming that the server computes the optimal Hamiltonian path at each scheduling time, the GTR is a 2.5-competitive algorithm. It was the first result to provide such a bound for N-OLTSP on a general metric space. Prior to GTR, no deterministic algorithm had been proposed for this setting with any competitive guarantee. Furthermore, no subsequent deterministic algorithm has improved upon this 2.5 ratio in the general case. Thus, GTR continues to represent the best-known theoretical bound for deterministic N-OLTSP on general metric spaces, underscoring its importance as both a theoretical milestone and a baseline for future algorithmic development.

However, the proposed GTR algorithm was used to solve the N-OLTSP, which differs from H-OLTSP. Therefore, in the same article [3], the authors indicated that GTR can be adapted to H-OLTSP by modifying the route to end at the origin, and that this version maintains the same competitive ratio of 2.5 as in N-OLTSP, but they did not show that this ratio is tight on general metric spaces for H-OLTSP. Unlike the GTR for N-OLTSP, which computes a Hamiltonian path starting from the server's current location to visit all unserved requests, the version for H-OLTSP computes a Hamiltonian path with fixed endpoints (details in Appendix B, Definition 12), which also starts from the current location but returns to the origin after visiting all unserved requests, as required by the definition of H-OLTSP.

2.1.2 PAH algorithm with TSP Scheduling

To further improve the competitive ratio of the version of GTR algorithm for solving H-OLTSP, the authors [3] proposed the PAH (Plan At Home) algorithm. The PAH algorithm uses the TSP (details in Appendix B, Definition 9) as the scheduling problem. *In this algorithm, the server always schedules its route at the origin.* Whenever the server is at the origin, it computes a new TSP tour. This TSP tour starts from the origin, visits all the current unserved requests, and then returns to the origin. If new requests are released the server checks if the latest request is farther from the origin than its current position. If this is the case, the server returns to the origin to reschedule all unserved requests; otherwise, it continues the current route. Again, several special cases may arise during execution, which are described in detail later in Section 3.1.

The authors proved that PAH is 2-competitive, assuming the server computes an optimal TSP tour each time.

Polynomial Time Algorithms

No polynomial-time algorithm exists for computing optimal TSP tours or Hamiltonian paths unless $P = NP$. Hence, the authors modified GTR and PAH and proposed two polynomial-time algorithms. In GTR, the server uses the MST (Appendix B, Definition 14) heuristic that runs in polynomial time to solve the Hamiltonian path with fixed endpoints problem for scheduling routes. We call this the

GTR-MST algorithm. In PAH, the server uses the Christofides' heuristic (Section 3.2.3), which runs in polynomial time, to solve the TSP for scheduling routes. This is we call PAH-Chris.

The authors [3] proved that both PAH-Chris are 3-competitive. They also proved that GTR-MST is 3-competitive for N-OLTSP. However, they did not provide a competitive ratio for using GTR-MST to solve H-OLTSP.

2.1.3 Other Works

The algorithms mentioned above are for general metric spaces. The literature also proposes solutions for OLTSP on the real line for certain applications for which scheduling is difficult on metric spaces. In [3], the authors propose algorithms for the server to handle requests on the real line for solving N-OLTSP and H-OLTSP with a competitive ratio of $\frac{7}{3}$ and $\frac{7}{4}$, respectively.

Besides proposing competitive algorithms for solving N-OLTSP and H-OLTSP on metric spaces and the real line, Ausiello et al. [3] also proved their lower bounds. That is, through lower bounds, they showed that no algorithm can achieve a better competitive ratio than these bounds. These lower bounds are shown as follows:

- N-OLTSP: 2 for general metric space (includes real lines)
- H-OLTSP:
  - On general metric space: $2 - \epsilon$ where $\epsilon > 0$

– On real line: $\frac{9+\sqrt{17}}{8}$

Bjelde et al. [4] improved the known lower bounds on the real line. They showed the tight lower bound for H-OLTSP is 2.04 and provided a matching 2.04-competitive algorithm. They also proposed a 1.64-competitive N-OLTSP algorithm that matches the previously known lower bound, proving its tightness.

Blom et al. [5] studied N-OLTSP on the non-negative real line and proposed a tight 1.5-competitive algorithm. They also introduced the concept of a *fair adversary*, which cannot delay requests in a way that unfairly traps the online algorithm. To solve this model, they proposed a 1.28-competitive algorithm.

Bampis et al. [6] considered a variation of OLTSP where the locations of all requests are known in advance, with unknown release times. They showed that the lower bound for H-OLTSP and N-OLTSP in this setting is 1.5, and they proposed algorithms that match these lower bounds. However, the algorithms are not polynomial-time.

Chen et al. [7] introduced randomized algorithms for OLTSP on the real line. They focused on two types: *zealous* and *non-zealous* algorithms. A zealous algorithm always moves to serve unserved requests without waiting. A non-zealous algorithm is allowed to wait for future requests. Against a general adversary, they proposed a randomized zealous algorithm with a competitive ratio of 1.625 and a non-zealous algorithm with a competitive ratio of 1.5. Against a fair adversary, the non-zealous algorithm achieved a competitive ratio of $\frac{9+\sqrt{177}}{16} \approx 1.562$.

Recently, Yeh et al. [8] proposed a new randomized zealous algorithm for the closed version of OLTSP on the line, where the server must return to the origin. Their algorithm, PRZ (Preferential Randomized Zealous), achieves a 5/3-competitive ratio against an oblivious adversary. This improves the previous best bound of 1.75 for deterministic zealous algorithms. They also proved that no randomized zealous algorithm can achieve a better competitive ratio than 1.5.

Recent research has also explored learning-augmented models in the context of OLTSP variants. These models assume that a prediction mechanism is available to estimate future requests. The study of such models opens a new line of research into how theoretical performance guarantees change when predictions are available. Several works [9; 10; 11] have examined different forms of OLTSP, including versions on lines, trees, flowers, etc. Although the problem settings differ, they all share the goal of understanding how predictive information can guide online decision-making. A central question is how much improvement can be achieved when predictions are accurate. In many cases, learning-augmented algorithms provide stronger performance guarantees under accurate predictions, while still retaining acceptable worst-case bounds when the predictions are poor.

2.2 Limitations and Gaps in the Literature

Based on the literature review, to our knowledge, research on the OLTSP is still limited, especially in general metric spaces. Most existing work focuses on specific

cases, such as the real line, grids, or tree structures. These settings are easier to analyze but do not represent many real-world environments, such as road networks, urban layouts, or irregular warehouse systems. There is little work that studies how OLTSP behaves in arbitrary or high-dimensional metric spaces where distances do not follow simple patterns. This makes it difficult to apply the existing results to practical problems.

Another significant gap is the lack of research on heuristic algorithms for OLTSP. In the offline TSP, heuristics like nearest-neighbor, insertion methods, and meta-heuristics (e.g., genetic algorithms, tabu search) are widely used because they can find good solutions within a reasonable time. However, in the online setting, where future input is unknown, very few studies have been done on how such heuristics can be adapted or combined with online decision-making rules.

In addition, there is a lack of experimental work. Many existing studies only provide theoretical results on competitive ratios but do not test how algorithms behave on real-world data. There is little understanding of how factors such as request density or metric structure affect performance. There is also no standard framework for benchmarking online algorithms for OLTSP, which makes it hard to compare results across different studies. Moreover, the lack of experimental studies means that there is currently no empirical evaluation of the practical strengths and limitations of these algorithms.

Overall, these gaps show that the current research on OLTSP is still in its early

stages. Many important directions remain open, including studying general metric spaces, designing practical heuristics, and evaluating algorithms through experiments. Addressing these challenges makes OLTSP models more useful in real-world applications, such as atomic delivery systems, robotics, and online scheduling.

2.3 Scheduling in Online Routing

Scheduling strategies play an essential role in the OLTSP. In this problem, the server must visit a sequence of locations revealed over time. The key challenge is that future requests are unknown. The server must decide where to go based only on current information. At the same time, it must be prepared for new requests that may appear later. In this section, we describe the role of scheduling strategies in OLTSP and the relevant scheduling strategies from the literature.

TSP and Hamiltonian path problem are well-studied and are the foundation for many offline and online routing algorithms. As a result, a large body of work has been developed over the years.

2.3.1 TSP

One of the most influential approximation algorithms for solving the symmetric TSP is the Christofides algorithm [12], which guarantees a solution within 1.5 times the optimal tour length. It builds a minimum spanning tree (MST), adds a minimum-weight perfect matching to connect odd-degree vertices, and then shortcuts repeated

nodes to construct a complete tour. This method remains a widely cited benchmark for approximation performance in TSP research.

For practical applications, especially in large-scale instances, the Lin-Kernighan heuristic is one of the most effective algorithms [13] for improving the tours in solving the TSP. It is a variable-depth local search method based on a generalized k-opt strategy. The algorithm explores sequences of edge exchanges to reduce tour length and adaptively determines the value of k at each step. Despite being heuristic, the Lin-Kernighan approach often produces near-optimal solutions and has been used as a baseline in many experimental studies.

Another vital algorithm for improving the tours in solving the TSP is GENI (Generalized Insertion), proposed by Gendreau et al. [14]. Unlike standard insertion heuristics, GENI considers multiple insertion positions for each new node and applies restricted 3-opt or 4-opt moves during insertion. This allows the algorithm to blend constructive methods with local optimization, making it effective in static and dynamic environments. Gendreau et al. also proposed a complete framework called GENIUS, which combines GENI with a post-optimization procedure known as Unstringing and Stringing (US). This postprocessing phase further refines the tour and improves solution quality. GENIUS has demonstrated strong performance in benchmark evaluations and is adaptable to dynamic and incremental settings, making it relevant for online applications.

The above-mentioned TSP tour scheduling strategies are particularly suitable for

H-OLTSP, where the server must return to the origin after serving all released requests. Ausiello et al. [3] proposed the PAH algorithm in this setting. This algorithm computes an exact TSP tour starting and ending at the origin whenever the server is idle at the origin. Due to the NP-hardness of exact TSP computation, they also proposed a polynomial-time version that replaces the optimal tour with an approximate one using Christofides' heuristic. Although this reduces computational complexity, it results in a higher competitive ratio.

In this thesis (Chapter 3), together with Christofedes, we also study GENI as a scheduling strategy for the PAH algorithm.

2.3.2 Hamiltonian Path Problem

In contrast, when the server does not need to return to a fixed point to schedule, Hamiltonian path-based scheduling becomes more appropriate. This approach is used in N-OLTSP. Ausiello et al. [3] introduced the GTR algorithm with Hamiltonian path scheduling strategy for solving N-OLTSP. GTR computes a shortest Hamiltonian path that covers all unserved requests starting from the server's current location. However, the exact computation of such a path can be done using dynamic programming, but this approach is exponential in time. The authors proposed a heuristic variant using an MST approximation to address this. The MST provides a simple structure that approximates the request layout, and it can be transformed into a path by traversal and shortcutting. This approach maintains efficiency while

keeping the solution cost within acceptable bounds. The authors [3] proved that a 2-approximate Hamiltonian path can be constructed between any two specified endpoints using the MST heuristic (details about approximation ratio is in Appendix B.2). Based on this result, the GTR algorithm can be extended to solve the H-OLTSP by computing a Hamiltonian path from the server's current location to the origin.

Although the GTR algorithm has a weaker theoretical guarantee than PAH regarding competitive ratio, it has a much simpler design. Unlike PAH, which forces the server to return to the origin every time it needs to recompute the route, GTR algorithm avoids such unnecessary movements. This characteristic suggests that GTR might perform better in practice, especially under typical request patterns where frequent returns to the origin are inefficient. Therefore, *in this thesis, we evaluate the practical performance of GTR under different scheduling strategies* (Chapter 7) and compare it to PAH (Chapter 8).

2.3.3 Other Works

In addition to classical heuristics, which are problem-specific strategies designed to quickly generate good solutions using domain knowledge, various metaheuristic methods have been developed to solve the TSP. Metaheuristics differ from heuristics in that they are general-purpose frameworks that guide the search process and are applicable to a wide range of combinatorial optimization problems. Common

examples include tabu search [15], genetic algorithms [16], and simulated annealing [17]. These approaches are particularly effective in large-scale or time-sensitive applications due to their ability to escape local optima and explore complex solution spaces.

Since metaheuristics are designed for general combinatorial optimization, they can also be applied to the Hamiltonian path problem with fixed start and end points (Appendix B, Definition 12). This variant, which requires finding a path that begins and ends at specified nodes, is relevant in the GTR algorithm used for N-OLTSP. To the best of our knowledge, no metaheuristic method has been developed specifically for the Hamiltonian path problem with fixed endpoints as used in GTR. However, because of the close relationship to the general Hamiltonian path problem, existing metaheuristic techniques can be adapted with minimal modification. Although metaheuristics are not the main focus of this thesis, they represent a broader class of solution strategies and suggest possible future directions for research on online scheduling algorithms.

Chapter 3

PAH Algorithms

In this chapter, we present the PAH algorithm evaluated in this thesis. The formal definitions and operational logic of the PAH algorithm are provided in Section 3.1. Following the definition, we describe the scheduling strategies used in conjunction with the PAH algorithm. Section 3.2 outlines the scheduling methods applied to PAH. These strategies play a critical role in determining how the algorithms construct routes in response to dynamically arriving requests and are essential for understanding the evaluation results presented in later chapters.

3.1 Description of PAH

In the PAH algorithm, the server always schedules its route at the origin. Several special cases may arise during the algorithm's execution. These cases are described

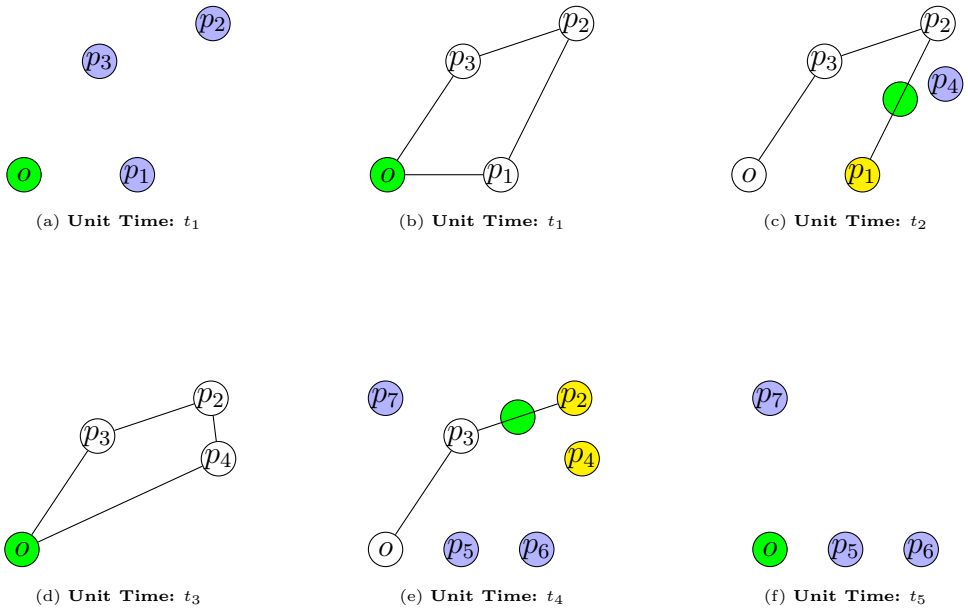
in the Definition 2 based on Ausiello et al. [3] proposed.

Definition 2. *PAH*

- ***Case 1*** *The server is at the origin:*
  - *To schedule or reschedule a route that visits all outstanding requests and returns to origin.*
  - *The server starts moving on the new route.*
- ***Case 2*** *If the server is not at the origin and new requests are released, the server will decide whether or not to return to the origin to reschedule based on its location and the location of new requests.*
  - ***Case 2.1*** *All newly released requests are positioned at most as far from the origin as the server's current location:*
    - *These new requests will be ignored, and the server continues moving on its current route. The ignored requests will be scheduled after the current route is completed (re-enter case 1).*
  - ***Case 2.2*** *Otherwise, the server will return to the origin. (Enter case 1)*

3.1.1 An example of the PAH Algorithm in Execution

Based on the definition, we illustrate the different cases through an example, Figure 3.1. The server operates in a metric space that is defined by a list of points



(a) **Unit Time:** t_1 (b) **Unit Time:** t_1 (c) **Unit Time:** t_2

(d) **Unit Time:** t_3 (e) **Unit Time:** t_4 (f) **Unit Time:** t_5

Figure 3.1: An Example of Execution of PAH

and a distance function between every pair of points. t is time that measured in unit steps. In each unit step, the server can move a distance of one. Each unit step counts toward the total completion time, which is the objective of H-OLTSP. In this example, we assume $t_1 < t_2 < t_3 < t_4 < t_5$. The green marker indicates the current location of the server. The yellow markers represent points the server has already visited. The purple markers denote *newly* released points the server needs to visit.

Initially (Figure 3.1 (a)), the server is at the origin (o) at the time t_1. Three new requests (p_1, p_2, and p_3) are released at t_1: That is, (p_1, t_1), (p_2, t_1), and (p_3, t_1). Since the server is at the origin, we apply Case 1.

The server schedules a TSP tour that will visit all outstanding requests and return to the origin. This is shown in Figure 3.1(b). The server moves on the new route.

At time t_2, assume the server has completed the request (p_1, t_1) (marked yellow) and is moving on the path between p_1 and p_2. Assume, during its travel, a new request $((p_4, t_2))$ is released (Figure 3.1 (c)). Since the server is not at the origin, and new requests are released during its traversal, Case 2 applies. The server decides whether or not to return to the origin to reschedule based on its location and the location of the new request (p_4, marked in purple). Assume that the server is closer to the origin than the location of the new request. That is, $d(p(s), o) < d(p_4, o)$, where $p(s)$ is the position of the server and $d()$ is the distance function that calculates the distance between two input points. Then, Case 2.2 applies. The server will return to the origin and return to Case 1 (Figure 3.1 (d)). At time t_3, the server schedules a TSP tour to visit all uncompleted requests and travels on this tour to complete the scheduled requests.

Assume the server completes requests p_4 and p_2 (marked as yellow) and moves from p_2 to p_3 at t_4 (Figure 3.1 (e)). Let's assume three new requests (p_5, t_4), (p_6, t_4), and (p_7, t_4) are released at t_4. Since the server is not at the origin, Case 2 applies. The server needs to determine whether to reschedule based on the location of these three new requests.

Based on the locations (Figure 3.1 (e)), let's assume, we know that $d(p(s), o) > d(p_5, o)$, $d(p(s), o) > d(p_6, o)$, and $d(p(s), o) > d(p_7, o)$. Since the locations of all new requests are closer to the origin than the server is to the origin, Case 2.1 applies. The server will ignore the new requests and continue to move on its current route. The

ignored requests will be scheduled after the current route when the server reaches the origin, re-entering Case 1 (Figure 3.1 (f)) at time t_5.

3.2 Scheduling Strategies for PAH

3.2.1 Scheduling Problem

Since the PAH algorithm requires the server to return to the origin before scheduling or rescheduling, it naturally treats the scheduling step as a Traveling Salesperson Problem (TSP). Specifically, whenever the server needs to compute a new route, it calculates a TSP tour that visits all currently unserved requests and returns to the origin. To describe this in detail, we use the example in Figure 3.2.

In the example, the location of the server is marked as green. We assume a H-OLTSP instance: $\{< M, d >, o, Q = ((p_1, t_1), (p_2, t_1), (p_3, t_1))\}$ where $< M, d >$ is the metric space, o is the origin, Q is a list of pairs of location and release time. In this instance, points (p_1, p_2, p_3) are released at time t_1 (Figure 3.2 (a)). It is important to note that $p_1, p_2, p_3, o \in M$.

At time t_1 the server is at the origin o (Figure 3.2 (b)). The PAH algorithm schedules a route that visits all points (p_1, p_2, p_2) and returns to the origin. That is, it finds a TSP tour. In this case, the PAH algorithm first transfers the route scheduling to a TSP instance, then uses a solution of such instance as a route. In detail, it first makes the graph by points (p_1, p_2, p_3, o). That is Given $G = (V, E)$

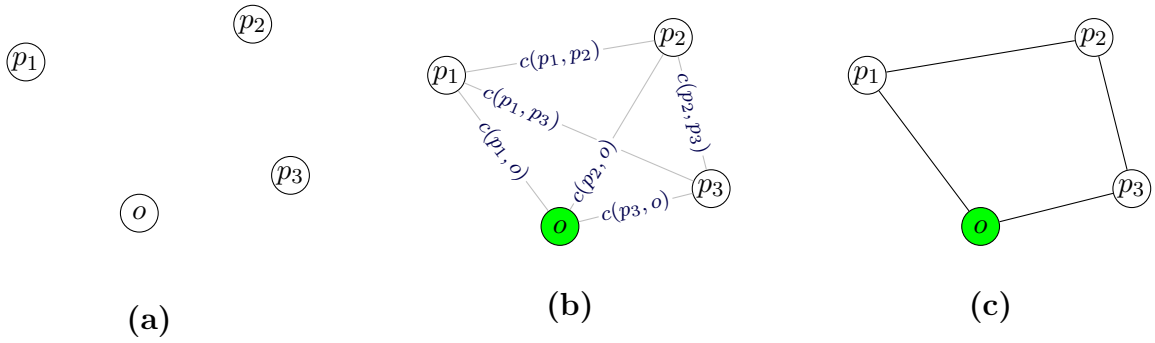


Figure 3.2: Examples of Scheduling Problems in PAH

where $V = \{p_1, p_2, p_3, o\}$ and $E = V \times V$, the algorithm builds the cost function c from the distance function $< M, d >$. That is $\forall u, v \in V : c(u, v) = d(u, v)$. The algorithm then constructs the TSP instance (G, c) as shown in Figure 3.2 (b) to find a route. The PAH algorithm solves the instances to find a TSP tour (or route) with a TSP algorithm such as Christofides, for example. An example of a route is shown in Figure 3.2 (c).

3.2.2 Choice of Route Direction

Since a TSP tour is a cycle, the server starts at the origin and can begin the tour in either of two possible directions, each leading to the same set of visits. For example, in the case shown in Figure 3.2 (c), the server may first visit p_1 or p_2. In the paper proposing PAH [3], the authors did not specify how this choice should be made. In our implementation, we resolve this ambiguity by choosing the direction randomly: with equal probability the server follows one direction or the other. This

avoids bias and provides a neutral strategy when no additional information favors one side.

3.2.3 Scheduling Algorithms Studied

The choice of scheduling algorithm significantly impacts the competitive ratio of the PAH algorithm. Moreover, route planning and scheduling these routes is important for real-time applications to improve performance. The authors [3] in studying PAH's competitive analysis considered scheduling algorithms that run in polynomial time, such as the Christofides algorithm.

This thesis tries to understand how PAH performs when combined with different scheduling algorithms in practical settings. Specifically, we aim to investigate the real-world effectiveness of PAH and various scheduling algorithms. Although competitive analysis offers a theoretical baseline by assessing worst-case performance, it does not necessarily capture how these algorithms behave in dynamic and realistic environments.

Since there are many existing algorithms for solving the TSP, this thesis focuses only on those that are directly related to PAH's theoretical analysis. *We consider two scheduling algorithms: an exact algorithm using dynamic programming and the Christofides algorithm.* These algorithms are chosen because their combination with PAH has been formally analyzed in the original theoretical work on H-OLTSP. Moreover, using these algorithms also gives us the opportunity to compare their theoreti-

cal results with their practical performance in simulated environments. The dynamic programming method is selected because it produces optimal TSP solutions with reasonable computational complexity for small to medium-sized instances. Christofides algorithm is included because it offers a polynomial-time solution with a known approximation ratio, making it suitable for evaluating efficiency under theoretical bounds. Our goal is to test whether these theoretical combinations also perform well in practice.

To go beyond the limitations of these two algorithms, *we also include the GENI algorithm.* GENI introduces features not supported by the others, such as controllable randomness, neighbourhood-based decision-making, and the ability to reschedule routes. These features are valuable in online and dynamic settings where adaptability can impact overall performance. By comparing PAH combined with these different scheduling strategies, we aim to evaluate not only theoretical alignment but also practical effectiveness in more realistic environments.

Next, we describe the three scheduling algorithms (dynamic programming, GENI and Christofides) used in this evaluation. Each algorithm is applied within the PAH to compute a route for the current set of released requests. Throughout these descriptions, we assume that the scheduling problem is a TSP defined on a metric graph $G = (V, E)$, where distances or costs satisfy the triangle inequality.

I. Dynamic Programming

In this thesis, we use the Dynamic Programming (DP) that is proposed by Bellman [18] for TSP. Since the DP technique is well known, the technique is explained in Appendix A.1.

II. Christofides Heuristic

The authors [3] proposed PAH with Christofides heuristic, providing a polynomial algorithm for H-OLTSP. The Christofides' heuristic [12] computes approximate solutions for the metric TSP. The algorithm is $\frac{3}{2}$-approximation and runs in $O(n^3)$ time, where $n = |V|$. To describe the algorithm, we assume a TSP instance (G, c) where $G = (V, E)$ and a cost function $c : e \mapsto \mathbb{Z}^+$ for all $e \in E$. The algorithm takes the TSP instance as input, then it applies the following steps:

- **Step 1** Compute a MST (Appendix B, Definition 14) of the input graph G, which connects all vertices with the minimum total cost.
- **Step 2** Get a set $V_{odd} \subset V$ of vertices that have an odd degree in the MST.
- **Step 3** Compute a minimum cost perfect matching (Appendix B, Definition 13) $E_{matching} \subset E$ among the vertices in V_{odd}, using the cost function c.
- **Step 4** Merge the edges from the MST and $E_{matching}$ to create a multi-graph G_{sub} such that all vertices have even degree.

- **Step 5** Get an Eulerian circuit (Appendix B, Definition 15) C_e in G_{sub}, which is a closed path that visits each edge of G_{sub} exactly once.

- **Step 6** Convert the Eulerian circuit C_e into a TSP tour by shortcutting previously visited vertices while preserving the visitation order.

A detailed example of Christofides Heuristic is provided in Appendix C.2.

III. GENI Algorithm

In addition to these two scheduling algorithms that have theoretical results with PAH that the authors [3] showed in the original article, we also evaluated the performance of combining PAH with the GENI algorithm [14] for solving the TSP.

In this section, we present the main process of the algorithm to provide a high-level understanding of its core logic. This description focuses on the primary flow and does not include all implementation details. Due to the complexity and technical nature of the algorithm, a complete step-by-step explanation in the main text is not sufficient and may lead to confusion. Therefore, a detailed pseudocode is provided in Appendix A.2. It offers a precise and language-independent representation that more effectively captures the full structure and logic of the algorithm. For specific mechanisms such as the handling of Type I and Type II insertions, please refer to the pseudocode in Appendix A.2.

The GENI algorithm randomly generates a tour that visits three vertices, and then the remaining vertices are inserted into this initial tour by two insertion algo-

rithms. These two insertion algorithms can insert a vertex between two discontinuous vertices. In each iteration of insertion, GENI uses both insertion algorithms (Type I and Type II insertions) and selects the result with a lower cost. In detail, the main steps of GENI are described as follows:

1. **Initialization:** Begin with an initial tour of three arbitrary vertices.

2. **Neighborhood Definition:** For each vertex v not yet in the tour, define its p-nearest neighbors among the vertices currently in the tour. This set is denoted $N_p(v)$ and is used to restrict the set of candidate insertions.

3. **Vertex Selection:** Select a vertex v not yet inserted.

4. **Insertion Evaluation:** For the selected vertex v, evaluate multiple possible ways to insert it into the tour. Two types of generalized insertions are considered:

   - *Type I:* Insert v in such a way that it connects to two existing vertices in the tour (not necessarily adjacent), and a segment between them is reversed to maintain a valid tour.
   - *Type II:* Insert v between two non-adjacent vertices by reconnecting multiple segments of the tour and reversing certain portions if needed. This allows greater flexibility in reshaping the tour during insertion.

   Only combinations where the involved vertices belong to the respective p-neighborhoods are considered to reduce computation.

5. **Insertion Execution:** Among all feasible insertions (of both types, and for both orientations of the tour), select the one that results in the minimum total tour cost. Insert v accordingly.

6. **Update:** Update the tour structure and the p-neighborhoods to reflect the addition of v.

7. **Repeat:** Return to Step 3 until all vertices are part of the tour.

This algorithm has a time complexity of $O(np^4 + n^2)$ where n is the number of vertices and p is the number of neighborhoods. The GENI algorithm implemented in this study is based on the description provided by Gendreau et al. [14]. *However, due to the lack of detailed implementation details in the original paper, certain aspects of the algorithm were inferred or adapted during implementation.* The resulting interpretation has been presented in pseudocode form to provide clarity and transparency in the Appendix A.2. Although we have made every effort to remain faithful to the original author's intent and approach, our implementation may differ from the original in some respects. The results and conclusions presented in this paper are based on this interpretation and should be considered with this potential variation in mind.

Variants of the GENI Algorithm Since the GENI algorithm is highly flexible in terms of configuration, we evaluate two main types of GENI variants in this thesis. The first group of variants explores the effect of different neighborhood sizes, which

directly influence both the solution quality and computational cost. Specifically, we consider the following:

- GENI-3 where the value indicates that only the 3 nearest neighbors (in terms of distance or cost) are considered when evaluating possible insertions. These values are commonly used in the literature and strike a balance between performance and speed.

- GENI-N, where all current tour vertices are treated as neighbors (i.e., the full neighborhood is used). This configuration represents the upper bound of GENI's search capability, allowing for the most thorough insertion evaluation. Although this provides potentially better solutions, it also results in the highest computational complexity.

To better understand the performance of GENI limits, we include GENI-N in our experiments despite its heavy runtime cost. However, its behavior becomes similar to PAH algorithm's scheduling process, which reschedules a new tour to cover a new request.

Therefore, we introduce a practical modification. In our modified version of GENI-N, instead of recomputing the entire tour from scratch with every unversed request, the algorithm leverages the current route. It attempts to insert the new request using GENI's generalized insertion logic. This approach preserves the core behavior of GENI-N while significantly reducing its computational overhead by rescheduling

its ability. *We call this algorithm GENI-N-RS, which means GENI algorithm with N neighborhood and Rescheduling ability.* We describe the modification in an example of a scheduling scenario in PAH in Figure 3.3.

Example The input to the GENI algorithm is a TSP instance, (G, c). The first step is to generate a 3 nodes tour from G, then it iteratively inserts the remaining nodes into the tour so that it can cover all nodes. Our modification is to remove the initialization step and give the GENI algorithm an initialized tour. Then it inserts the remaining nodes into it. We describe this modification in detail with the PAH scheduling. We assume that the server is moved via the old scheduled TSP tour.

The scheduling scenario is illustrated in Figure 3.3 (a). The server, shown as a green dot, is traveling from point p_1 to p_2, and has just visited p_1. Visited nodes are marked in yellow. At this moment, three new requests have been released, as shown in purple. In this case, the PAH algorithm will instruct the server to return to the origin for rescheduling. This is because the point p_5 is farther from the origin than the current position of the server. As a result, the server goes back to the origin. When it arrives at the origin (as shown in Figure 3.3(b)), it must compute a TSP tour that includes all unserved requests.

In the standard GENI algorithm, the TSP tour starts with three randomly chosen nodes and then inserts the remaining nodes one by one. We assume that the points o, p_4, and p_7 are picked as the initialized tour, which is shown in Figure 3.3 (c). In our modified version, we use the previous partial tour T_{old} as the initial tour.

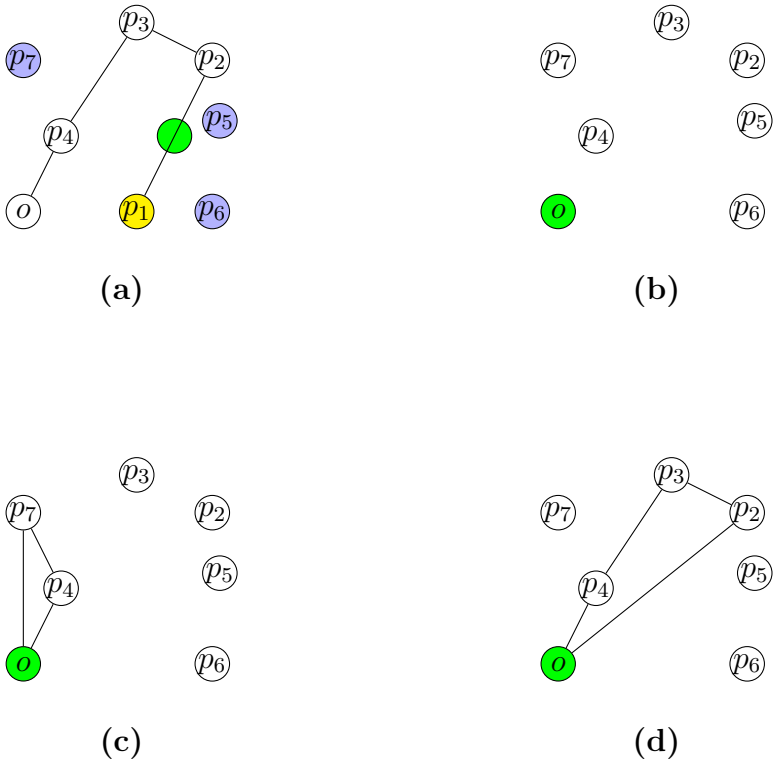


Figure 3.3: An Example of Initialization in GENI and GENI-RS

Specifically, we connect the origin to the next unvisited node to form an initial tour. We assume that V_{visted} are nodes that the server visited from T_{old}. In this case, $T_{visited} = [p_1]$ and $T_{old} = [p_2, p_3, p_4, o]$, the initial should be the origin o connected to the next unvisited node of T_{old}. This modified initialized tour is $[o, p_2, p_3, p_4, o]$ which is shown in Figure 3.3 (d). Then, the remaining nodes are inserted into the tour.

3.3 Summary

This chapter presented the PAH algorithm for H-OLTSP along with three scheduling strategies used to compute TSP tours during each rescheduling event: dynamic programming, Christofides, and the GENI algorithm. The PAH algorithm was described in detail, including how it updates the server's planned route in response to new requests while ensuring a return to the origin.

Each of the three scheduling algorithms was then explained. The dynamic programming method computes optimal TSP tours achieving minimum cost at the expense of high computational time. The Christofides algorithm constructs a $\frac{3}{2}$-approximate tour using minimum spanning trees and perfect matchings, offering a good balance between cost and runtime. The GENI algorithm incrementally builds a tour through Type-I and Type-II insertions, allowing it to efficiently adapt to newly added requests with moderate computational effort. Based on this characteristic, we introduce GENI-N, which extends GENI with a dynamic neighborhood mechanism, and GENI-N-RS, which further adds rescheduling capability to GENI-N for improved responsiveness.

These scheduling algorithms offer different tradeoffs between tour quality and runtime, which directly influence the responsiveness and efficiency of PAH. In Chapter 6, we evaluate PAH combined with each scheduling algorithm and compare their performance under varying request-release scenarios. The comparison focuses on two key aspects: responsiveness (how quickly the algorithm reacts to new requests)

and efficiency (the completion time that the server visits all requests and return to origin).

Chapter 4

GTR Algorithm

In this chapter, we present the GTR algorithm evaluated in this thesis. The formal definitions and operational logic of each algorithm are provided in Section 4.1. Following the definition, we describe the scheduling strategies used in GTR algorithm in Section 4.2.

4.1 Description of GTR Algorithm

The GTR algorithm guides the server to schedule or reschedule a route every time a new request is released. The GTR algorithm was originally introduced for the N-OLTSP. In their paper, Ausiello et al. [3] stated that the same greedy idea can also be applied to the H-OLTSP by modifying the route so that it ends at the origin. However, they did not provide a formal definition or pseudocode for this H-OLTSP

version. To address this gap, we present an explicit definition of GTR for H-OLTSP in this thesis, based on the descriptions and proofs given for N-OLTSP. We have tried to remain as close as possible to the original intent, but since the H-OLTSP variant was never formally specified, our definition may differ slightly in detail. For clarity and reproducibility, the precise version of GTR used in our experiments is provided Definition 3.

Definition 3. *GTR*

Case 1: No new requests at this time

- ***Case 1.1:*** *If there is a scheduled route:*
  - *The server continues moving along the current scheduled route.*
- ***Case 1.2:*** *If no route is scheduled:*
  - *The server remains idle.*

Case 2: New requests released

- ***Case 2.1:*** *The server is at a point (node)*
  - *Compute a shortest route from the current location that visits all unserved requests and return to origin at end.*
  - *Begin following this route immediately.*
- ***Case 2.2:*** *The server is on a path between two points x and y*

- ***Case 2.2-a:*** *Each of x and y is either a requested point or the origin.*

  $\Rightarrow$ *Follow the shortest route that first* ***visits either*** x ***or*** y***, then the remaining unserved requests and return to origin.***

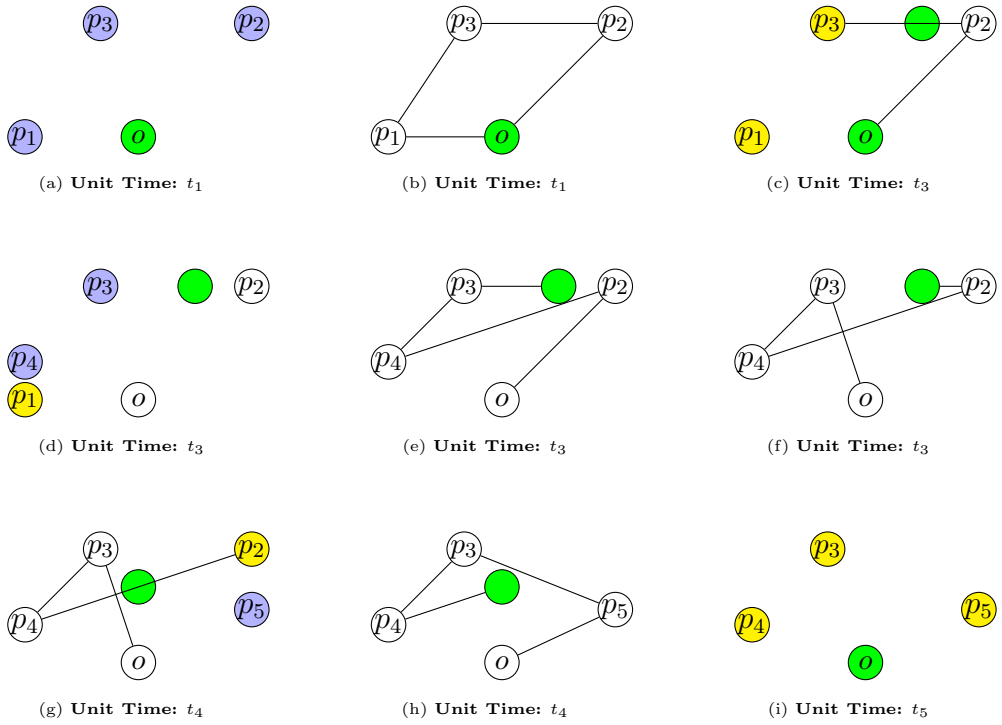
- ***Case 2.2-b:*** *One of the two points, x or y, belongs to the set of requested points or is the origin*

  $\Rightarrow$ *Follow the shortest route that* ***visits the requested point first, then the remaining unserved requests and return to origin.***

4.1.1 An example of the GTR Algorithm in Execution

Based on the definition, we illustrate the different cases through an example (Figure 4.1). We assume that the server operates on the metric space and unit time $t_1 < t_2 < t_3 < t_4 < t_5$. The green marker indicates the current location of the server. The yellow markers represent points the server has already visited. The purple markers denote *newly* released points that the server needs to visit.

Initially (Figure 4.1 (a)), the server is at the origin (o) at the time t_1. Three new requests (p_1, p_2, and p_3) are released at t_1: That is, (p_1, t_1), (p_2, t_1), and (p_3, t_1). Since the server is at the origin, a point or node, we apply Case 2.1. The server schedules a Hamiltonian path that starts at the current location o, visits all unserved requests (p_1, p_2, p_3), and returns to the origin o. An example of such a route is shown in Figure 4.1 (b). As the route computation time is not included in the server's completion time, in other words, the computation does not affect the objective value



(a) **Unit Time:** t_1 (b) **Unit Time:** t_1 (c) **Unit Time:** t_3

(d) **Unit Time:** t_3 (e) **Unit Time:** t_3 (f) **Unit Time:** t_3

(g) **Unit Time:** t_4 (h) **Unit Time:** t_4 (i) **Unit Time:** t_5

Figure 4.1: An Example of GTR

of H-OLTSP, both the releasing of requests and the computing of the Hamiltonian path are done at time t_1. Hence, the server starts to move on the route at t_1.

At time t_2, we assume that the server is moving along the scheduled route and no new requests are released. In this case, we apply Case 1.1, so the server continues to follow the current route. At time t_3, we assume the server has already completed requests (p_1, t_1) and (p_3, t_1) (highlighted in yellow) and is now moving on the path between p_3 and p_2 (Figure 4.1(c)). At this moment, two new requests (p_4, t_3) and (p_3, t_3) are released (Figure 4.1(d)). Since the server is currently between two unserved points, we apply Case 2.2-a. The server computes two possible Hamiltonian

paths (shown in Figure 4.1(e) and Figure 4.1(f)). It then chooses the shorter one and starts moving along that route. We assume that the server begins to follow the path shown in Figure 4.1(f).

At the time t_4, we assume that the server is still moving along the current route. It has already visited p_2 and is now traveling from p_2 to p_4. At this moment, a new request (p_5, t_4) is released. In this case, the server is between two points, but only one of them (p_4) is an unserved request. Therefore, we apply Case 2.1. The server computes a Hamiltonian path that starts by visiting p_4, then visits the remaining unserved requests (p_3 and p_5), and finally returns to the origin. The new route is shown in Figure 4.1(h). As a result, the server begins moving along the new route at time t_4.

We assume that no new requests are released, and the server finishes the current route at time t_5. Therefore, for all time units before t_5, Case 1.1 applies, meaning the server continues moving along the planned route. At time t_5, the server has completed the route and is located at the origin (Figure 4.1(i)). Since there are no pending requests, Case 1.2 applies, so the server stays at the origin. In this situation, although all requests have been served, the server does not know this in advance. The algorithm ensures that the server is at the origin once all requests are completed.

4.2 Scheduling Strategies for GTR

4.2.1 Scheduling Problem

In the GTR algorithm for the H-OLTSP, the server updates its route immediately whenever a new request is released. Each time scheduling or rescheduling is triggered, the *algorithm formulates the problem as a Hamiltonian path problem with fixed endpoints*(Appendix B, Definition 12). Specifically, it constructs a graph $G = (V, E)$, where the vertex set V contains all unserved requests along with the origin. The edge set E includes all possible pairs of vertices in V. The cost function c is defined using the distance function from the metric space of the H-OLTSP. That is, for every $v_1, v_2 \in V$, the cost is given by $c(v_1, v_2) = d(v_1, v_2)$. Using this graph and cost function, the algorithm defines a Hamiltonian path problem instance (G, c, v_s, v_t), where v_s and v_t are the designated start and end points. The server solves this instance and uses the resulting Hamiltonian path as the new route to follow.

However, as seen from the definition, different cases determine how scheduling is performed. Although all cases use the Hamiltonian path problem with fixed endpoints as the core scheduling problem, they apply it in different ways depending on the situation.

Discussion: Scheduling Cases In Case 2.1 of the GTR algorithm (Definition 3), a new request is released while the server is located at a specific point in the metric space. In this case, we assume that the constructed graph and cost function are given

by (G, c). The algorithm computes a Hamiltonian path $HP_{v_s \to v_t}(G, c)$, where v_s is the current location of the server. This location corresponds to a vertex in the graph, since the server is positioned at a specific node or point. Hence, the $HP_{v_s \to v_t}(G, c)$ is the new route to move on in this case.

In Case 2.2 of the GTR algorithm (Definition 3), a new request is released while the server is in transit between two locations in the metric space. Let these two points be x and y, and assume the server is moving from x towards y. The algorithm considers two sub-cases based on whether x and y are unserved requests.

In Case 2.2-a, for points x and y that each of x and y is either a requested point or the origin. The server is currently located somewhere between these two points, denoted as $p(s)$. The algorithm constructs two possible Hamiltonian paths: $HP_{p(s),x \to v_o}(G, c)$ and $HP_{p(s),y \to v_o}(G, c)$, where v_o is the origin. Each of these paths starts from the current location $p(s)$, first visits either x or y, then visits all other unserved requests exactly once, and ends at the origin v_o. The server computes the cost of both paths and selects the shorter one as the new route to follow.

In Case 2.2-b, only one of the two points, either x or y, is an unserved request or the origin. In this situation, only one Hamiltonian path needs to be computed. Without loss of generality, assume that y is the unserved request or origin and x is not. Then the algorithm constructs the path $HP_{p(s),y \to v_o}(G, c)$, which starts at the server's current location $p(s)$, goes to y, visits all remaining unserved requests exactly once, and ends at the origin. The server then begins moving along this route.

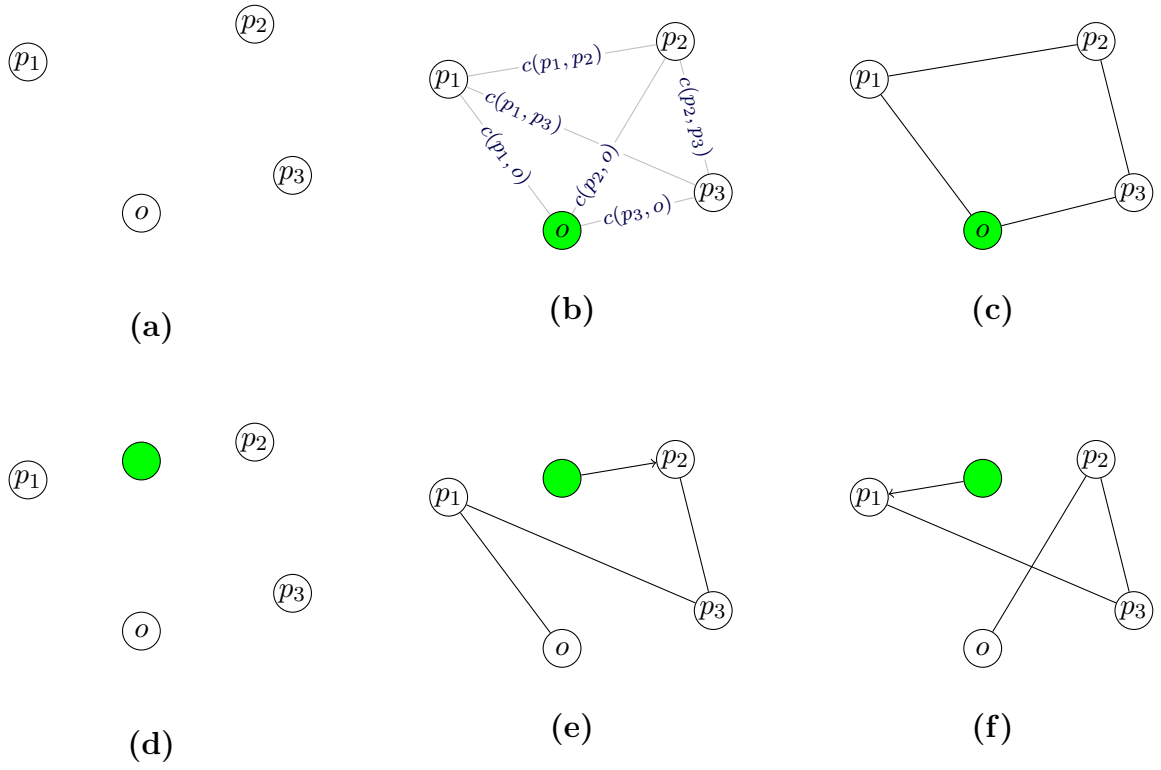


Figure 4.2: Examples of Scheduling Problem in GTR

We illustrate the explanation above with an example. In the Figure 4.2, the server's location is marked in green. Consider an H-OLTSP instance defined as $\langle M, d\rangle, o, Q = ((p_1, 1), (p_2, 1), (p_3, 1), (p_1, 5))$, where $\langle M, d\rangle$ is the metric space, o is the origin, and Q is the list of requests. Each request is a pair consisting of a location and its release time. In this instance, the points p_1, p_2, and p_3 are released at time 1, and an additional request at p_1 is released at time 5. It is important to note that p_1, p_2, p_3, and o are all elements of the metric space M.

At time 1, the server is located at the origin o. The GTR algorithm needs to

schedule a route that visits all requests at points p_1, p_2, and p_3, and then returns to the origin. This is illustrated in Figure 4.2(a).

To do this, the GTR algorithm first converts the scheduling task into an instance of the Hamiltonian path problem with fixed endpoints. Specifically, it constructs a graph $G = (V, E)$ where the vertex set is $V = p_1, p_2, p_3, o$ and the edge set is $E = V \times V$, which includes all pairwise connections. The cost function c is defined using the distance function d from the metric space $\langle M, d \rangle$. That is, for all $u, v \in V$, the cost is $c(u, v) = d(u, v)$.

Then, the algorithm creates the Hamiltonian path problem instance (G, c, o, o), where both the start and end points are the origin. This instance is shown in Figure 4.2(b). Since the server is located at the origin, which is a specific point in the metric space, this falls under Case 2.1 of the GTR algorithm. The server computes a *feasible* solution to the instance, denoted by $HP_{o \to o}(G, c)$, and uses it as the route to follow. The resulting path is illustrated in Figure 4.2(c).

At time 5, we assume the server has completed the request at p_1 and is currently moving toward p_2. At this moment, a new request at p_1 is released. This situation corresponds to Case 2.2 of the GTR algorithm. Since the server is between two points, and both p_1 and p_2 are unserved requests, the server must compute two possible Hamiltonian paths: $HP_{p(s), p_1 \to o}(G, c)$ and $HP_{p(s), p_2 \to o}(G, c)$, where $p(s)$ is the current location of the server, and o is the origin. These two paths represent the possible routes starting from the server's current position, visiting either p_1 or

p_2 first, then all remaining unserved requests exactly once, and ending at the origin. The two computed paths are shown in Figure 4.2(e) and Figure 4.2(f).

4.2.2 Scheduling Algorithms Studied

In this section, we describe the scheduling strategies used for the GTR algorithm. The authors [3] proposed that GTR can be implemented using either an optimal or an approximate Hamiltonian path. However, they did not specify which exact algorithm should be used to compute the optimal path. In our implementation, we use a dynamic programming approach to obtain the optimal Hamiltonian path with fixed endpoints. The authors proposed using the MST heuristic, which we also study for the approximate case. In addition to this, we introduce a modified version of the GENI algorithm adapted to compute Hamiltonian paths. The details of this modification are also presented in this section. The motivation for incorporating GENI into GTR is similar to that for its use with PAH. (described in Section 3.2). The MST heuristic, dynamic programming, and GENI for computing Hamiltonian paths are described next.

In the following descriptions of scheduling algorithms, we assume that Hamiltonian path with fixed endpoints problem instances are defined on complete, weighted, metric graphs, such graph $G = (V, E)$, where V is a set of n vertices and E is the set of edges. The distance function $d : V \times V \rightarrow \mathbb{Z}^+$ assigns a positive integer weight to each edge that satisfies the triangle inequality. Moreover, we assume that the

vertices $v_o, v_s \in V$ where v_o is the designated origin and v_s is the start vertex. The objective is to find a minimum-cost path that starts at the vertex v_s, visits the other vertex in $V \setminus \{v_s\}$ exactly once, and visits v_0 at end.

I. MST Heuristic

The MST heuristic offers a simple and efficient way to compute an approximate solution to the Hamiltonian path problem in metric spaces. It is particularly useful when exact methods are too computationally expensive for real-time applications. We describe this based on the textbook [19] and Ausiello et al. [3] described. The steps of MST Heuristic are described as follows:

1. **Construct a MST:** Build an MST T on the set V.
2. **Double Every Edge :** Add an edge between any pair of two vertices of the MST to make a multi-graph G_d.
3. **Find an Eulerian Circuit:** Traverse the multi-graph G_d to find an Eulerian Circuit C_e.
4. **Add an edge** (v_s, v_o)**:** Add an edge (v_s, v_o) by replacing the shortest path from v_s to v_o in C_e
5. **Generate the TSP tour:** Convert the C_e into a TSP tour by shortcutting repeated vertices.

6. **Convert into Hamiltonian Path:** Delete the edge (v_s, v_o) to get a Hamiltonian path that with endpoints v_s and v_o.

II. Dynamic Programming

For computing the optimal solution of the Hamiltonian path problem with fixed endpoints, we also use the dynamic programming introduced by Bellman [18]. We specify some steps of the dynamic programming of TSP to solve the Hamiltonian path with fixed endpoints. The general process remains the same, but instead of starting from a fixed origin, the starting point is flexible. The algorithm still ensures that all locations are visited once, and the path ends at a designated endpoint.

III. GENI for GTR

The GENI algorithm that is proposed by Gendreau et al. [14] was originally developed to solve the TSP, where the goal is to find the shortest possible tour that visits every node exactly once. A small change is made to its initialization and insertion steps to adapt GENI for the Hamiltonian path problem with fixed start and end points.

First, the algorithm starts with a three-node tour that includes the designated start point, the fixed endpoint, and one arbitrary node chosen from the remaining set. This forms the initial structure for the tour.

During the insertion process, the edge connecting the start and end points is kept fixed. The GENI algorithm is not allowed to modify or remove this edge while

inserting the other nodes. Each new node is added using Type-I or Type-II insertion procedures, and the start-end connection is preserved throughout.

After all nodes are inserted, the resulting tour contains the start and end points as two adjacent nodes. To turn this tour into a valid Hamiltonian path, the edge between the start and end points is removed. The final path begins at the start point, ends at the endpoint, and visits all other nodes exactly once.

Example We show an example in Figure 4.3 to illustrate the explanation above. Suppose we are given a Hamiltonian path with fixed endpoints problem instance (G, c, v_s, v_t), which is shown in Figure 4.3. To better highlight the key idea, we assume that in each step, the best insertion always occurs between the two closest nodes.

The initial tour includes the nodes v_s, v_t, and v_1, where the edge between v_s and v_t is fixed. This fixed edge is marked in green in Figure 4.3 (b). Next, the algorithm inserts v_2 into the current tour (Figure 4.3 (c)). Due to the fixed edge, v_2 can only be inserted between v_s and v_1 or between v_1 and v_t. We assume that the better insertion is between v_s and v_1, which results in the tour shown in Figure 4.3 (d).

Then, v_3 is inserted using the same approach, and the resulting tour is shown in Figure 4.3 (e). After the GENI algorithm produces a TSP tour, we remove the fixed edge between v_s and v_t. This produces a Hamiltonian path that starts at v_s and ends at v_t, as shown in Figure 4.3 (f).

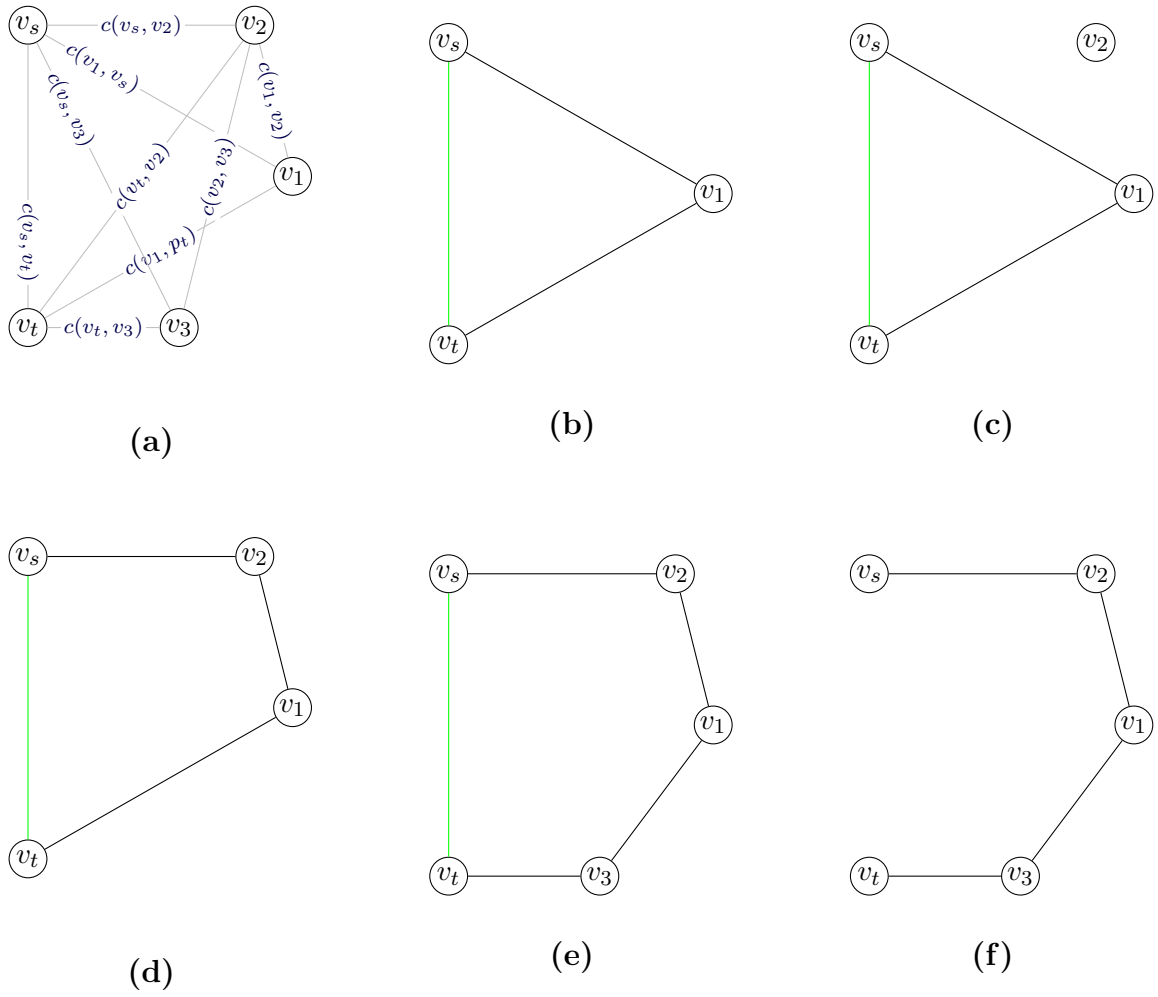


Figure 4.3: Examples of GENI for GTR

4.3 Summary

This chapter presented the GTR algorithm for H-OLTSP along with three scheduling strategies used to compute Hamiltonian paths with fixed endpoints after each request release: dynamic programming, MST-based approximation, and the GENI

algorithm. The GTR algorithm was described in detail, including how it greedily selects a direction based on the nearest unvisited extreme and immediately computes a Hamiltonian path starting from the server's current location to cover all unserved requests.

Each of the three scheduling algorithms was then explained. The dynamic programming method generates optimal Hamiltonian paths, ensuring minimum path cost but with high computational time. The MST-based algorithm offers a fast 2-approximation by building a minimum spanning tree and applying shortcutting from the current position to the furthest unserved request. The GENI algorithm was modified to compute Hamiltonian paths with fixed endpoints. This modified version also incrementally builds a path and efficiently adapts to newly added requests with moderate computational effort. Based on this adaptability, we also introduce GENI-N, which uses a dynamic neighborhood mechanism, and GENI-N-RS, which further adds rescheduling capability to GENI-N.

These scheduling strategies offer different tradeoffs between path quality and runtime, which directly influence the responsiveness and efficiency of GTR. In Chapter 7, we evaluate GTR with each of these scheduling algorithms and compare their performance under varying request-release scenarios. The evaluation focuses on two key metrics: responsiveness, measured by how quickly the algorithm reacts to new requests, and efficiency, measured by the total travel cost to complete all requests and return home.

Chapter 5

Evaluation Framework

This chapter provides the foundation for all empirical results presented in later chapters. We develop a structured evaluation framework to evaluate the practical performance of algorithms for the H-OLTSP. The goal of this framework is to simulate realistic conditions. Using this framework, we can capture relevant performance between different algorithms. The evaluation framework assesses algorithm behavior under more general and practical conditions to help understand algorithms' strengths and limitations in realistic settings, including different spatial configurations and workloads.

Section 5.1 introduces the evaluation aspects. Then, in Section 5.2, we explain assmuptions and problem settings that are used in the evaluation. This is followed by the simulation environment in Section 5.3. Section 5.4 explains the hidden factor among H-OLTSP instances. Section 5.5 discusses how the H-OLTSP problem in-

stances are generated, as there are no standard public datasets specifically available for H-OLTSP. Section 5.6, provides a detailed overview of the structure and operational flow of the evaluation framework. Finally, we describe how this evaluation fraework can be used for comparion among H-OLTSP algorithms in Section 5.7.

5.1 Evaluation Aspects

To evaluate the practical performance of H-OLTSP algorithms, we consider two fundamental aspects: *responsiveness* and *efficiency*. They are core concepts that guide how we interpret the behavior of H-OLTSP algorithms.

5.1.1 Responsiveness

Responsiveness reflects how quickly the algorithm produces decisions, indicated by its total CPU runtime during simulation. It reflects how quickly the algorithm can respond to dynamic changes. In the H-OLTSP, the dynamic change is when new requests are released over time.

Traditional theoretical models that analyze the competitive ratio do not directly consider responsiveness. In these models, the time it takes an algorithm to compute or update a route is not included in the competitive ratio analysis.

Competitive analysis (Appendix B.1) focuses only on how the total completion time compares to the optimal offline solution. In an online setting, when requests arrive frequently or the number of requests requiring service increases, the algorithm

may take longer to process the new input and decide on the next scheduling step. These delays can seriously affect the server's response speed to changing situations. Therefore, to evaluate the algorithms more realistically, we include the *response speed (or responsiveness)* as a separate aspect that reflects the actual time spent on decision making during execution. As a result, algorithms that appear to be efficient in competitive analysis may exhibit considerable latency in the real world because of these real-time computational requirements.

To assess this critical aspect of the *practical* performance of H-OLTSP algorithms, the proposed evaluation framework explicitly includes responsiveness as an evaluation aspect. In the evaluation, we quantify the responsiveness by measuring the *total CPU runtime* consumed by the simulated algorithms during the completion of an OLTSP instance.

In our simulations, the main CPU runtime is occupied by the algorithm's decision-making steps (described in Section 5.6), and thus variations in CPU runtime only reflect differences in the algorithm's responsiveness. Shorter CPU runtime indicates that the algorithms are more responsive and can schedule routes more quickly. By analyzing responsiveness under different levels of complexity, we can evaluate the efficiency of each algorithm in maintaining timely responses in real-world operational scenarios.

5.1.2 Efficiency

Efficiency refers to how well an algorithm solve H-OLTSP instances. It reflects the performance of an algorithm by measuring the completion time (the objective value of the H-OLTSP). That is, the time steps to complete all requests and return to the starting point. Therefore, efficiency measures an algorithm's ability to guide a server in completing an H-OLTSP instance in fewer objective value.

Competitive analysis focuses on efficiency. But it focuses only on the worst-case efficiency. Therefore, in the evaluation framework, we empirically assess the efficiency of each algorithm by recording its objective values on multiple randomly generated problem instances. These instances vary in the number of requests and the distribution of release time. We aim to evaluate the practical performance of the H-OLTSP algorithm by considering more general, realistic scenarios rather than focusing on worst-case instances. This allows us to understand the actual quality of the algorithm's decisions in practice, beyond the theoretical guarantees provided by competitive analysis.

5.2 Assumptions and Problem Setting for Practical Performance

To better evaluate the performance of the OLTSP algorithm in real-world applications, we define a set of settings or assumptions for the OLTSP problem. These

assumptions do not affect the correctness of the OLTSP algorithms, which means these assumptions are applied at the problem level and do not restrict or invalidate any of the OLTSP algorithms under consideration. That is, all algorithms remain applicable and correct under these assumptions. Introducing these assumptions aims to better align the problem setting with real-world data, particularly by making it compatible with TSPLIB [1] instances used to define the metric space. These assumptions help structure the OLTSP model in a way that reflects practical applications, making it easier to construct a consistent evaluation framework and interpret algorithm performance in realistic scenarios.

First, we evaluate H-OLTSP algorithms in discrete metric spaces, where the set of locations is finite and the distance between any two distinct points is a positive integer. In such spaces, the server can only move between defined points in the metric, and each movement has a well-defined, countable travel cost. This excludes any notion of continuous movement or fractional positions, and aligns with practical settings such as graph-based road networks or TSPLIB instances, where distances are represented as integers and locations are treated as nodes. Additionally, all request release times are integers, and time is modeled as advancing in discrete steps of one unit. That is, time progresses as $t = 0, 1, 2, \ldots$, with each step representing one unit of elapsed time.

This discretion setting allows for a more straightforward and more consistent simulation environment. It avoids complications arising from floating-point rounding,

ensures deterministic behavior when measuring completion time and travel durations, and makes implementing the evaluation framework more straightforward. Moreover, many benchmark instances in TSPLIB already use integer-valued distances, which aligns naturally with this assumption. Modeling time in discrete units also reflects real-world scheduling systems, which often operate on fixed time intervals (e.g., seconds, minutes, or slots).

Second, this thesis uses the concept of U-Turn (Definition 4). Ausiello et al. [3] introduced the U-Turn and mentioned it is allowed in OLTSP. If the assumption of U-Turn is applied, it means that when a server moves on a path between two points, if it wants to change direction, then U-Turn is the only option. The server's movement is limited to a predetermined route between specified points. For example, if a server travels from point a to point b, it cannot change its destination midway. If the server must change direction, applying a U-Turn to return to the point a is its only option.

This movement constraint applies uniformly to all algorithms, including the optimal offline adversary. The U-Turn assumption is part of the problem model itself, not a restriction on specific algorithms. Even though the optimal offline algorithm has full knowledge of all future requests, it must operate within the same physical movement rules as any online algorithm. Therefore, if the server under the offline algorithm needs to reverse direction while moving from a to b, it must first complete a U-Turn by returning to a. This ensures a fair and consistent comparison across all

algorithms under the same motion constraints.

Definition 4. *U-Turns in OLTSP*

A U-turn is an action in which the server changes its direction of travel and moves back toward a location it has already visited. In detail, assume that a server is dealing with an H-OLTSP instance $\{< M, d >, Q, o\}$ *where* $< M, d >$ *is a metric space,* o *is the origin, and* Q *is the list of requests. We suppose the server is moving from point* a *to point* b*, where* $a, b \in M$*. A U-Turn occurs at time* t *if the server was moving in one direction before time* t*, and after time* t*, it starts moving in the opposite direction, heading back toward point* a*.*

We provide a detailed example when the u-turn is applied, consider a scenario where the server (s) moves from x_1 to x_2 and redirects to x_3 (Figure 5.1(a)) where x_1, x_2 and x_3 are points of metric space. In detail, x_1 and x_2 are the requesting locations. This scenario occurs when the server has just visited x_1 and is on its way to visit x_2 from x_1. At this point, a new request x_3 is released. In this case, if the server wants to move toward x_3, it must first complete its journey to x_2, or apply a U-Turn to return x_1. These two options are shown in Figures 5.1(b) and 5.1(c). Hence, if U-Turn is applied, the server can only alter its path after reaching one of the exits of the path, reflecting the constraints of operating within the metric space(a highway-like road network) [3].

In this thesis, we allow the use of U-turns under the assumption that the original OLTSP model permits it and do not compromise the correctness of the algorithm.

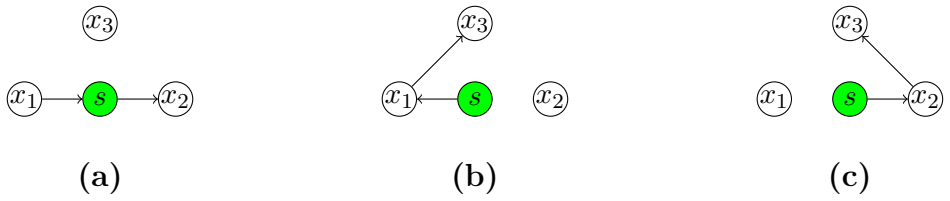


Figure 5.1: An Example of U-Turn

Our primary goal is to evaluate the algorithm in a practical setting. We base our experiments on TSPLIB [1] instances derived from real geographic data to support this. However, these instances represent offline TSP problems defined over graphs or discrete metric spaces. In such settings, movement occurs from one node to another along the edges defined in the graph. A node can only be reached if it is directly connected to the current node by an edge. Once moving along an edge, switching to another node that is not part of that edge is impossible. The movement must continue to one of the two endpoints before further transitioning. This ensures that all movements follow the structure of the graph.

Thirdly, they bring up another issue based on the U-Turn and discrete metric space assumptions. Suppose the server has just completed a request at point a and is moving toward point c to serve the subsequent request. During this movement, the server may pass through other points in the metric space. However, in our model, if the server passes through a location where a request has already been released or becomes released while in transit, that request is not considered completed unless the server explicitly planned to serve it. In other words, simply passing through

a requested location without deciding to stop there does not count as visiting or serving it. The server must intentionally choose to include a request in its current decision for that request to be marked as served.

This assumption is introduced to maintain consistency with the discrete nature of the metric space used in our evaluation. In a discrete metric space, the server moves along predefined edges between distinct points, and there is no meaningful notion of "incidentally visiting" intermediate locations unless explicitly chosen. By treating each movement as a direct transition from one point to another, we ensure that serving a request requires a deliberate action from the server, not just incidental passage. Without this assumption, the model could ambiguously allow requests to be completed without planning, breaking the alignment between algorithmic decisions and observed performance. Thus, the assumption enforces clarity in decision-making and ensures that the evaluation faithfully captures the server's planned behavior.

In addition, this assumption reflects the reality of discrete systems such as road networks and warehouse layouts, where the server cannot serve a location unless it intentionally stops there. Take robots, for example. When a robot is sent to a location, it must go there and stop to do a task. Just passing through a location without stopping means that the task at that point is not done.

In the PAH algorithm, we often need to measure the distance between the server and the origin. When the server is located exactly at a request point, this distance is given directly by the metric. However, if the server is currently moving between

two points in the metric space, the standard distance function does not apply, since the server is not at a defined point of the space. In this case, we define the distance as the length of the shortest route the server would have to travel in order to return to the origin from its current position.

5.3 Simulation Environment

The H-OLTSP algorithms are designed for a server or agent that operates in a real-time, online environment. In such settings, the server makes decisions step by step as requests are released. These decisions affect the environment, such as the position of the server or which requests require service. We create a simulation to study the performance of H-OLTSP algorithms. The simulation creates a virtual environment that mimics the real-world setup. It releases requests over time, tracks the movement of the server in the metric space, and processes the algorithm's decisions in real time. This way, the H-OLTSP algorithm can be executed on a computer without needing a physical server or agent.

During the simulation, we record data that aid in evaluating the algorithms. This includes the total completion time (the objective value) and the CPU runtime (which reflects how fast the algorithm makes decisions). By running the same simulation setup with different algorithms, we can make fair comparisons and understand how each algorithm performs under the same conditions. We implemented this simulation environment using an operation unit that performs the following functions:

- Recording and advancing time: the operation unit increments the simulation clock by one unit of time and updates all system states accordingly.
- Releasing requests: The operation unit will release new requests to allow the algorithm to adjust its current scheduling policy based on the latest release request.
- Server movement: The server moves through the metric space according to the decisions made by the algorithm. After each decision, the operation unit updates the server's position by simulating its movement. The movement is processed step by step. The operation unit updates the clock and server location at each time step based on the current motion.
- State information: Besides the release requests, the operation unit provides the algorithm with other state information, such as the current time and the server's location.
- Request completion verification: The operation unit verifies the completion of each request.
- Logging: The operation unit records responsiveness, efficiency, and server movement per unit of time.

The operation unit provides the interaction between the H-OLTSP algorithm, online instances, and the server. This provides a means to test and evaluate the

algorithm on a set of instances under the same constraints for consistent evaluation and comparison. The following section discusses the generation of problem instances.

5.4 Request Load or Load

Before describing how the problem instances are generated, it is important to first explain a hidden factor within them: the request load or load. In the evaluation of H-OLTSP algorithms, request load refers to the number of active requests the server must consider at a given step time, which is formally defined in the Definition 5. The Load is a key factor that influences both scheduling complexity and the algorithm's performance. Unlike the total number of requests in an instance, which represents the overall problem size, the request load determines how difficult the scheduling decisions are at each moment.

Definition 5. *Load or Request Load*

In the context of the H-OLTSP, the request load *(or simply* load*) at a scheduling time t is defined as the number of unserved requests that have been released by time t and must be considered when computing a new route.*

Let $Q = \{(p_1, t_1), (p_2, t_2), \ldots, (p_N, t_N)\}$ be the full set of requests, where each pair (p_i, t_i) consists of a location p_i and its release time t_i. Then, the load at time t is defined as:

$$\text{load}(t) = |\{(p_i, t_i) \in Q \mid t_i \leq t \text{ and } p_i \text{ is unserved at time } t\}|$$

This quantity represents the size of the current scheduling problem faced by the server. A higher load typically increases the computational cost of scheduling.

Request load is a hidden factor because it is not directly tied to the total number of requests. An instance can have a large number of requests while maintaining low request load if the requests are spread out over time. In such cases, the algorithm only needs to handle a few requests at each decision point, and the scheduling complexity stays low.

For example, consider an instance with 100 total requests. If the release times are spaced apart using an increasing interval of $2 \times d_{\max}$, then the requests become active one by one where d_{max} in maximum distance between tow points in metric of the instance. At each step, the server only sees a single active request, so the algorithm performs a simple scheduling action without needing to solve a complex tour. In contrast, if all 100 requests are released at once, the algorithm must immediately compute a full tour over all requests, resulting in significantly higher scheduling complexity.

This distinction is important for evaluating both responsiveness and efficiency. An algorithm may perform well under large instances with low load, but struggle when the request load becomes high. To understand this behavior, experiments need to control and vary the request load independently of total request count, allowing us to assess how each algorithm handles different levels of scheduling difficulty.

5.5 Problem Instance Generation

A generator capable of generating diverse H-OLTSP instances is essential to our evaluation framework because it allows us to comprehensively evaluate the algorithm's performance. This section discusses the construction of the generators used in the evaluation framework.

As described in Definition 1, an H-OLTSP instance consists of three parts: a metric space, an origin, and a request list. Since no standard benchmark dataset is tailored for H-OLTSP, we construct the problem instances by adapting data from TSPLIB [1], a well-established benchmark library designed for evaluating classical TSP algorithms.

5.5.1 Metric Space and Origin Generation

Metric Space: We begin by discussing how the metric space is constructed. TSPLIB [1] is used as the metric space because it provides standardized and curated TSP instances with real-world coordinates and well-defined distance functions. Unlike raw geographic datasets, TSPLIB is designed specifically for evaluating routing algorithms, offering consistency and avoiding the need for complex preprocessing. It includes a wide range of instance sizes and structures, which supports controlled and diverse evaluations. Additionally, TSPLIB is widely supported by existing tools and APIs, making it convenient to load and use the data directly in simulation and algorithm testing. In TSPLIB, each instance provides a set of nodes and pairwise

distances between them. We use these two elements to construct the metric space directly. Specifically, we use the set of nodes as the set of points M on the metric space. And we define the distance function d in $M \times M$ using those pairwise distances between each node. To ensure that the resulting space satisfies the mathematical properties of the metric space, we use only symmetric TSP instances from TSPLIB [1]. These instances provide symmetric distances and are well-suited for modeling Euclidean distances since they are often based on realistic maps.

Origin: Once the metric space $\langle M, d \rangle$ has been constructed, we randomly select a point from M as the origin, representing the server's fixed start and end positions.

5.5.2 Request List Generation

The request list directly affects the performance of the algorithms. In H-OLTSP, each request contains a location and the release time.

Generation of Requested Locations

For location generation, we uniformly randomly sample from M to ensure spatial diversity of requests in the instance. In particular, to enhance generality and diversity, we randomly select all points in the metric space. This means that the origin may also be selected as a request location.

Generation of Release Times

Generating release times for requests is an important part of instance generation because it directly affects the request load. Request load refers to how many requests are active at the same step time, and this is determined by how the release times are distributed. If many requests are released at the same or nearly the same time, they become active together, resulting in a high load. On the other hand, if the release times are more spread out, only a small number of requests become active during each scheduling step, which creates a low load.

This relationship is important because it controls the difficulty of the scheduling problem the algorithm faces. High load increases scheduling complexity, as the algorithm must consider more requests at once. Low load simplifies the decision process, since fewer requests are available at each step. Therefore, the way release times are generated directly influences the load level and, in turn, the performance behavior observed during evaluation.

Because the request load depends on how release times are distributed, the way we generate release times should reflect the nature of the scheduling challenge. In this thesis, we aim for a general and flexible setup. To achieve this, we generate release times by sampling from a normal distribution.

This choice reflects the behavior of many real-world systems. Requests often arrive in bursts around peak periods, rather than being evenly or randomly spread over time. The normal distribution captures this pattern by concentrating most

requests near the mean while still allowing a few to appear earlier or later. It provides a realistic balance between structured timing and natural randomness.

In addition, the normal distribution supports controlled experimentation. By adjusting the standard deviation, we can control how tightly or loosely the requests are distributed in time. A smaller standard deviation results in many requests becoming active at nearly the same step time, creating a high load and increasing scheduling difficulty. A larger standard deviation spreads the release times further apart, reducing the number of active requests at each scheduling step and lowering the load. This flexibility allows us to evaluate how different algorithms perform under varying load conditions, and helps us compare their responsiveness and efficiency in a consistent and meaningful way.

In the normal distribution $\mathcal{N}(\mu, \sigma^2)$, the mean μ defines the center point of the request release time, and the standard deviation σ controls the tightness of the distribution of the request release time around the center point. A smaller σ will cause many requests to aggregate at the central time point, resulting in a high-load scenario. A larger σ will produce a wider release time distribution, thereby reducing the degree of aggregation in the request time and achieving a more relaxed scheduling environment.

Furthermore, we use a truncated normal distribution to ensure that all release times fall within a fixed time interval $[0, t_{\text{last}}]$. This control is necessary because the request load depends not only on the standard deviation σ but also on the overall

time range. For example, even if σ is small and requests are tightly clustered, a very long simulation interval would still spread them out and result in low load. Conversely, if σ is large but the time interval is very short, many requests may become active at the same time, creating an unintended high load. By fixing the time range, we make the influence of σ meaningful and ensure that the generated release times produce consistent and predictable load conditions. This allows us to vary request load systematically by adjusting the number of requests while keeping the time interval fixed.

5.5.3 Structure of Generation

The design of the problem instance generator enables us to obtain a set of instances that are spatially random and temporally diverse. Specifically, all requested locations are random, which avoids the trap of only considering the best or worst case and makes the instance set general. The temporal diversity enables the instance set to comprehensively test the algorithm's performance by changing parameters such as the number of requests, μ, σ, and the truncation range. The specific steps of problem instance generation are described in Definition 6.

Definition 6. *H-OLTSP Instance Generator*

Inputs:

1. *A symmetric TSP instance of TSPLIB*

2. *Number of requests N*

3. *Last possible release time* t_{max}

The Generator do following steps:

1. *Use the TSP instance to construct the metric space* $\langle M, d \rangle$.

2. *Randomly select a point from* M *as the origin* o *(uniform distribution).*

3. *Generate a list of requests* Q*, where each request* $r = (p, t)$*:*

   - *Location* p*: randomly pick from* M *(uniform distribution),*
   - *Release Time* t*: sample from a truncated normal distribution* $\mathcal{N}(\mu, \sigma^2)$ *on the bounded interval* $[0, t_{\max}]$.

Outputs: $\langle M, d \rangle$*,* o*,* Q

5.6 Evaluation Framework Workflow

The evaluation process consists of four main phases: input parameters, instance generation, dynamic environment simulation, and evaluation metrics output. The framework is only used to systematically show how we perform the evaluation steps in this thesis. It is not a very complete plug-and-play framework for general usage.

Figure 5.2 shows an overview of the process. The structure and functionality of the framework are described next.

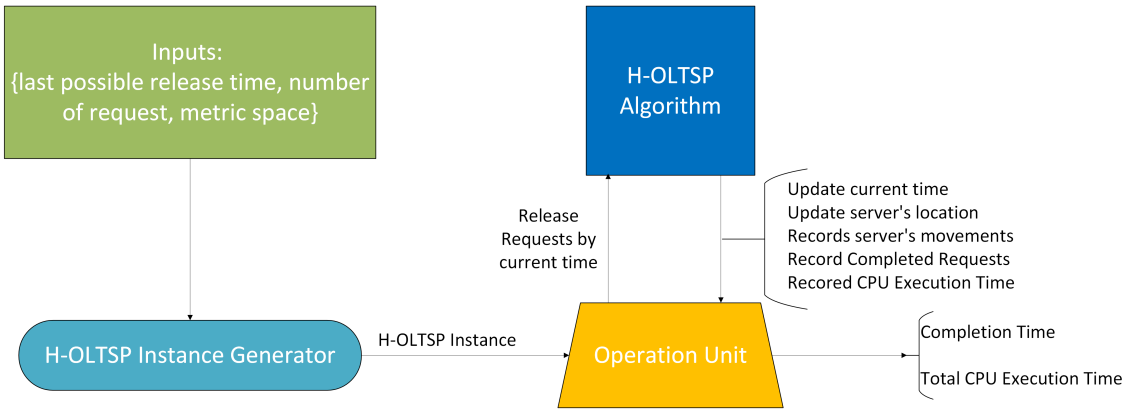


Figure 5.2: Structure of Evaluation Framework

5.6.1 Input Parameters and Instance Generation

The evaluation framework first accepts three required parameters that define how H-OLTSP instances are generated in each experiment. These parameters are as follows:

- a symmetric TSP instance from TSPLIB [1].
- number of requests that specifies the number of requests generated by the generator.
- last possible release time.

The metric space is constructed directly from a symmetric TSP instance in TSPLIB [1]. The set of nodes in the instance defines the set of points M in the metric space, and the distance between each node in the instance defines a function

d in $M \times M$. When the evaluation framework receives the above three parameters, it generates a H-OLTSP instance by the instance generator (Definition 6).

5.6.2 Dynamic Environment Simulation

After the H-OLTSP instance is generated, the simulation environment is initialized. The environment is implemented by an operation unit and manages the interaction between the algorithm, the online instance, and the server. At each time unit, the operation unit detects the decisions updated by the algorithm. When the algorithm makes a new decision, the operation unit immediately simulates the execution of the server. At the same time, at each time unit, the operation unit also monitors whether new requests are released. If there are new requests, it passes them to the algorithm as soon as they are released. The algorithm can obtain system information from the operation unit, including the current time, the server location, and all requests issued up to that moment. Based on this information, the algorithm determines the server's next destination.

Although a single movement decision may take multiple time units to complete (depending on the length of the route), the operation unit continuously monitors the system status at each unit time, and any changes in the system status are immediately notified to the algorithm. For example, if a new request is released during the simulation of server movement, the algorithm will be notified immediately. At this time, the algorithm can interrupt the current route of the server and make new deci-

sions. In addition, the environment will promptly update the location of the server, check the completed requests, and record key performance indicators, including the total travel distance, unit time, and the cumulative CPU time used by the algorithm. This process continues until the algorithm successfully guides the server to process all requests and return to the origin.

5.6.3 Outputting Evaluation Metrics

At the end of the simulation, the framework outputs two final metrics:

- Completion time: the total unit time required to serve all requests and return to the origin. It reflects the efficiency.
- CPU execution time: the total CPU time of the simulation, which reflects the responsiveness.

5.7 Discussion-Evaluation Framework for Comparison

In the simulation, we generate an instance under one configuration and simulate all the algorithms to be compared to solve this instance. This design choice is crucial to maintain consistency and eliminate differences caused by the generation of random instances. Suppose we compare instances where each algorithm solves the same

feature instance but with different configurations, even slight differences in request location or release time may affect performance metrics. This will make it challenging to attribute performance differences to the algorithm rather than differences in the test instances.

To evaluate multiple algorithms, we extend the evaluation framework to allow a single H-OLTSP instance to be tested in multiple simulation environments. Specifically, when an H-OLTSP instance is generated, it is not used for a single evaluation, but is replicated and distributed to multiple simulation environments. Each simulation environment is associated with a specific algorithm to be evaluated, and each environment is independent of the others.

Each simulation environment independently simulates the same H-OLTSP instance with an algorithm by each operation unit, allowing each algorithm to interact with the same request sequence under the same conditions. The simulation process in each environment is the same as the process described previously for evaluating a single algorithm. Similarly, after the simulation is completed, each simulation environment generates a set of evaluation outputs. These outputs include the completion time (objective value) and the CPU execution time for each algorithm. By collecting and comparing these results across all environments, we can make a consistent and fair comparison of the performance of multiple algorithms under the same conditions.

Chapter 6

Evaluation of PAH with Different Scheduling Strategies

This chapter presents the evaluation of the PAH algorithm with different scheduling strategies. The details of PAH and the scheduling methods are described in Chapter 3. Section 6.1 explains the configuration of the evaluation framework and the environment settings. These are also used in the evaluations in Chapters 7 and 8. Section 6.2 presents the experimental results, and Section 6.3 discusses the experimental results under two aspects: responsiveness and efficiency.

6.1 Evaluation Configuration

This section describes how the experiments in this chapter and future chapters are configured. The goal is to compare different algorithms under consistent spatial and temporal conditions while varying key parameters that influence request load. The configuration includes both the hardware used to run the experiments and the input settings used to generate problem instances for evaluation.

6.1.1 Hardware Configuration

All experiments are executed on a Linux machine equipped with an Intel Core i7-8700k processor and 32 GB memory. The evaluation framework is implemented in Python 3 and runs sequentially. This setup is sufficient to process the problem instances used in this study and allows for stable and interpretable measurements. Since no parallelism or hardware acceleration is used, all differences in runtime and scheduling delay reflect only the behavior of the algorithms themselves.

6.1.2 Evaluation Framework Configuration

The evaluation framework requires three main inputs to generate a problem instance for the Homing Online Traveling Salesperson Problem (H-OLTSP):

- TSP instance that defines the metric space.
- Release-time distribution for generating when requests appear.

- Total number of requests in the instance.

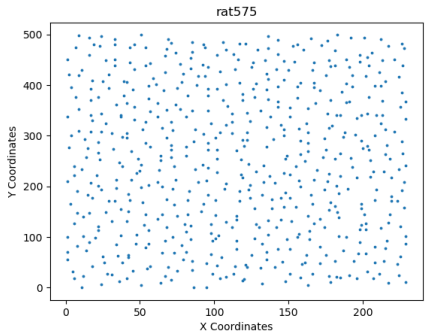
Some of these inputs are fixed to maintain consistency across experiments, while others are varied to explore how algorithm performance changes under different levels of request load. The configuration choices for each input are described below.

Fixed Metric Space

All problem instances use the same TSP instance, named `rat575` (from TSPLIB [1]), which contains 575 locations and a symmetric distance matrix. This instance defines the metric space used in all experiments. Figure 6.1 presents the details of `rat575`. The left side displays all the points in the instance, where distances can be computed based on their X and Y coordinates. The right side provides information on the corresponding distance values.

We use the TSPLIB instance rat575 because it offers a balanced size suitable for our experimental needs. Smaller instances may limit spatial variety and reduce the range of possible request locations, which can constrain the diversity of request patterns. On the other hand, using very large TSPLIB instances significantly increases the time required to generate requests. This is because each experiment involves repeatedly loading the TSPLIB file, and larger files take longer to read and parse. With 575 nodes, rat575 is considered a mid-size instance in TSPLIB. It provides enough spatial coverage to support request sets ranging from 5 to 40, which is the range used in our experiments. Choosing an instance that is much larger than

the maximum number of requests ensures we maintain sufficient variety in request generation without incurring unnecessary data loading overhead.



Mean Distance	197.35
Maximum Distance	534
Minimum Distance	2
Median Distance	182

Figure 6.1: Summary of rat575 (an instance of TSPLIB[1])

We fix the metric space for all experiments to ensure fair comparison. Since the focus of this evaluation is on scheduling behavior rather than spatial properties, varying the structure of the metric space is unnecessary. All algorithms operate on the same list of locations and pairwise distances, so fixing the instance removes any unrelated variation that could interfere with comparing responsiveness and efficiency.

Temporal Request Configuration: Fixed Interval with Varied Distribution and Load

To generate request release times, the framework uses a normal distribution truncated to a fixed time interval. The motivation for this design is discussed in Section 5.5, and here we explain how it is applied in the experiments. The fixed interval defines the time window during which all requests may be released and remains the

same across all experiments to ensure fair comparison. The mean of the normal distribution is placed at the center of this interval and held constant. This practical choice ensures that request release times are symmetrically distributed, avoiding clustering too early or too late in the simulation.

Centering the mean is also important for correctly controlling request load through the standard deviation σ. If the mean were skewed toward one end of the interval, even a small σ could result in requests being unintentionally spread out over time due to truncation, effectively lowering the actual load. Likewise, a large σ with an off-center mean could cause many release times to be clipped at the interval boundaries, distorting the intended distribution. By keeping the mean fixed at the center, we ensure that σ behaves as expected. That is smaller σ values produce higher load by clustering requests near the center, and larger values produce lower load by spreading requests more widely. This setup makes the control of load levels predictable and meaningful throughout the experiments.

Within this fixed temporal structure, two parameters are varied across experiments to control request load and simulate different scheduling conditions. They are

- the standard deviation (σ) of release times controls how tightly or loosely the requests are distributed around the center of the interval. A small standard deviation causes many requests to be released close together in time. This leads to more requests being active and unserved during each scheduling step,

increasing the scheduling complexity and resulting in higher request load. In contrast, a larger standard deviation spreads requests more evenly across the interval. This reduces the number of active requests at any given time, making scheduling easier and producing a lower load.

- the number of requests (N) determines the total volume of requests in each instance. When the time interval is fixed, increasing the number of requests increases the density of release times, especially around the center of the interval. This makes it more likely that several requests will be active at the same time, which also increases request load. By varying the total number of requests, we simulate systems with light, moderate, or heavy demand while keeping the temporal structure consistent.

Together, the standard deviation and the number of requests define each instance's load characteristics. By adjusting these two parameters, the framework generates a wide range of problem conditions. This allows us to observe how algorithms' performance changes in response to different types of scheduling demand.

In each experiment, a problem instance is generated using a specific configuration of these parameters. All algorithms are then evaluated on the same instance. This ensures that every comparison is made under identical conditions and that performance differences are caused by algorithmic behavior rather than differences in the data.

6.2 Experimental Results

In this section, we present and explain the experimental results of PAH combined with different scheduling algorithms under different cases. In the results and following sections, we use the names PAH-DP (PAH using dynamic programming), PAH-Chris (PAH using Christofides heuristic), PAH-GENI-3 (PAH using GENI with searching neighborhood of size 3), PAH-GENI-N (PAH using GENI with searching full tour), and PAH-GENI-N-RS (PAN-GENI-N with rescheduling modification). All these scheduling algorithms are described in Section 3.2. In addition, in all tables, OPT is the offline optimal cost, N is the number of requests and σ is the standard deviation.

In the following experiments, we use various standard deviations where $\sigma = d_{median}, 5\times d_{median}$, and $10\times d_{median}$ for the normal distribution of request release times to control the temporal spread of requests and simulate different load conditions. The choice of d_{median} as the base unit is grounded in the structure of the metric space. Since d_{median} represents the typical distance between two locations, it provides a natural scale to relate time and space in the simulation.

In the experiments, we define three load conditions by varying the standard deviation σ of the normal distribution that generates the request release times. The high-load condition uses $\sigma = d_{median}$, the mid-load condition uses $\sigma = 5\times d_{median}$, and the low-load condition uses $\sigma = 10\times d_{median}$. When $\sigma = d_{median}$, most requests appear within a narrow time window, creating high-load conditions where many unserved requests are present when the server needs to reschedule. This increases scheduling

pressure and highlights algorithm responsiveness and efficiency under stress. With $\sigma = 5 \times d_{median}$, requests are more spread out. At $\sigma = 10 \times d_{median}$, the release times further spread out.

It is important to note that these load conditions do not directly redefine load(t) from Definition 5, which remains the number of unserved requests active at time t. Instead, the load condition reflects the statistical tendency of the release-time distribution to produce different scheduling pressures. When σ is small (high-load condition), requests are clustered in time, so many become active within a short interval and load(t) tends to be large at decision points. When σ is moderate (mid-load condition), requests are more evenly spread and load(t) is typically smaller. When σ is large (low-load condition), requests are widely dispersed over the interval, so load(t) remains low most of the time.

Thus, the load conditions describe the distributional setting that influences how load(t) behaves during the experiment, with smaller σ leading to higher scheduling pressure and larger σ leading to lower scheduling pressure. This setting for release-time distributions is applied consistently in all experiments presented in this thesis.

We present and analyze the experimental results with three different cases:

1. Short request release time interval (Section 6.2.1): In this case, requests are restricted to a short limited time window. Increasing the number of requests within this window will highly increase the system load. This allows us to study the strategies under high scheduling pressure. We examine three differ-

ent request release time distributions, each modeled by a normal distribution $\mathcal{N}(\mu, \sigma)$ with varying standard deviations: $\sigma = d_{median}$, $\sigma = 5 \times d_{median}$, and $\sigma = 10 \times d_{median}$ where d_{median} is the median distance between points in the metric space (rat575). In this setting, load (details in Definition 5) refers to the number of unserved requests present when the server recomputes a route.

2. Extended Time Interval (Section 6.2.2): To understand how the statistical shape of the release time distribution influences scheduling behavior, we also include experiments under an extended time interval. In this setting, the release window is comparable to or larger than σ, allowing the normal distribution to be preserved more accurately. As a result, request release times follow the intended pattern, and scheduling pressure varies more clearly between high-load, mid-load, and low-load conditions. This extended setting provides a more practical environment for evaluating how each scheduling algorithm performs when requests are spread over time in a statistically meaningful way.

3. Extended Number of Requests: We examine the results for larger problem sizes to understand how increasing the number of requests affects the role of scheduling in PAH (Section 6.2.3). While the extended time interval keeps the load relatively low, a large N still introduces additional complexity. This setting allows us to observe how each algorithm scales in terms of both efficiency and responsiveness, and whether their scheduling quality and computational cost remain practical as the problem size grows.

6.2.1 Experimental Results under Extreme Case

We first evaluate PAH with different scheduling algorithms in instances where the number of requests ranges from 5 to 17, using a fixed time interval $[0, 10 \times d_{min}]$ where d_{min} is the minimum distance of the metric space (rat575). This setup is designed to control and gradually increase the complexity of scheduling by fixing the total time span while varying the number of requests. When there are only five requests, it is possible for the server to complete each request individually before the next one is released, resulting in a scheduling load of 1 at every rescheduling event. This can occur, for example, when all requested locations are at the minimum possible distance from the origin. If each request is released at intervals of $2 \times d_{min}$, the server can serve one request at a time without accumulating unserved requests. However, by setting the maximum release time to $10 \times d_{min}$, we ensure that once the number of requests exceeds 5, the time interval between requests becomes too small for the server to complete each one individually. In other words, the average inter-arrival time drops below the time needed to serve a single request. This guarantees that some requests will remain unserved when new requests are released, causing the server to handle multiple outstanding requests during rescheduling. By the Pigeonhole Principle, this leads to a scheduling load greater than 1, and the complexity of scheduling increases accordingly by increasing the total number of requests.

It is important to note that in this setting, the overall time interval for request release is extremely short. As a result, scheduling pressure remains consistently high,

regardless of the distribution used. In the extreme-case experiments, the standard deviation σ is intentionally set larger than the time interval, even though this distorts the normal distribution by truncating much of its mass. This design choice introduces sufficient variability in release times and preserves meaningful differences between high-load, mid-load, and low-load conditions, despite the compressed time frame. While the resulting distributions are no longer fully normal, they still reflect the effects of densely packed requests in a constrained environment. Under this extreme setup, we focus on observing how scheduling algorithms affect PAH performance as the number of requests increases, particularly under intense scheduling pressure.

Tables 6.1, 6.2, and 6.3 report the average completion time of PAH combined with various scheduling algorithms under a fixed, short release time interval. In this setting, we vary the standard deviation σ (d_{median}, $5 \times d_{\text{median}}$, and $10 \times d_{\text{median}}$) to represent different load conditions. However, because the overall interval is extremely short, most requests are still released close together in time, limiting the actual impact of σ on the spacing between requests. As a result, this setup primarily tests how each scheduling algorithm performs under high scheduling pressure as the number of requests increases. Despite the limited influence of σ here, we include results for all three values to enable comparison with the extended time interval experiments, where σ meaningfully affects the distribution and provides deeper insight into algorithm behavior across varying load conditions.

N	5	7	9	11	13	15	17
PAH-DP	**736.80**	**1006.40**	**1126.00**	**1087.20**	**1172.40**	**1247.40**	**1299.60**
PAH-Chris	782.80	1059.40	1193.40	1112.60	1299.20	1352.60	1407.00
PAH-GENI-3	745.00	1088.20	1174.60	1161.40	1223.00	1363.40	1378.60
PAH-GENI-N-RS	764.80	1008.20	1128.20	**1087.20**	1175.20	1249.20	1305.80
PAH-GENI-N	742.80	1008.20	1127.00	**1087.20**	**1172.40**	1250.00	1306.00
OPT	717.60	985.20	1105.40	1065.80	1152.60	1225.60	1280.20

Table 6.1: Average Completion Time ($\sigma = d_{median}$)

N	5	7	9	11	13	15	17
PAH-DP	**901.00**	**1059.20**	**1138.00**	**1160.80**	**1266.80**	**1284.20**	**1381.40**
PAH-Chris	923.00	1105.80	1175.80	1198.20	1362.20	1373.20	1524.20
PAH-GENI-3	942.40	1083.20	1167.40	1222.80	1327.60	1398.00	1485.20
PAH-GENI-N-RS	903.80	1061.00	**1138.00**	**1160.80**	1268.20	1290.80	1390.60
PAH-GENI-N	903.80	**1059.20**	**1138.00**	1163.20	1273.80	1285.40	**1381.40**
OPT	881.40	1039.20	1117.60	1141.00	1246.80	1266.20	1361.40

Table 6.2: Average Completion Time ($\sigma = 5 \times d_{median}$)

N	5	7	9	11	13	15	17
PAH-DP	**920.20**	**1001.80**	**1013.20**	**1146.20**	**1261.60**	**1123.80**	**1316.40**
PAH-Chris	962.60	1043.40	1112.60	1223.00	1337.00	1214.20	1422.40
PAH-GENI-3	923.80	1028.80	1074.00	1212.00	1392.60	1207.40	1474.40
PAH-GENI-N-RS	**920.20**	**1001.80**	**1013.20**	1147.60	**1261.60**	1124.60	1318.00
PAH-GENI-N	923.80	**1001.80**	1017.40	1147.60	1264.40	**1123.80**	1320.80
OPT	902.60	981.60	992.40	1125.80	1241.40	1104.20	1296.20

Table 6.3: Average Completion Time ($\sigma = 10 \times d_{median}$)

Across all three tables of average completion time, PAH-DP consistently achieves the lowest average completion time in every column, which confirms that dynamic programming produces the most efficient routes in terms of objective value. PAH-GENI-N-RS and PAH-GENI-N closely follow. In contrast, Christofides and GENI-3 perform reasonably in smaller instances but fall behind as N increases.

Table 6.4 shows the average CPU execution time of PAH combined with each scheduling algorithm, recorded during the same experimental runs used to simulate the server completing requests in Tables 6.1, 6.2, and 6.3. While those tables report the completion time as the objective value of H-OLTSP, the values here represent the actual CPU time required to compute each schedule during the simulations. Subfigures (a), (b), and (c) correspond to the high-load ($\sigma = d_{\text{median}}$), mid-load ($\sigma = 5 \times d_{\text{median}}$), and low-load ($\sigma = 10 \times d_{\text{median}}$) conditions, respectively.

N	5	7	9	11	13	15	17
PAH-DP	4.45	9.78	34.47	192.96	1026.55	5467.77	18932.51
PAH-Chris	6.31	9.71	12.69	**13.72**	**19.08**	**21.42**	**21.60**
PAH-GENI-3	**4.19**	**7.75**	**10.65**	14.12	19.23	24.24	25.71
PAH-GENI-N-RS	4.39	13.85	44.11	114.82	292.99	664.05	1199.68
PAH-GENI-N	4.67	15.11	55.29	179.00	482.54	1127.06	1675.41

(a) $\sigma = d_{median}$

N	5	7	9	11	13	15	17
PAH-DP	5.28	9.72	29.88	221.23	966.70	5193.39	26828.01
PAH-Chris	7.39	9.57	10.95	15.31	20.52	**20.84**	**26.38**
PAH-GENI-3	**4.92**	**7.23**	**9.97**	**14.95**	**19.74**	23.79	33.11
PAH-GENI-N-RS	5.31	14.47	40.05	124.12	275.46	658.77	1366.13
PAH-GENI-N	5.30	15.66	48.73	190.56	454.79	1021.63	2256.49

(b) $\sigma = 5 \times d_{median}$

N	5	7	9	11	13	15	17
PAH-DP	5.58	9.26	36.90	233.73	902.48	4467.34	31409.00
PAH-Chris	7.96	7.88	11.77	15.57	**19.96**	**17.83**	**24.20**
PAH-GENI-3	**5.07**	**6.84**	**9.52**	**14.02**	20.04	20.75	29.85
PAH-GENI-N-RS	5.12	13.83	35.53	123.21	269.71	657.10	1440.68
PAH-GENI-N	5.70	14.30	52.68	200.11	455.38	838.25	2313.47

(c) $\sigma = 10 \times d_{median}$

Table 6.4: Average CPU Execution Time Under Different Distributions (ms)

Across all three tables of CPU execution time, PAH-DP clearly exhibits the highest computational cost, especially as the number of requests N increases. Although its CPU execution time remains negligible for small N, it grows rapidly

with the size of the problem. This exponential growth reflects the high time complexity of dynamic programming for computing the optimal TSP tour. In contrast, PAH-Chris and PAH-GENI-3 maintain consistently low CPU execution times for all values of N. Meanwhile, PAH-GENI-N-RS and PAH-GENI-N exhibit moderate CPU execution time growth, reflecting the cost of dynamic neighborhoods. As expected, PAH-GENI-N-RS is faster than PAH-GENI-N, especially for larger N, since its rescheduling ability.

In summary, the results demonstrate a clear trade-off: PAH-DP achieves the best completion time but with a high computational cost, while GENI-N-RS and GENI-N provide a strong balance between completion time and CPU execution time. PAH-Chris and PAH-GENI-3, although faster, generally yield higher objective values(competition time) and are more appropriate for lightweight or time-constrained settings.

6.2.2 Experimental Results under Extended Release-Time Interval

We initially used the time interval $[0, 10 \times d_{min}]$ to create extreme scheduling pressure in early experiments. This short interval ensures that when the number of requests exceeds five, the server cannot complete one request before the next arrives, forcing it to handle multiple unserved requests at each decision point. This setup is useful for stressing the system and observing algorithm behavior under constant high-

load conditions. However, it compresses the release timeline and limits the ability to simulate more realistic scenarios. For example, even when we vary the standard deviation σ, the tight time frame truncates the normal distribution, reducing the distinction between different load settings.

To support a more general evaluation while keeping consistency with earlier experiments, we extend the request release-time interval to $[0,\ 10 \times d_{median}]$, where d_{median} is the median distance between points in the metric space. In previous experiments, the interval was defined as $[0,\ 10 \times d_{min}]$, where d_{min} is the smallest pairwise distance. However, d_{min} tends to be very small and sensitive to outliers. As a result, using larger multiples of d_{min} would excessively expand the time interval and reduce control over scheduling pressure. At the same time, we want the interval to align with the scale used in the normal distribution for release times, where σ is set to d_{median}, $5 \times d_{median}$, or $10 \times d_{median}$.

We extend the release time interval from $[0, 10 \times d_{min}]$ to $[0, 10 \times d_{median}]$. This change provides a longer and more balanced interval while introducing only a minimal change. While this choice does not perfectly capture all aspects of the distribution, it produces looser and more meaningful variation in the distribution of release times compared to using d_{min}. To vary the request load, we set $\mu = 5 \times d_{median}$ as the center, and varying the standard deviation σ across d_{median}, $5 \times d_{median}$, and $10 \times d_{median}$. A small value of σ produces tightly clustered release times around the center, creating higher scheduling pressure. Larger σ values result in more dispersed release times. we

define three load conditions: high-load, mid-load, and low-load, obtained by setting σ to d_{median}, $5 \times d_{median}$, and $10 \times d_{median}$, respectively. This setup provides distinct load conditions while keeping the design consistent with the scale of the metric space.

N	5	7	9	11	13	15	17
PAH-DP	**2039.60**	2154.00	2192.40	2362.00	**2389.20**	2540.80	**2493.80**
PAH-Chris	2099.40	2212.20	2253.60	2398.40	2417.40	2640.80	2587.20
PAH-GENI-3	2053.60	2223.40	2266.00	2433.40	2431.60	2616.00	2584.80
PAH-GENI-N-RS	2082.60	**2152.80**	**2180.40**	2362.60	2440.80	2547.40	2501.20
PAH-GENI-N	2079.60	2156.60	2185.60	**2360.00**	2456.20	**2529.40**	2496.40
OPT	1574.60	1594.80	1621.60	1718.60	1841.80	1939.60	2011.40

Table 6.5: Average Completion Time ($\sigma = d_{median}$)

N	5	7	9	11	13	15	17
PAH-DP	2355.60	2399.40	2668.60	2624.20	2731.40	2862.80	**2929.00**
PAH-Chris	2306.40	2466.80	2804.80	2683.80	2767.20	2983.20	3097.80
PAH-GENI-3	**2299.80**	**2389.20**	2679.80	2732.40	2780.00	2918.60	3074.40
PAH-GENI-N-RS	2362.20	2464.80	2692.60	**2575.60**	**2673.40**	**2806.40**	3034.00
PAH-GENI-N	2367.80	2399.20	**2544.80**	2629.60	2702.60	2879.00	3013.40
OPT	1680.80	1902.80	1876.60	1958.40	1995.60	2050.40	2278.00

Table 6.6: Average Completion Time ($\sigma = 5 \times d_{median}$)

N	5	7	9	11	13	15	17
PAH-DP	2209.20	2141.80	2484.80	**2732.40**	**2701.20**	**2797.60**	2932.00
PAH-Chris	2209.20	**2140.80**	2512.20	2769.80	2897.00	2890.00	3007.40
PAH-GENI-3	**2206.80**	2141.80	2491.80	2773.60	2820.60	2925.00	3016.40
PAH-GENI-N-RS	2209.20	**2140.80**	2480.80	2797.80	2729.20	2811.40	**2918.00**
PAH-GENI-N	**2206.80**	2142.20	**2465.20**	2766.80	2727.80	2827.60	3041.40
OPT	1737.60	1849.00	1942.60	2048.80	2026.40	2178.40	2214.20

Table 6.7: Average Completion Time: $\sigma = 10 \times d_{median}$

Tables 6.5, 6.6, and 6.7 show the average completion time (objective value) of PAH with different scheduling algorithms under the same three release-time distributions ($\sigma = d_{\text{median}},\ 5\times d_{\text{median}},\ 10\times d_{\text{median}}$) but with a significantly larger total release-time interval. Unlike the earlier experiments, where the maximum release time was tightly constrained to 10 × minimum distance, these experiments use a broader interval to simulate more relaxed and realistic temporal settings.

Compared to earlier experimental results (Tables 6.1–6.3), the objective values (completion times) in this experiment are significantly higher in absolute value. This is expected as the wider release interval naturally leads to later request arrivals. It increases the total time needed to complete all requests, even when the scheduling decisions are efficient. However, in many cases, the relative differences between scheduling algorithms become smaller.

In Table 6.6, Table 6.5 and Table 6.7, PAH-DP still achieves the best or near-best

performance in several N, but its advantage is less consistent compared to the earlier experimental results. PAH-GENI-N-RS and PAH-GENI-N perform very closely to DP and in several cases match or outperform it.

N	5	7	9	11	13	15	17
PAH-DP	8.72	10.52	22.79	65.11	426.60	1685.65	23375.90
PAH-Chris	9.95	11.59	14.88	16.53	20.41	**24.99**	**32.83**
PAH-GENI-3	8.52	**9.69**	**12.67**	**15.87**	**19.27**	26.69	39.29
PAH-GENI-N-RS	8.74	12.09	31.80	52.47	167.61	407.31	1032.74
PAH-GENI-N	**8.42**	13.91	36.01	73.62	276.66	683.55	2329.16

(a) $\sigma = d_{median}$

N	5	7	9	11	13	15	17
PAH-DP	9.82	11.67	15.95	39.53	169.75	257.29	1770.01
PAH-Chris	10.64	14.08	18.37	23.22	22.92	27.54	**34.39**
PAH-GENI-3	**9.11**	**11.21**	14.30	**18.88**	**21.21**	**26.17**	34.52
PAH-GENI-N-RS	10.14	11.52	15.39	35.61	91.64	154.27	334.12
PAH-GENI-N	9.82	11.55	**14.18**	54.93	154.38	259.80	839.87

(b) $\sigma = 5 \times d_{median}$

N	5	7	9	11	13	15	17
PAH-DP	7.96	9.68	20.15	18.77	95.63	688.42	1679.18
PAH-Chris	9.26	11.33	16.15	23.30	25.97	**24.98**	**34.62**
PAH-GENI-3	7.99	9.35	**13.65**	**18.47**	**21.99**	27.71	35.36
PAH-GENI-N-RS	8.36	9.63	25.93	26.58	108.94	298.98	403.95
PAH-GENI-N	**7.93**	**9.10**	25.39	27.64	140.84	453.56	1059.33

(c) $\sigma = 10 \times d_{median}$

Table 6.8: Average CPU Execution Time Under Different Distributions (ms)

Figure 6.8 presents the average CPU execution time of PAH combined with different scheduling algorithms, recorded during the simulation of completion time(objective values) results in Tables 6.5, 6.6, and 6.7. Each subfigure corresponds to one of the release-time distributions: (a) $\sigma = d_{\text{median}}$, (b) $\sigma = 5 \times d_{\text{median}}$, and (c) $\sigma = 10 \times d_{\text{median}}$.

Across all settings, the overall trends are consistent with those observed in the earlier tables of CPU execution time (Table 6.4). In most columns, PAH-DP exhibits the highest CPU execution time, particularly as the number of requests N increases. However, compared to the earlier experimental results tighter release intervals, the CPU execution time of PAH-DP is noticeably reduced. This reduction happens because the scheduling events are less pressure when the time interval is longer. That is when algorithm call the scheduler, the number of requests to handle at each step is usually smaller, which makes the computation faster. PAH-Chris and PAH-GENI-3 continue to provide the fastest CPU execution times overall. Their CPU execution time remains extremely low across all values of N. PAH-GENI-N and PAH-GENI-N-RS show moderate and scalable CPU time growth. PAH-GENI-N-RS is consistently faster than PAH-GENI-N because its rescheduling ability.

All in all, As the time interval increases , the advantage of PAH-DP becomes smaller. In these more relaxed and common situations, PAH-GENI-N and PAH-GENI-N-RS offer a better balance between completion time and CPU execution time. They are able to keep the objective value (completion time) low while using

much less CPU execution time. PAH-Chris and PAH-GENI-3 are still the fastest options and work well when quick decisions are needed, but they may lead to higher objective values (completion times) in heavy-load conditions.

6.2.3 Experimental Results under Extended Number of Requests

In the previous experiments, we evaluated PAH-DP alongside PAH combined with other scheduling algorithms. Since PAH-DP is a non-polynomial time algorithm, we limited the total number of requests to around 17 to keep reasonable experiment time. To further examine the performance of the polynomial-time scheduling algorithms, we now evaluate them on larger instances. The goal is to understand how these algorithms perform under higher pressure, where both the number of requests increase significantly.

The release-time interval is extended to $[0,\ 20 \times d_{median}]$ to support larger problem instances while maintaining meaningful control over scheduling load. In the previous experiment, the interval was set to $[0,\ 10 \times d_{median}]$, where the number of requests ranged from 5 to 17. As the number of requests increases is ranging from 20 to 50 in this experiment, the previous time interval is narrow for these number of requests. If many requests are packed into a narrow window, they tend to cluster. To address this, the release-time interval needs to be lengthened to allow the request times to spread out more, which helps maintain meaningful variation in scheduling load across

different σ values. We extend the interval from $10 \times d_{median}$ to $20 \times d_{median}$ to match the growth in problem size, where the maximum number of requests increases from 17 to 50 (about a 2.9-fold increase). Instead of scaling the interval exactly by 2.9, we choose a simpler factor of 2 to maintain a balance. Extending the interval too much would lower the scheduling pressure and make the increased request count less influential. On the other hand, keeping the interval too short would lead to heavy clustering of requests. By choosing a clean factor of 2, we ensure that the distribution remains meaningful while allowing the increased number of requests to create appropriate scheduling pressure.

We test the performance of each scheduling algorithm with N (total number of requests) ranging from 20 to 50. For each value of N, five random instances are generated. The average completion time (objective value) and CPU runtime is recorded for each case. The results are presented in Table 6.9 and 6.10.

N	20	25	30	35	40	45	50
PAH-Chris	4699.80	4608.20	4881.40	5082.40	4834.40	5386.60	5234.20
PAH-GENI-3	4545.80	4534.00	4849.00	4947.00	4882.40	5356.40	5257.40
PAH-GENI-N-RS	4431.00	**4361.60**	4719.40	4880.60	4770.00	5196.40	5167.20
PAH-GENI-N	**4407.40**	4442.20	**4641.60**	**4847.60**	**4742.80**	**5138.60**	**4940.40**

Table 6.9: Average Completion Time ($\sigma = 5 \times d_{median}$)

Table 6.9 shows the average completion time (objective value) of PAH combined

with four different polynomial-time scheduling algorithms. The results show that PAH-GENI-N gives the lowest completion time (objective value) in most cases. And, PAH-GENI-N-RS often performs close to PAH-GENI-N-RS. These results show that both PAH-GENI-N and PAH-GENI-N-RS can handle larger problems efficiently. On the other hand, PAH-Chris and PAH-GENI-3 have higher completion times across all sizes tested.

N	20	25	30	35	40	45	50
PAH-Chris	44.86	**53.39**	**72.30**	**92.41**	**129.24**	**149.22**	**177.33**
PAH-GENI-3	**43.51**	55.52	92.62	168.16	246.98	314.74	397.94
PAH-GENI-N-RS	314.82	918.46	5029.05	23038.50	57189.85	65189.98	220488.27
PAH-GENI-N	604.26	1023.07	8797.16	46738.93	185142.00	198761.21	602823.66

Table 6.10: Average CPU Execution Time in ms ($\sigma = 5 \times d_{median}$)

Table 6.10 shows the average CPU execution time. These results match the experiments in Table 6.9, where the number of requests increases from 20 to 50. PAH-Chris is the fastest algorithm in most cases. Its runtime stays very low. PAH-GENI-3 is also fast, but its runtime grows more quickly than PAH-Chris as the number of requests increases. PAH-GENI-N has the highest runtime among the four algorithms. Its CPU execution time increases rapidly with N. PAH-GENI-N-RS also requires more time than PAH-Chris and PAH-GENI-3, but it is much faster than PAH-GENI-N. This shows that the rescheduling in PAH-GENI-N-RS helps reduce

computation compared to the full recomputation used in PAH-GENI-N.

Overall, in this experiment, the results show that PAH-GENI-N-RS and PAH-GENI-N maintain strong efficiency as the number of requests increases. PAH-Chris and PAH-GENI-3 are less efficient but remain highly responsive, with very low CPU execution time. In contrast, PAH-GENI-N-RS and PAH-GENI-N require significantly more computation time, especially as N grows. Although PAH-GENI-N-RS is much faster due to its rescheduling ability.

6.3 Discussing on Experimental Results

The performance of PAH with different scheduling algorithms is evaluated under two key aspects: *efficiency*, reflected by completion time (the objective value of H-OLTSP), and *responsiveness*, reflected by average CPU runtime.

These results show that load (request load) has a big effect on both efficiency and responsiveness. When the high-load condition that is many requests are released in a short time, the choice of scheduling algorithm becomes more important. In terms of efficiency, algorithms that compute better tours, like Dynamic Programming help PAH reduce travel distance. Moving on a more efficient tour leads to lower completion time.

The same pattern appears in responsiveness. Slower scheduling algorithms take much more time when the high-load condition, because they must handle more requests during each rescheduling step. For example, PAH-GENI-N becomes very slow

when there are many requests. But when the low-load condition that is requests are more spread out, this disadvantage is less noticeable. All algorithms can respond quickly in such relaxed conditions.

In summary, both efficiency and responsiveness are more affected by the choice of scheduling algorithm when scheduling pressure is high. Efficient schedulers help PAH perform better by lowering the completion time. At the same time, more efficiency scheduling algorithms are often slower, which make it become less practical such as dynamic programming. When the low-load condition, the difference between algorithms becomes smaller, especially in terms of responsiveness. Since efficiency and responsiveness cannot be achieved fully at the same time, a good scheduling choice for PAH should also consider the practical scenario. We need to decide which of the two aspects is more important in the given setting, so we can select the algorithm that performs better in that aspect.

Chapter 7

Evaluation of GTR with Different Scheduling Strategies

In this chapter, we evaluate the performance of GTR under various scheduling algorithms. The experimental setup is consistent with that used in the evaluation of PAH, combined with different scheduling strategies. We also adopt the same structures and procedures from the PAH evaluation. Section 7.1 presents the experimental results, followed by a detailed discussing in Section 7.2 based on two aspects: Responsiveness and Efficiency.

7.1 Experimental Results

In this section, we present the experimental results. In following sections, we use the names GTR-DP (GTR using dynamic programming), GTR-MST (PAH using Minimal Spanning Tree heuristic), GTR-GENI-3 (GTR using GENI with a search neighborhood of size 3), GTR-GENI-N (GTR using GENI with full tour as neighborhood) and GTR-GENI-N-RS (GTR-GENI-N with rescheduling modification). Moreover, in all tables of results, N is the number of requests and σ is the standard deviation. For consistency, this chapter employs the same load condition settings as in Chapter 6, with $\sigma = d_{median}$, $5 \times d_{median}$, and $10 \times d_{median}$ representing high-load, mid-load, and low-load conditions, respectively. Here, d_{median} denotes the median distance between locations in the metric space, based on the instance rat575 from TSPLIB [1].

We begin with the results under a short time interval setting (Section 7.1.1). In this case, the release times of requests are tightly clustered, and increasing the number of requests significantly increases the scheduling pressure. This setup represents an extreme scenario where scheduling pressure is very high. Next, we examine the results from an extended time interval setting (Section 7.1.2). This setting reflects a more practical scenario, where requests are spread more evenly over time. In this case, the scheduling pressure is generally reduced. Finally, we analyze the results for larger problem sizes (Section 7.1.3). This part of the experiment focuses on how scaling up the number of requests affects the role of the scheduling algorithm within

GTR. This allows us to observe how each algorithm handles increased complexity and whether its scheduling quality and CPU execution remain practical as the problem size grows.

7.1.1 Experimental Results under Extreme Case

Table 7.1, Table 7.2, and Table 7.3 present the average completion time under three different load conditions, generated by varying σ. This is same setting with the extreme case experiment of PAH in Chapter 6. From the three tables, we observe that under high-load and mid-load conditions, GTR-DP achieves the lowest completion time in most cases. However, when the request distribution becomes more spread out in the low-load condition, GTR-DP is no longer the best one. GTR-GENI-N and GTR-GENI-N-RS yield better results in most cases of low load condition. Across the three tables, the completion times of GTR-GENI-N and GTR-GENI-N-RS are often close to, or the same as, those of GTR-DP. In contrast, GTR-MST and GTR-GENI generally yield higher completion times than the other methods.

Table 7.4 reports the average CPU execution time for the same experiments used in the cost evaluation. Each subtable corresponds to a different load condition. As the number of requests N increases, the computation time grows for all methods, but at different rates depending on the algorithm. GTR-MST is the quickest algorithm in most cases. GTR-DP has the lowest CPU execution time for small instances but becomes impractical for large N. GTR-GENI-N and GTR-GENI-N-RS have similar

N	5	7	9	11	13	15	17
GTR-DP	**980.60**	**977.40**	**1188.80**	1247.20	**1407.00**	1406.80	**1495.40**
GTR-MST	987.40	1031.40	1329.20	1295.60	1552.20	1493.60	1707.00
GTR-GENI-3	1037.00	**977.40**	1327.80	1364.80	1541.20	1411.60	1602.80
GTR-GENI-N-RS	**980.60**	**977.40**	1221.60	**1243.80**	1426.80	**1376.60**	1522.80
GTR-GENI-N	**980.60**	**977.40**	1286.80	1247.20	1473.40	1408.60	1513.60
OPT	920.00	875.80	997.20	1099.20	1175.80	1229.40	1335.00

Table 7.1: Average Completion Time ($\sigma = d_{median}$)

N	5	7	9	11	13	15	17
GTR-DP	**1116.40**	**1061.20**	**1283.00**	**1279.00**	**1315.80**	1580.80	**1578.60**
GTR-MST	1147.20	1133.20	1391.00	1452.40	1518.40	1674.00	1778.40
GTR-GENI-3	1157.20	1165.80	1337.40	1401.00	1354.40	1730.40	1705.20
GTR-GENI-N-RS	**1116.40**	1114.40	1288.00	1297.80	1333.40	**1568.40**	1644.80
GTR-GENI-N	**1116.40**	**1061.20**	1288.00	1292.20	1319.20	1626.20	1647.80
OPT	987.20	893.40	974.60	1136.40	1169.60	1264.00	1324.40

Table 7.2: Average Completion Time ($\sigma = 5 \times d_{median}$)

CPU execution times. GTR-GENI-3 offers a good trade-off, being more scalable than DP and GENI-N, but not as fast as GTR-MST.

The average completion time and average CPU execution time tables are based on the same set of experiments. Analyzing them together enables a direct comparison

N	5	7	9	11	13	15	17
GTR-DP	858.40	1050.60	1320.20	1304.60	1340.80	1581.20	1517.60
GTR-MST	887.20	1085.80	1350.20	1388.40	1412.20	1710.20	1635.20
GTR-GENI-3	872.20	1072.40	1380.40	1378.00	1377.40	1747.00	1594.60
GTR-GENI-N-RS	889.00	1038.80	**1292.00**	**1297.20**	1342.00	1591.00	**1503.00**
GTR-GENI-N	**854.60**	**1036.20**	1320.20	1330.00	**1275.00**	**1545.60**	1523.60
OPT	772.60	940.80	991.80	1146.60	1138.00	1280.20	1229.20

Table 7.3: Average Completion Time ($\sigma = 10 \times d_{median}$)

of each algorithm's trade-off between solution quality and CPU execution time:

- GTR-DP achieves the best completion time in many scenarios. However, its CPU execution time increases dramatically with N, becoming impractical for larger instances. This makes GTR-DP suitable only when high-quality schedules are required for small sized problems or long computation time is acceptable.

- GTR-MST generally shows the lowest CPU execution, especially when problem sizes is big, making it the most efficient method computationally. However, its completion times are the highest in many scenarios, indicating poor schedule quality. This method is best suited when fast decisions are critical and some cost inefficiency is tolerable.

- GTR-GENI-3 strikes a middle ground. Its completion times are better than

GTR-MST and sometimes close to the best-performing methods. Its CPU execution time remains relatively low and scales moderately with N. This balance makes GTR-GENI-3 a good option when both speed and cost matter, but neither is critical.

- GTR-GENI-N consistently produces near-optimal completion times in most cases. However, its CPU execution time increases quickly with N, though not as drastically as GTR-DP. It is most suitable when high schedule quality is required and moderate computational cost is acceptable.

- GTR-GENI-N-RS shows similar performance to GTR-GENI-N.

7.1.2 Experimental Results under Extended Time Internal

The following tables present the average completion times under an extended time interval setting, where request release times are spread more evenly throughout the scheduling window. While the σ values remain the same (d_{median}, $5 \times d_{median}$, and $10 \times d_{median}$), this experiment differs from the earlier results by using a longer total release time span rather than a tightly packed time frame. This change reflects more relaxed conditions. Comparing to the short time interval results, average completion times increase significantly across all algorithm in this experiment. This because the maximum possible release time increased.

Table 7.5 reports the results for high-load condition under the extended interval. In most cases, the overall performance trends remain consistent: GTR-DP, GTR-

GENI-N, and GTR-GENI-N-RS achieve the best or near-best results, often with very similar completion times. GTR-GENI-3 performs competitively on small instances but becomes less reliable as N increases. GTR-MST continues to be the weakest method, producing the highest completion times in most cases. In contrast, Tables 7.6 (mid-load condition) and 7.7 (low-load condition) show that the differences among methods diminish as the load decreases, and the gap between algorithms becomes smaller.

Based on these three tables (Table 7.5, Table 7.6 and Table 7.7), we can conclude that extending the request release interval increases completion times but reduces the performance gap between algorithms. GTR-GENI-N and GTR-GENI-N-RS remain reliable across all conditions. GTR-DP still performs well, but its dominance is less pronounced than in the short-interval experiments. GTR-GENI-3 and GTR-MST become more competitive under extended intervals.

Table 7.8 reports the average CPU execution time of experiments described in the previous average completion time (objective value) tables (Table 7.5, Table 7.6 and Table 7.7). As before, each subtable corresponds to a different load condition determined by the value of σ. These results reflect the computational effort required to compute each route when requests are released over a broader time window. Compared to the earlier CPU execution time tables under short time intervals, the values here are generally lower and more stable.

All three subtables of Table 7.8 exhibit a similar trend. For each σ setting, GTR-

MST consistently achieves the lowest CPU execution time in big N cases. GTR-GENI-3 also maintains moderate CPU execution time growth across all σ values. In contrast, GTR-DP, GTR-GENI-N, and GTR-GENI-N-RS show sharp increases in CPU execution time as N grows when $\sigma = d_{median}$. Despite this, all methods run faster compared to the short interval setting. As σ increases, the differences in CPU execution time between algorithms become smaller, and the CPU execution times grow more slowly with N. This indicates that a more relaxed time distribution (larger σ) reduces computational complexity, which narrows the performance gap among all algorithms.

The extended time interval experiments reveal an important trade-off between scheduling quality and computational cost. As the release interval becomes longer, all algorithms experience higher completion times due to the increased maximum release time, but the differences in scheduling performance become less pronounced. Algorithms such as GTR-GENI-N and GTR-GENI-N-RS continue to achieve strong results across all load conditions, maintaining their status as reliable schedulers. However, their computational cost remains high, especially as N increases. GTR-DP, which was dominant in the short interval settings, still performs well but loses its clear advantage under extended intervals. Meanwhile, GTR-GENI-3 becomes more competitive, providing good cost performance with more reasonable CPU execution time, especially in mid and low-load conditions. GTR-MST remains the most efficient in terms of CPU execution time, but its scheduling quality continues to lag behind

in big instances. Overall, the results show that when the scheduling environment is more relaxed, the benefit of using complex and slower algorithms diminishes. In such settings, the trade-off favors algorithms that are faster and still reasonably efficient, making GTR-GENI-3 and even GTR-MST more practical choices compared to more computationally expensive methods like GTR-GENI-N.

7.1.3 Experimental Results under Large Number of Requests

Table 7.9 presents the average completion time for larger problem sizes, with N ranging from 20 to 50, under the low-load condition ($\sigma = 5 \times d_{median}$). Compared to the previous tables with smaller N, the overall completion times are higher, as expected, but the performance trends among algorithms remain consistent. GTR-GENI-N achieves the best results for most N values, particularly from $N = 25$ to 40, indicating strong scheduling performance as problem size increases. GTR-GENI-N-RS also performs well. GTR-GENI-3 and GTR-MST show higher average completion times for most N, similar to their behavior in smaller-scale experiments. However, in the case $N = 20$, GTR-GENI-3 achieved the lowest completion.

Table 7.10 shows the CPU execution time of the algorithms for larger N values. The results clearly highlight the differences in scalability among the methods. GTR-MST maintains stable execution time as N increases, confirming its strong scalability. GTR-GENI-3 shows moderate growth in execution time and remains computationally practical even for larger instances. In contrast, GTR-GENI-N and

GTR-GENI-N-RS exhibit steep increases in execution time, becoming significantly slower as N grows. This pattern is consistent with what was observed in the CPU execution time tables of small N, but the performance gap becomes much larger at scale. However, GTR-GENI-N-RS runs significantly faster than GTR-GENI-N as N increases, indicating that the rescheduling strategy becomes increasingly important for improving runtime at larger problem sizes.

In summary, these results for large N highlight the growing trade-off between scheduling quality and computational efficiency. GTR-GENI-N achieves the best completion times overall, with GTR-GENI-N-RS performing similarly. However, both become much slower as N increases, though GTR-GENI-N-RS scales better due to its rescheduling design. GTR-GENI-3 offers a good balance between completion time and CPU execution time. GTR-MST remains the fastest but gives higher completion times in most cases.

7.2 Discussing of Experimental Results

The performance of GTR with different scheduling algorithms was evaluated under two key aspects: efficiency (reflected by average completion time) and responsiveness (reflected by average CPU execution time).

The experimental results show that load condition is the main factor influencing both efficiency and responsiveness. In these experiments, the load conditions are controlled either by adjusting the standard deviation of release times (σ) or by

increasing the number of requests while keeping the time interval fixed. Both of these actions increase the number of active requests in a short period, making the scheduling task more demanding.

For the high-load condition, the choice of scheduling algorithm plays a more critical role. In terms of efficiency, algorithms such as GTR-DP, GTR-GENI-N and GTR-GENI-N-RS generally generate better-quality tours, resulting in lower completion times. However, these benefits come at the cost of responsiveness. Both GENI-N and GENI-N-RS become significantly slower as they must handle more requests during each rescheduling step. GTR-DP also provides strong cost performance but scales poorly in CPU execution time as the number of requests increases.

For the low-load condition, either because requests are more spread out or because there are fewer requests in total, all algorithms become faster or more responsive. The difference in completion times also becomes smaller. GTR-MST remains the fastest in all settings but produces higher costs. GTR-GENI-3 provides a good trade-off, offering acceptable completion times while maintaining relatively low run-time across different load levels.

As the number of requests grows, the trade-off between efficiency and responsiveness becomes more noticeable. Although GTR-GENI-N provide the best completion times in most cases, its run-time grows quickly and can become impractical in large instances. These patterns show that increasing the number of requests in a fixed time frame has a similar effect to increasing request load, and both make efficient

scheduling more computationally expensive.

In summary, both efficiency and responsiveness in GTR are strongly affected by request load. Higher load makes efficient algorithms slower, and faster algorithms less optimal. The scheduling strategy should be selected based on which factor is more important for the application, whether that is minimizing completion time or ensuring quick responsiveness during operation.

N	5	7	9	11	13	15	17
GTR-DP	**3.89**	**4.74**	9.49	27.99	140.87	573.22	4761.24
GTR-MST	4.25	5.31	**7.06**	**8.01**	**9.06**	**10.31**	**12.15**
GTR-GENI-3	4.32	4.85	7.72	9.63	11.98	21.23	16.51
GTR-GENI-N-RS	4.00	6.63	20.47	62.10	142.76	302.06	823.74
GTR-GENI-N	4.04	6.63	20.10	56.80	143.04	304.02	822.57

(a) $\sigma = d_{median}$

N	5	7	9	11	13	15	17
GTR-DP	**4.14**	**4.99**	8.69	23.70	146.48	758.75	3681.86
GTR-MST	4.74	5.88	**6.93**	**8.95**	**10.06**	**10.40**	**12.08**
GTR-GENI-3	4.41	5.67	7.12	9.90	11.94	14.23	17.50
GTR-GENI-N-RS	4.23	6.94	19.44	48.60	160.63	349.55	713.15
GTR-GENI-N	4.25	6.96	19.38	49.78	142.14	348.88	723.44

(b) $\sigma = 5 \times d_{median}$

N	5	7	9	11	13	15	17
GTR-DP	**3.18**	**4.72**	8.30	23.29	147.73	847.16	4190.04
GTR-MST	3.89	6.03	**6.91**	**8.00**	**8.97**	**11.05**	**10.71**
GTR-GENI-3	3.35	4.96	7.47	9.12	11.33	18.01	16.54
GTR-GENI-N-RS	3.26	6.73	17.26	47.61	142.26	371.51	762.85
GTR-GENI-N	3.45	6.71	17.29	48.02	145.74	380.64	760.79

(c) $\sigma = 10 \times d_{median}$

Table 7.4: Average CPU Execution Time Under Different Distributions (ms)

N	5	7	9	11	13	15	17
GTR-DP	**2170.20**	1935.20	**2131.80**	**2271.20**	**2216.00**	2261.40	**2451.00**
GTR-MST	2193.80	1863.80	2238.20	2355.20	2334.80	2338.40	2678.20
GTR-GENI-3	**2170.20**	1951.20	2231.60	2308.80	2268.20	2338.20	2582.40
GTR-GENI-N-RS	**2170.20**	1943.20	2132.20	2272.80	2217.80	**2243.80**	2453.00
GTR-GENI-N	**2170.20**	**1859.40**	**2131.80**	2272.00	2241.20	2270.80	2461.80
OPT	1622.40	1623.80	1762.40	1875.40	1826.60	1903.00	2056.20

Table 7.5: Average Completion Time ($\sigma = d_{median}$)

N	5	7	9	11	13	15	17
GTR-DP	**1881.20**	**2353.20**	**2213.40**	**2500.60**	**2504.20**	2571.00	2522.60
GTR-MST	**1881.20**	**2353.20**	2219.20	2502.60	2517.00	2599.00	2570.40
GTR-GENI-3	**1881.20**	**2353.20**	**2213.40**	2538.60	2547.80	2615.60	2588.80
GTR-GENI-N-RS	**1881.20**	**2353.20**	**2213.40**	**2500.60**	2524.80	**2516.80**	2531.80
GTR-GENI-N	**1881.20**	**2353.20**	**2213.40**	**2500.60**	2525.40	2570.60	**2501.20**
OPT	1693.00	2038.80	1868.60	2101.40	2156.00	2185.80	2131.80

Table 7.6: Average Completion Time ($\sigma = 5 \times d_{median}$)

N	5	7	9	11	13	15	17
GTR-DP	**2253.20**	**2156.80**	**2298.40**	2592.80	2498.60	2501.20	**2684.00**
GTR-MST	**2253.20**	2163.80	2333.40	**2583.00**	2497.80	**2434.00**	2693.20
GTR-GENI-3	**2253.20**	2189.60	**2298.40**	2599.80	**2457.40**	2574.40	2777.40
GTR-GENI-N-RS	**2253.20**	**2156.80**	**2298.40**	2592.80	2494.20	2507.20	2730.20
GTR-GENI-N	**2253.20**	**2156.80**	**2298.40**	2598.00	2492.20	2512.60	2700.00
OPT	1936.80	1777.80	1936.00	2069.40	2083.40	2060.60	2204.40

Table 7.7: Average Completion Time ($\sigma = 10 \times d_{median}$)

N	5	7	9	11	13	15	17
GTR-DP	**4.40**	**3.83**	**6.31**	11.83	34.88	141.62	1253.85
GTR-MST	5.13	4.92	7.01	**8.37**	**9.33**	**10.50**	**12.67**
GTR-GENI-3	4.44	4.06	6.94	9.11	11.14	14.39	19.20
GTR-GENI-N-RS	4.43	4.03	10.45	24.81	62.46	121.21	438.18
GTR-GENI-N	4.48	4.29	10.61	30.30	75.47	179.04	594.24

(a) $\sigma = d_{median}$

N	5	7	9	11	13	15	17
GTR-DP	4.78	**7.15**	6.71	**6.81**	**8.97**	10.93	18.86
GTR-MST	5.30	7.87	7.71	8.11	10.01	**8.70**	**10.43**
GTR-GENI-3	4.79	7.19	**6.71**	7.13	9.58	9.67	11.83
GTR-GENI-N-RS	**4.77**	7.18	6.75	7.03	13.04	21.44	37.31
GTR-GENI-N	4.84	7.36	6.76	7.03	13.61	26.31	48.64

(b) $\sigma = 5 \times d_{median}$

N	5	7	9	11	13	15	17
GTR-DP	6.52	**5.45**	**5.29**	**7.72**	6.96	20.60	12.43
GTR-MST	7.01	6.55	6.68	8.78	8.32	**9.21**	**10.36**
GTR-GENI-3	**6.49**	5.91	5.41	7.99	**6.87**	10.30	11.90
GTR-GENI-N-RS	6.50	5.53	5.33	9.19	8.60	33.62	27.72
GTR-GENI-N	6.63	5.51	5.34	9.55	8.70	43.34	28.22

(c) $\sigma = 10 \times d_{median}$

Table 7.8: Average CPU Execution Time Under Different Distributions (ms)

N	20	25	30	35	40	45	50
GTR-MST	3841.20	4045.80	4181.80	4362.00	4555.00	4602.80	4651.00
GTR-GENI-3	**3709.80**	4081.20	4214.20	4318.80	4510.60	4602.60	4524.80
GTR-GENI-N-RS	3758.80	3957.80	4154.00	4277.40	4431.20	4530.80	4422.60
GTR-GENI-N	3746.20	**3922.40**	**4075.40**	**4212.20**	**4210.20**	**4466.80**	**4337.80**

Table 7.9: Average Completion Time ($\sigma = 5 \times d_{median}$)

N	20	25	30	35	40	45	50
GTR-MST	14.25	**14.72**	**18.06**	**23.19**	**29.67**	**34.16**	**43.94**
GTR-GENI-3	**13.91**	18.56	34.50	51.20	82.08	115.38	235.25
GTR-GENI-N-RS	22.06	63.80	473.99	1672.53	2433.30	4978.89	37362.23
GTR-GENI-N	23.40	74.12	969.59	4505.78	4834.99	9341.69	94923.72

Table 7.10: Average CPU Execution Time in ms ($\sigma = 5 \times d_{median}$)

Chapter 8

Comparison between PAH and GTR

As described earlier, PAH has a lower competitive ratio than GTR when both use optimal scheduling algorithms. This means that, in theory, PAH performs better than GTR if they both optimal scheduling algorithms (such as dynamic programming). The result is based on worst-case competitive analysis, which is a common method in theoretical studies. However, it is unclear whether these theoretical results are actually reflected in practice. Real-world environments often behave differently from worst-case models.

The goal of this chapter is to evaluate whether the theoretical advantages of PAH over GTR are reflected in practice. In Section 8.1, we present the experimental results comparing PAH and GTR. All algorithms are tested on the same request

sequences and use the same scheduling algorithms. This allows us to compare their practical performance under consistent conditions. Then, in Section 8.2, we discuss the results in terms of the two main evaluation aspects: responsiveness and efficiency.

8.1 Comparison between PAH and GTR

Since the goal is to compare PAH and GTR, we focus on comparing their decision-making mechanisms. We evaluate them under the extended time interval setting, which is the same configuration used in the earlier evaluation of GTR and PAH with different scheduling algorithms. We do not include the extreme short time interval setting in this comparison because, in that case, the performance is heavily influenced by the scheduling algorithm. When the time interval is very short, the algorithm makes very few decisions, so the objective value depends more on the quality of the scheduled routes than on the algorithm's decision logic.

To focus on the decision-making aspect, we use the extended time interval setting. The configuration details and the reasons behind each setting are explained in Chapter 6, as they are the same here. In Section 8.1.1, we compare PAH, GTR and PAN using dynamic programming as the scheduling algorithm. In Section 8.1.2, we compare them using other polynomial-time scheduling algorithms. It is important to note that, in tables of results, N is the number of requests and σ is the standard deviation.

8.1.1 Experimental Results under Optimal Scheduling Algorithms

The experimental results in Tables 8.1, 8.2, and 8.3 show the average completion time of PAH and GTR under different load conditions using dynamic programming for scheduling.

N	5	7	9	11	13	15	17
PAH-DP	1949.80	**2138.80**	**2208.00**	2258.60	2397.60	2511.20	2515.20
GTR-DP	**1893.40**	2233.20	2259.20	**2148.20**	**2305.40**	**2226.60**	**2294.80**
OPT	1455.80	1610.80	1732.20	1818.40	1887.40	1901.80	1995.40

Table 8.1: Average Completion Time ($\sigma = d_{median}$)

N	5	7	9	11	13	15	17
PAH-DP	2225.60	2628.80	2461.20	2830.80	2702.20	2747.80	3008.60
GTR-DP	**2146.20**	**2284.00**	**2167.80**	**2403.20**	**2476.20**	**2330.20**	**2604.80**
OPT	1732.00	1910.80	1859.20	2026.60	2038.40	1984.60	2210.00

Table 8.2: Average Completion Time ($\sigma = 5 \times d_{median}$)

N	5	7	9	11	13	15	17
PAH-DP	2187.00	2404.80	2565.00	2736.20	2713.80	2976.60	3009.20
GTR-DP	**2027.20**	**2157.80**	**2255.80**	**2348.40**	**2482.40**	**2609.60**	**2689.00**
OPT	1679.80	1870.60	1852.60	2059.20	2051.40	2118.60	2291.00

Table 8.3: Average Completion Time ($\sigma = 10 \times d_{median}$)

In Table 8.1, representing a high load with $\sigma = d_{median}$, GTR-DP consistently achieves lower completion times than PAH-DP across big instance sizes. The difference is especially notable at larger values of N, indicating GTR's advantage in highly dynamic environments where requests arrive closely together. This suggests that GTR's immediate rescheduling strategy allows it to handle frequent requests more efficiently than PAH, which must return to the origin before rescheduling. Table 8.2 corresponds to a medium load with $\sigma = 5 \times d_{median}$. Here, GTR-DP outperforms PAH-DP across all instance size. In the low load setting of Table 8.3 with $\sigma = 10 \times d_{median}$, the advantage of GTR-DP over PAH-DP persists. GTR-DP achieves consistently lower completion times across instance sizes.

The results in the three tables (Table 8.1, Table 8.2 and Table 8.3) show that the difference between the completion times of GTR-DP and PAH-DP and the offline optimal solution is much smaller than suggested by their theoretical competitive ratios. This is because the competitive ratios are derived under worst-case analysis, while our experiments reflect more typical scenarios. In these general settings,

both algorithms achieve completion times that are closer to offline optimal than the theoretical bounds suggest, with GTR-DP generally outperforming PAH-DP.

N	5	7	9	11	13	15	17
PAH-DP	8.80	11.20	28.14	87.85	456.65	2258.81	11081.49
GTR-DP	**4.02**	**5.41**	**7.01**	**7.31**	**21.81**	**64.72**	**275.41**

(a) $\sigma = d_{median}$

N	5	7	9	11	13	15	17
PAH-DP	9.93	12.48	15.54	50.17	71.26	508.22	4930.75
GTR-DP	**5.94**	**6.67**	**5.99**	**6.96**	**8.66**	**9.32**	**70.09**

(b) $\sigma = 5 \times d_{median}$

N	5	7	9	11	13	15	17
PAH-DP	8.82	9.67	16.69	23.22	28.73	145.11	897.61
GTR-DP	**5.79**	**5.53**	**5.36**	**6.98**	**6.68**	**8.32**	**12.79**

(c) $\sigma = 10 \times d_{median}$

Table 8.4: Average CPU Execution Time Under Different Distributions (ms)

The results of CPU execution time in Table 8.4 provide insight into the computational efficiency of PAH and GTR when paired with dynamic programming as the scheduling algorithm. The CPU execution time results reveal clear differences in computational cost between PAH-DP and GTR-DP under varying load conditions. PAH-DP consistently requires much more computation time than GTR-DP, especially as the number of requests increases. Hence, these runtime results reinforce the

conclusion that PAH-DP is computationally expensive and less scalable compared to GTR-DP.

8.1.2 Experimental Results under Polynomial-time Scheduling

The experimental results presented in Tables 8.5, 8.6, and 8.7 compare the average completion time of PAH and GTR under various polynomial-time scheduling algorithms across different load levels. These results provide a broader view of how these two algorithms perform when paired with polynomial-time scheduling algorithm.

N	5	10	15	20	25	30	35	40
PAH-Chris	2163.00	2414.00	2379.60	2681.60	2985.00	3003.40	3082.20	3227.80
GTR-MST	1980.60	2306.00	2386.00	2501.60	2739.00	2806.80	2846.80	3104.20
PAH-GENI-3	2124.40	2447.40	2412.20	2773.60	2973.00	3041.80	3120.40	3191.40
GTR-GENI-3	1975.60	2247.60	2414.20	2576.80	2712.20	2866.60	2973.40	3047.00
PAH-GENI-N	2130.00	2373.40	2332.80	2578.80	2848.40	2866.40	2935.20	3081.80
GTR-GENI-N	1975.60	2160.40	2373.60	2390.00	2517.60	2674.20	2698.00	2836.00
PAH-GENI-N-RS	2129.00	2366.80	2355.40	2583.00	2840.60	2876.60	2979.60	3078.00
GTR-GENI-N-RS	1975.60	2176.20	2375.60	2399.20	2504.00	2672.40	2678.00	2826.40
OPT	1579.20	1839.00	1901.60	\	\	\	\	\

Table 8.5: Average Completion Time ($\sigma = d_{median}$)

N	5	10	15	20	25	30	35	40
PAH-Chris	2100.60	2683.40	3056.00	3171.00	3392.60	3467.20	3611.20	3604.40
GTR-MST	1945.40	2341.20	2632.80	2645.20	3073.80	3130.20	3104.00	3250.20
PAH-GENI-3	2073.40	2659.00	3101.00	3083.60	3351.60	3385.60	3560.00	3565.00
GTR-GENI-3	1945.40	2392.80	2558.40	2591.00	2998.60	3044.80	3359.60	3050.20
PAH-GENI-N	2090.60	2632.00	2947.40	3058.00	3246.00	3267.00	3410.20	3424.60
GTR-GENI-N	1945.40	2374.20	2498.60	2544.80	2848.00	2945.80	3064.60	2949.20
PAH-GENI-N-RS	2068.60	2649.80	2973.40	3075.20	3209.20	3267.60	3401.60	3462.00
GTR-GENI-N-RS	1945.40	2342.00	2489.20	2584.60	2836.60	2919.80	3065.60	3004.20
OPT	1700.60	2029.80	2178.40	\	\	\	\	\

Table 8.6: Average Completion Time ($\sigma = 5 \times d_{median}$)

Generally, GTR tends to achieve lower completion times than PAH across most problem sizes and scheduling methods, especially as the number of requests grows. However, there are instances where PAH slightly outperforms GTR under certain scheduling algorithms. This indicates that the advantage of GTR is not uniform across all scenarios. Overall, GTR generally shows better completion times then PAH.

The table 8.8 shows the results of CPU execution time between GTR and PAH with different polynomial time scheduling algorithms. The results show that PAH takes much longer CPU execution time than GTR, especially as the problem size increases. This is most noticeable when using complex scheduling heuristics like

N	5	10	15	20	25	30	35	40
PAH-Chris	2294.80	2738.40	2636.60	3034.40	3305.60	3411.80	3569.80	3597.20
GTR-MST	2228.80	2646.80	2360.60	2743.40	2934.40	2830.20	3215.20	3164.80
PAH-GENI-3	2282.00	2764.80	2679.60	3123.80	3320.20	3395.00	3558.60	3664.00
GTR-GENI-3	2227.00	2629.80	2442.60	2777.60	2906.00	2923.20	3178.60	3229.00
PAH-GENI-N	2282.00	2718.20	2592.60	3106.20	3152.20	3299.00	3423.40	3411.60
GTR-GENI-N	2227.00	2629.80	2361.20	2579.00	2707.00	2751.60	3073.80	2944.20
PAH-GENI-N-RS	2282.00	2654.20	2650.40	3129.00	3194.80	3254.20	3459.00	3451.80
GTR-GENI-N-RS	2227.00	2629.80	2346.80	2582.20	2762.40	2780.80	3019.60	3041.00
OPT	1796.60	2077.60	1911.80	\	\	\	\	\

Table 8.7: Average Completion Time ($\sigma = 10 \times d_{median}$)

GENI-N and GENI-N-RS, where CPU execution runtime of PAH rises sharply. The high cost comes from PAH frequently returning to the origin, which triggers expensive full-route rescheduling with many pending requests. In contrast, GTR has much lower execution time across all algorithms and load levels. It reschedules immediately from its current position instead of delaying like PAH, which leads to lower scheduling complexity.

Overall, these results of CPU execution emphasize that while the computational demands of PAH limit its practical use for large-scale problems. GTR offers a more scalable alternative with lower CPU execution time, making it more suitable for real-time applications with complex scheduling needs.

N	5	10	15	20	25	30	35	40
PAH-Chris	0.04	0.08	0.11	0.19	0.27	0.36	0.53	0.71
GTR-MST	0.03	0.04	0.05	0.07	0.09	0.11	0.13	0.18
PAH-GENI-3	0.04	0.07	0.13	0.24	0.46	0.76	1.05	1.68
GTR-GENI-3	0.02	0.04	0.07	0.11	0.23	0.36	0.48	0.69
PAH-GENI-N	0.04	0.32	3.82	21.12	119.07	471.03	940.54	3120.63
GTR-GENI-N	0.02	0.07	0.86	4.19	31.55	116.58	342.17	828.79
PAH-GENI-N-RS	0.04	0.27	1.90	9.21	40.13	145.27	257.04	727.28
GTR-GENI-N-RS	0.02	0.07	0.71	2.80	15.72	52.69	129.03	319.44

(a) $\sigma = d_{median}$

N	5	10	15	20	25	30	35	40
PAH-Chris	0.05	0.10	0.15	0.21	0.29	0.38	0.45	0.56
GTR-MST	0.03	0.04	0.05	0.06	0.08	0.10	0.14	0.16
PAH-GENI-3	0.04	0.08	0.13	0.23	0.41	0.53	0.89	1.13
GTR-GENI-3	0.03	0.03	0.05	0.10	0.14	0.27	0.51	0.64
PAH-GENI-N	0.04	0.11	1.61	8.38	49.12	148.94	504.08	1051.19
GTR-GENI-N	0.03	0.03	0.08	1.00	2.21	25.16	91.63	177.38
PAH-GENI-N-RS	0.04	0.11	0.77	4.30	17.59	64.69	177.42	369.69
GTR-GENI-N-RS	0.03	0.04	0.08	0.65	1.23	12.76	41.11	87.12

(b) $\sigma = 5 \times d_{median}$

N	5	10	15	20	25	30	35	40
PAH-Chris	0.05	0.10	0.14	0.20	0.28	0.38	0.43	0.62
GTR-MST	0.03	0.04	0.04	0.06	0.09	0.09	0.12	0.15
PAH-GENI-3	0.04	0.10	0.14	0.24	0.40	0.65	0.76	1.39
GTR-GENI-3	0.03	0.03	0.04	0.08	0.18	0.19	0.38	0.71
PAH-GENI-N	0.04	0.09	0.98	7.64	74.71	193.75	424.47	1547.61
GTR-GENI-N	0.03	0.03	0.10	0.94	8.69	5.82	30.81	142.68
PAH-GENI-N-RS	0.04	0.10	0.75	4.81	34.41	75.30	189.42	509.49
GTR-GENI-N-RS	0.03	0.03	0.08	0.61	5.43	3.08	12.69	54.86

(c) $\sigma = 10 \times d_{median}$

Table 8.8: Average CPU Execution Time Under Different Distributions (ms)

8.2 Discussion on Experimental Results

Based on the experimental results, we discuss the performance of PAH and GTR in terms of two key criteria: efficiency and responsiveness. Across all tested scheduling algorithms and load conditions, GTR generally performs better than PAH, especially as the problem size increases.

In terms of efficiency, GTR usually achieves lower completion times than PAH under the same scheduling algorithm. This difference becomes more noticeable under larger problem sizes. The return-to-origin rule of PAH delays rescheduling and causes extra travel, which increases the total completion time. While the difference is smaller under low-load condition or small N, the trend remains: GTR is more efficiency than PAH in solving H-OLTSP problems.

In terms of responsiveness, GTR shows much lower CPU execution time than PAH. PAH always returns to the origin before rescheduling, which leads to schedule for more requests at once. This causes its CPU execution time to grow quickly as N increases, especially with complex heuristics like GENI-N and GENI-N-RS. GTR, by updating the route immediately at the current location, handles fewer requests per reschedule and maintains shorter execution time even for large instances.

When compared with the theoretical results, both algorithms perform better than what their competitive ratios suggest. Our experimental results also show that the theoretical results do not reflect practical performance, since the worst-case bounds overestimate the gap from the offline optimum. In the tested instances, which are

small in size and generated from normal distributions, the completion times are much closer to optimal than the theoretical results suggested. Overall, the results indicate that both algorithms are more effective in practice than theory alone implies in small-sized cases.

In summary, the experiments demonstrate that GTR generally provides better efficiency and responsiveness than PAH in practical settings. In only a few cases, PAH achieves slightly lower completion times, but GTR generally requires much less CPU execution time. This highlights the responsiveness advantage of GTR, which is particularly important for online scheduling. Overall, these results indicate that GTR is well-suited for solving H-OLTSP in real-world applications where both efficiency and responsiveness are essential.

Chapter 9

Conclusion

This thesis evaluated the practical performance of PAH and GTR algorithms for the H-OLTSP problem through an empirical study conducted in a controlled simulation environment. The results showed that load strongly influenced both responsiveness and efficiency in H-OLTSP algorithms. Under high-load condition, complex strategies like DP produced shorter routes and better efficiency, confirming theoretical expectations: PAH-DP outperformed PAH-Chris, and GTR-DP outperformed GTR-MST. However, this comes at the cost of higher computation time. Under low-load condition, the efficiency gap narrows and simpler methods perform similarly. The results also showed that practical strategies like GENI and its variants balance responsiveness and efficiency well. GENI-3 performed close to Chris or MST, while GENI-N improved efficiency and could match DP but with high runtime. GENI-N-RS reduced this cost by skipping some rescheduling, keeping similar efficiency with

better responsiveness. These patterns hold for both PAH and GTR, making GENI-N-RS a strong choice for large-scale H-OLTSP. Finally, our experiments show that GTR often performs better than PAH in practice. Moreover, for the small-sized instances tested, both algorithms achieved completion times closer to the offline optimal than suggested by their theoretical results. This work has bridged the gap between theoretical results and practical performance.

9.1 Limitations

While this thesis provides a detailed evaluation of online algorithms for the H-OLTSP, several limitations should be acknowledged to clarify the scope of the work and the context of the results. First of all, the experiments were conducted in a simulated environment using synthetic requests on the discrete metric space rat575 from TSPLIB. While this allows controlled evaluation, it may not capture real-world factors like road geometry or continuous movement, so the results may not fully generalize to physical systems. Secondly, this thesis focuses only on H-OLTSP, where the server must return to the origin. Other variants like N-OLTSP or those with time windows are not considered, so the conclusions may not apply to those settings.

These limitations do not reduce the value of the findings but highlight that this work focuses on a specific problem scope under controlled assumptions. The findings offer a foundation for future empirical research and can help guide broader extensions of OLTSP algorithms.

9.2 Future Work

This thesis offers a practical evaluation of online algorithms for H-OLTSP, with several directions for future work. First, other OLTSP variants such as those with time windows or deadlines could be explored to generalize the results. Second, using real-world road networks instead of discrete TSPLIB data would make the evaluation more realistic. Third, adaptive or learning-based schedulers could be studied, especially for applications like warehouse automation where historical data is available. Finally, combining or switching between scheduling strategies based on instance conditions may improve performance across diverse scenarios. These extensions would deepen understanding of OLTSP and enhance practical applicability.

Appendix A

Pseudocode and Descriptions of Algorithms

A.1 Dynamic Programming

We assume TSP instances are defined on complete, weighted, metric graphs, $G = (V, E)$, where V is a set of n vertices and E is the set of edges. The distance function $d : V \times V \to \mathbb{Z}^+$ assigns a positive integer weight to each edge, satisfying the triangle inequality. Moreover, we assume vertex $v_0 \in V$ is the designated origin. The objective is to find a minimum-cost tour that starts at vertex v_0, visits each other vertex in $V \setminus \{v_0\}$ exactly once, and returns to v_0. Bellman [18] defined a recursive cost function $f(v_i; S)$:

- $v_i \in V$ is the current vertex,
- $S \subseteq V \setminus \{v_0, v_i\}$ is the set of remaining unvisited vertices,
- $f(i; S)$ is the minimal cost of a path that starts at vertex v_i, visits every vertex in S exactly once, and ends at vertex v_0.

The dynamic programming algorithm is based on the principle of optimality, which is that the optimal tour from the current state must include the optimal tour from any of its next states. The recurrence relation is given by:

$$f(v_i; S) = \min_{v_j \in S} (d(v_i, v_j) + f(v_j; S \setminus \{v_j\}))$$

The base case is:

$$f(v_i; \emptyset) = d(v_i, v_0)$$

This means that if there are no remaining vertices to visit, the cost is simply the distance from the current vertex back to the start vertex v_0. This formulation allows the problem to be solved recursively for increasing sizes of subsets, with memoization to avoid recomputation. The total time complexity is $O(n^2 \cdot 2^n)$ and the space complexity is $O(n \cdot 2^n)$. While exponential, this is practical for instances with around 17 vertices.

Based on Bellman's work [18], the dynamic programming steps are described below:

- **Step 1: Initialization.** The algorithm begins by computing the base cases for each vertex $v_i \in V \setminus \{v_0\}$ with an empty set of unvisited vertices. For each v_i, we set:

  $$f(v_i; \emptyset) = d(v_i, v_0)$$

  This represents the cost of completing the tour directly from v_i back to the origin when there are no other vertices left.

- **Step 2: Computing Small Subproblems.** Next, the algorithm computes $f(v_i; S)$ for all subsets S of size 1. For each pair of distinct vertices v_i and v_j, it compute:

  $$f(v_i; \{v_j\}) = d(v_i, v_j) + f(v_j; \emptyset) = d(v_i, v_j) + d(v_j, v_0)$$

  This represents the cost of going from i to j, and then returning to 0.

- **Step 3: Recursively Computing All Subproblems.** For each larger subset S (with size $|S| \geq 2$), and for every vertex $v_i \notin S \cup \{v_0\}$, the algorithm compute:

$$f(v_i; S) = \min_{v_j \in S} \left(d(v_i, v_j) + f(v_j; S \setminus \{v_j\}) \right)$$

  This means that the cost of completing the tour from v_i through S is the minimum over all choices of a next vertex v_j, where the tour proceeds to v_j and then continues optimally through the remaining vertices in $S \setminus \{v_j\}$.

- **Step 4: Calculating Final Result.** After all subproblems are solved, the total minimum cost of TSP tour starting and ending at vertex v_0 is given by:

$$f(v_0; V \setminus \{v_0\})$$

  This is the minimum cost of starting at the origin, visiting all other vertices, and returning to the origin.

- **Step 5: Tour Construction.** Once the minimal cost has been computed via dynamic programming, the actual optimal tour can be reconstructed by backtracking through the choices made during the recursive computation. At each step, for a given state $(v_i; S)$, we record which vertex $v_j \in S$ achieved the minimum in:

$$f(v_i; S) = \min_{v_j \in S} \left(d(v_i, v_j) + f(v_j; S \setminus \{v_j\}) \right)$$

  By storing this decision (i.e., the selected next vertex) in a separate table during the forward computation, we can reconstruct the full tour in reverse

order, starting from the initial state $(0; V \setminus \{0\})$ and iteratively following the recorded transitions until reaching the base case.

Appendix C.1 shows a detailed example of dynamic programming for solving TSP.

A.2 GENI

Algorithm 1 Type-I Insertion

Inputs: Current tour T, new point v, distance function d and neighborhood N_p

$T_{best} \leftarrow []$ ▷ Initialize the best tour with a empty list
$C_{best} \leftarrow +\infty$ ▷ Initialize the best cost with positive infinity
for each pair of vertices $(v_i, v_j) \in N_p(v)$ **do**
  for each vertex $v_k \in N_p(v_i + 1)$ on the path $(v_j \rightarrow v_i)$ **do**
    if $v_k \neq v_i$ and $v_k \neq v_j$ **then**
      ▷ Inserting v between v_i and v_j with v_k as an intermediary on T.
      $T' \leftarrow$ copy T
      $T' \leftarrow T'$ removes edges (v_i, v_{i+1}), (v_j, v_{j+1}), and (v_k, v_{k+1})
      $T' \leftarrow T'$ adds edges (v_i, v), (v, v_j), (v_{i+1}, v_k), and (v_{j+1}, v_{k+1})
      $T' \leftarrow T'$ reverses paths $(v_{i+1}, \ldots, v_j)$ and $(v_{j+1}, \ldots, v_k)$
      if The Cost of $T' \leq C_{best}$ **then**
        $T_{best} \leftarrow T'$
        $C_{best} \leftarrow$ the cost of T_{best}
      end if
    end if
  end for
end for

Output: a TSP tour T_{best}

Algorithm 2 Type-II Insertion

Inputs: Current tour T, new point v, distance function d and neighborhood N_p

$T_{best} \leftarrow []$ ▷ Initialize best tour with a empty tour
$C_{best} \leftarrow +\infty$ ▷ Initialize best cost with positive infinity
for each pair of vertices $(v_i, v_j) \in N_p(v)$ **do**
  for each vertex $v_k \in N_p(v_i + 1)$ on the path $(v_j \rightarrow v_i)$ **do**
    for each vertex $v_l \in N_p(v_j + 1)$ on the path $(v_i \rightarrow v_j)$ **do**
      if $v_k \neq v_j$ and $v_k \neq v_{j+1}$ and $v_l \neq v_i$ and $v_l \neq v_{i+1}$ **then**
        ▷ Inserting v between v_i and v_j with v_k and v_l as intermediaries on T
        $T \leftarrow$ copy T
        $T' \leftarrow T'$ removes (v_i, v_{i+1}), (v_{l-1}, v_l), (v_j, v_{j+1}), and (v_{k-1}, v_k)
        $T' \leftarrow T'$ adds (v_i, v), (v, v_j), (v_l, v_{j+1}), (v_{k-1}, v_{l-1}) and (v_{i+1}, v_k)
        $T' \leftarrow T'$ reverse paths $(v_{i+1}, \ldots, v_{l-1})$ and $(v_l, \ldots, v_j)$
        if The Cost of $T' \leq C_{best}$ **then**
          $T_{best} \leftarrow T'$
          $C_{best} \leftarrow$ the cost of T_{best}
        end if
      end if
    end for
  end for
end for

Output: a TSP tour T_{best}

Algorithm 3 GENI

Inputs: Distance function: d, Origin o, Points to visit: $P = \{p_1, p_2, p_3.....\}$

Initialize a tour T with an arbitrary subset of three points
Initialize the p-neighborhood N_p of each point in P
while there exists a point $p \in P$ that is not in the tour T **do**
    Select a point P that is not yet visited by the tour T
    if $|T| < 6$ **then**
        $T \leftarrow$ Type-I Insertion with T, p, d and N_p as inputs.
    else
        $T_{\text{type-I}} \leftarrow$ Type-I Insertion with T, p, d and N_p as inputs.
        $T_{\text{type-II}} \leftarrow$ Type-II Insertion with T, p, d and N_p as inputs.
        $T \leftarrow$ one of $T_{\text{type-I}}$ and $T_{\text{type-II}}$ with lower cost.
    end if
    update the p-neighborhood N_p of each point in P
end while

Output: a TSP tour T

Appendix B

Definitions

B.1 Competitive Ratio

Competitive analysis is a standard method for evaluating the worst-case theoretical performance of online algorithms. As we described, an input is revealed piece by piece, and the algorithm must make decisions without knowing the future pieces of the input in an online problem. The main idea of competitive analysis is to compare the performance of an online algorithm to an optimal offline algorithm. The offline algorithm, which is often referred to as an *adversary*, knows the entire input in advance and can make the best possible decisions. The competitive ratio is the key measure used in competitive analysis to compare the performance of an online algorithm to the optimal offline solution. We define it in Definition 7 based on the textbook by Borodin et al. [2].

Definition 7. *Competitive Ratio*

Let ALG be an online algorithm and OPT be an optimal offline algorithm. Suppose that both algorithms solve the same problem in the same input sequence σ. The cost of ALG on σ is indicated by $ALG(\sigma)$ and the cost of OPT is denoted by $OPT(\sigma)$. There are two cases of online problems:

- ***Minimization problems:*** *The algorithm ALG is* c-competitive *if there exists a constant $\alpha \geq 0$ such that for all input sequences σ:*

$$ALG(\sigma) \leq c \cdot OPT(\sigma) + \alpha.$$

- ***Maximization problems:*** *The algorithm ALG is* c-competitive *if there exists a constant* $\alpha \geq 0$ *such that for all input sequences* σ*:*

$$ALG(\sigma) \geq \frac{1}{c} \cdot OPT(\sigma) - \alpha.$$

In both cases, the value c *is called the* competitive ratio *of the algorithm ALG.*

For a c-competitive algorithm, a lower value of c means that the algorithm performs closer to optimal. In other words, a smaller value of c means better performance. This framework helps to analyze the worst-case performance of online algorithms.

B.2 Approximation Ratio

The approximation ratio is a way to measure how good an approximation algorithm is. An approximation algorithm is an algorithm that approximates an optimal solution and often designed for NP-Hard optimization problems where finding the an optimal solution requires exponential time, in the worst case. These algorithms give solutions that are close to the best possible answer. To measure the quality of the solution, we compare the result of the approximation algorithm to the result of an optimal algorithm that always gives the best solution. We define it in Definition 8 based on the textbook by Williamson et al. [19].

Definition 8. *Approximation Ratio*

Let ALG be an approximation algorithm and OPT be an optimal algorithm. Suppose that both algorithms solve the same problem in the same input sequence σ. The cost of ALG on σ is indicated by $ALG(\sigma)$ and the cost of OPT on σ is denoted by $OPT(\sigma)$. There are two cases of problems.

- ***Minimization problems:*** *The algorithm is called α-approximation if for all inputs σ:*

$$ALG(\sigma) \leq \alpha \cdot OPT(\sigma).$$

- ***Maximization problems:*** *The algorithm is called α- approximation algorithm if for all inputs σ,*

$$ALG(\sigma) \geq \frac{1}{\alpha} \cdot OPT(\sigma).$$

In both cases, the value α is called the approximation ratio *of the algorithm ALG.*

For α-approximation algorithms, a smaller α means the output of algorithm is closer to the best possible solution. This measure helps us understand how much quality we trade off for faster computation. Here we introduce the approximation ratio because it can help us understand the theoretical performance of scheduling algorithms that describes in later chapters.

B.3 Scheduling Definitions

We introduce problems here to help you understand the scheduling strategies we will mention in the later chapters. In the OLTSP, the servers have to schedule routes

to visit requested locations. Usually, they input the list of requested locations into such problems and then use the output of the problems as the route. In this section, we describe the scheduling problems associated with both versions of OLTSP, even though this thesis primarily focuses on the H-OLTSP. The purpose is to explain how scheduling challenges differ between the two versions, which helps in understanding the algorithms reviewed in Chapter 2. The formal definition of scheduling problems is given. We define these problems on graphs following the definitions presented in the textbook [20].

Definition 9. *TSP*

Inputs:

- *a complete undirected graph* $G = (V, E)$ *where* V *is list of vertices and* E *are the list of edges.*
- *a cost function* $c : V \times V \mapsto \mathbb{Z}^+$.

This problem aims to find a tour *that starts at a vertex, visits all other vertices of* V*, and then returns to the starting vertex while minimizing the cost of the tour.*

Definition 10. *Hamiltonian Path*

Inputs:

- *a complete undirected graph* $G = (V, E)$ *where* V *is list of vertices and* E *are the list of edges.*

- *a cost function* $c : V \times V \mapsto \mathbb{Z}^+$.

This problem aims to find a path *that visits all the vertices of* V *while minimizing the total cost of the path.*

As for the OLTSP, the Hamiltonian path is used with specific constraints, such as a fixed start or fixed end. This is because the path must begin at the current location of the server and, in the homing version, must also end at the origin. To properly handle these requirements, we introduce the following definitions for variations of the Hamiltonian path that incorporate these structural constraints.

Definition 11. *Hamiltonian Path with Fixed Start* s

Inputs:

- *a complete undirected graph* $G = (V, E)$ *where* V *is list of vertices and* E *are the list of edges.*
- *a cost function* $c : V \times V \mapsto \mathbb{Z}^+$.
- *a start vertex* s *where* $s \in V$

This problem aims to find a path *that starts at* s*, and visits all the vertices of* $V \setminus \{s\}$ *exactly once while minimizing the total cost of the path.*

A feasible solution to this problem is notated as $HP_s(G, c)$ *that means a path starts from* s *and visits all other vertices of* G *exactly once*

Definition 12. *Hamiltonian Path with Fixed Endpoints (Start* v_s *and End* v_t*)*

Inputs:

- *a complete undirected graph* $G = (V, E)$ *where* V *is list of vertices and* E *are the list of edges.*
- *a cost function* $c : V \times V \mapsto \mathbb{Z}^+$.
- *a start vertex* s *where* $v_s \in V$
- *a end vertex* t *where* $v_t \in V$

This problem aims to find a path *that starts at* s*, and visits all the vertices of* $V \backslash \{s, t\}$ *exactly once, then return to* t *while minimizing the total cost of the path.*

A feasible solution to this problem is notated as $HP_{v_s \to v_t}(G, c)$ *means that a path starts from* v_s *, ends at* v_t*, and visits all other vertices exactly once in between.*

There are special cases of the Hamiltonian path with specific constraints that need to be mentioned. We assume a graph $G = (V, E)$ and a cost function $c : e \mapsto \mathbb{Z}^+$ where $\forall e \in E$. We suppose a vertex $v_i \in V$. The feasible solution $HP_{v_i \to v_i}(G)$ is also a feasible solution to TSP on inputs (G, c). In addition, $\forall v_i, v_j \in V$, $HP_{v_i}(G)$ and $HP_{v_i \to v_j}(G)$ are feasible solutions to the standard Hamiltonian path problem with input (G, c).

B.4 Background Definitions

We introduce key concepts to help explain the scheduling algorithms, following the textbook by Williamson et al. [19].

Definition 13. *Minimal Cost Perfect Matching*

Inputs:

- *a finite undirected graph* $G = (V, E)$.
- *a list of vertices* $V' \subset V$ *and* $|V'|$ *is even.*
- *a cost function* $c : e \mapsto \mathbb{Z}^+$ *for* $\forall e \in E$.

The minimal perfect matching is a subset of edges

$$M \subset E$$

such that every vertex of V' *is incident to exactly one edge in* M. *In addition,* M *minimizes total cost.*

Definition 14. *Minimal Spanning Tree (MST)*

Inputs:

- *a finite undirected graph* $G = (V, E)$.
- *a cost function* $c : e \mapsto \mathbb{Z}^+$ *for* $\forall e \in E$.

The MST is a tree (acyclic and connect) $T = (V, E_t)$ *that spans all vertices of* V *with minimizing cost where* $E_t \subseteq E$.

Definition 15. *Eulerian Circuit or Traversal*

Inputs:

- *a finite undirected graph* $G = (V, E)$.

The Eulerian Circuit is a tour that starts and ends at same vertex $v \in V$ *and traverses every edge of* E *exactly once without repetition.*

Appendix C

Examples

C.1 Example: Solving TSP with Dynamic Programming

We apply the dynamic programming algorithm that is proposed by Bellman [18] on a complete metric graph with four vertices: $V = \{v_0, v_1, v_2, v_3\}$. The cost matrix $C = [d(i, j)]$ is defined as:

$$C = \begin{bmatrix} 0 & 4 & 6 & 7 \\ 4 & 0 & 5 & 8 \\ 6 & 5 & 0 & 3 \\ 7 & 8 & 3 & 0 \end{bmatrix}$$

Vertex v_0 is fixed as the starting and ending vertex. The goal is to find the minimum-cost tour starting at vertex v_0, visiting each of the other vertices exactly once, and returning to v_0.

Step 1: Base Case ($|S| = 0$)

$$f(v_1; \emptyset) = d(1, 0) = 4, \quad f(v_2; \emptyset) = 6, \quad f(v_3; \emptyset) = 7$$

Step 2: Subsets of Size 1 ($|S| = 1$)

$$f(v_1; \{v_2\}) = d(1,2) + f(2; \emptyset) = 5 + 6 = 11$$

$$f(v_1; \{v_3\}) = 8 + 7 = 15$$

$$f(v_2; \{v_1\}) = 5 + 4 = 9$$

$$f(v_2; \{v_3\}) = 3 + 7 = 10$$

$$f(v_3; \{v_1\}) = 8 + 4 = 12$$

$$f(v_3; \{v_2\}) = 3 + 6 = 9$$

Step 3: Subsets of Size 2 ($|S| = 2$)

$$\begin{aligned} f(v_1; \{v_2, v_3\}) &= \min\{d(1,2) + f(v_2; \{v_3\}),\ d(1,3) + f(v_3; \{v_2\})\} \\ &= \min\{5 + 10,\ 8 + 9\} = \min\{15,\ 17\} = 15 \\ f(v_2; \{v_1, v_3\}) &= \min\{5 + 15,\ 3 + 12\} = \min\{20,\ 15\} = 15 \\ f(v_3; \{v_1, v_2\}) &= \min\{8 + 11,\ 3 + 9\} = \min\{19,\ 12\} = 12 \end{aligned}$$

Step 4: Final Computation

$$\begin{aligned} f(v_0; \{v_1, v_2, v_3\}) &= \min\{d(0,1) + f(v_1; \{v_2, v_3\}), d(0,2) + f(v_2; \{v_1, v_3\}), d(0,3) + f(v_3; \{v_1, v_2\})\} \\ &= \min\{4 + 15,\ 6 + 15,\ 7 + 12\} = \min\{19,\ 21,\ 19\} = 19 \end{aligned}$$

Step 5: Reconstructing the Optimal Path

Assume we choose the first minimal option: $f(v_0; \{v_1, v_2, v_3\}) = 4 + f(v_1; \{v_2, v_3\})$.

- From $f(v_1; \{v_2, v_3\}) = 15$, the better option is to go to v_2: $5 + f(v_2; \{v_3\}) = 5 + 10 = 15$.
- From $f(v_2; \{v_3\}) = 10$, go to v_3, then return to v_0.

So the path is:

$$v_0 \to v_1 \to v_2 \to v_3 \to v_0$$

with total cost $4 + 5 + 3 + 7 = 19$, which matches the computed optimal cost.

C.2 Example: Solving TSP with Christofides Heuristic

We explain the Christofides heuristic that is proposed by Christofides [12] on solving a TSP instance. We assume a TSP instance (G, c) where $G = (V, E)$ is a graph and c is the cost function that assigns a weight to each edge $e \in E$. The instance is shown in Figure C.1 (a), with edge costs labeled directly on the graph.

The first step of the algorithm is to compute a MST of G, which is shown in Figure C.1 (b). Next, the algorithm finds a minimum-cost perfect matching on the set of vertices with odd degree in the MST. In this example, all vertices have odd degree, so the matching includes all of them. The resulting matching is shown in Figure C.1 (c).

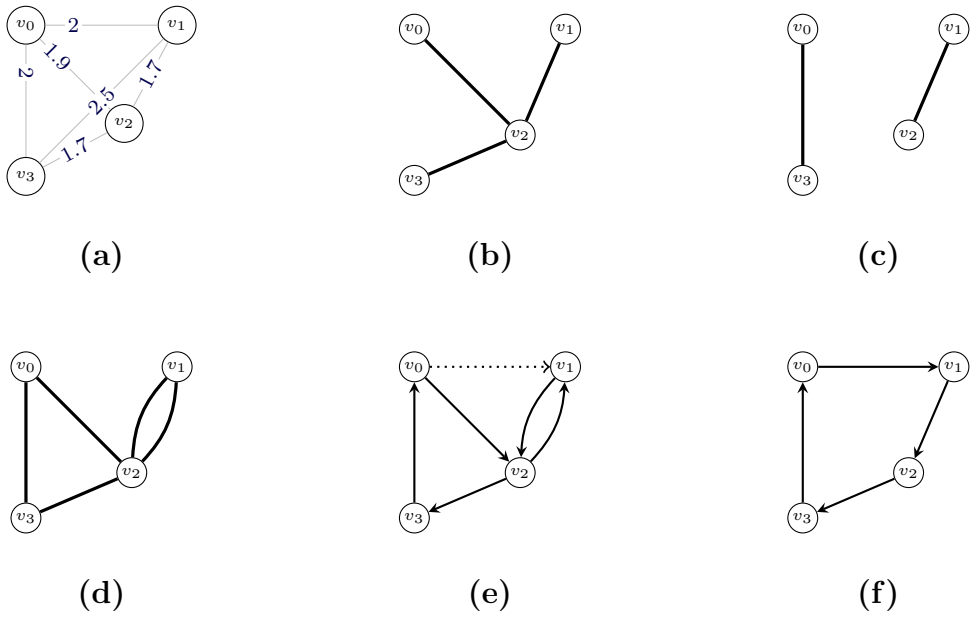


Figure C.1: Example of Christofides Heuristic

Then, the MST and the matching are merged to form a multigraph, which is shown in Figure C.1 (d). An Eulerian circuit of this multigraph is computed and illustrated with black arrows in Figure C.1 (e). In this case, the subpath $v_0 \to v_2 \to v_1$ can be shortcut to $v_0 \to v_1$, as shown by the dotted line. Finally, after applying the shortcutting, we obtain a TSP tour. This tour is shown in Figure C.1 (f).

Bibliography

[1] "TSPLIB: A traveling salesman problem library," http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html, accessed: 2025-06-11.

[2] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge, UK: Cambridge University Press, 1998.

[3] G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo, "Algorithms for the on-line travelling salesman," *Algorithmica*, vol. 29, no. 4, pp. 560–581, 2001. [Online]. Available: https://link-springer-com.uml.idm.oclc.org/article/10.1007/s004530010071

[4] A. Bjelde, J. Hackfeld, Y. Disser, C. Hansknecht, M. Lipmann, J. Meißner, M. SchlÖter, K. Schewior, and L. Stougie, "Tight bounds for online tsp on the line," *ACM Trans. Algorithms*, vol. 17, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3422362

[5] M. Blom, S. O. Krumke, W. E. D. Paepe, and L. Stougie, "The online tsp

against fair adversaries," *INFORMS Journal on Computing*, vol. 13, no. 2, pp. 138–148, 2001.

[6] E. Bampis, B. Escoffier, N. Hahn, and M. Xefteris, "Online tsp with known locations," in *Algorithms and Data Structures*, P. Morin and S. Suri, Eds. Cham: Springer Nature Switzerland, 2023, pp. 65–78.

[7] P.-C. Chen, E. D. Demaine, C.-S. Liao, and H.-T. Wei, "Waiting is not easy but worth it: the online tsp on the line revisited," 2019.

[8] J.-W. Yeh, H.-T. Wei, E. D. Demaine, and C.-S. Liao, "Online traveling salesmen should travel randomly: A randomized zealous algorithm for the online tsp on the line," https://doi.org/10.2139/ssrn.4903580, 2024, sSRN working paper.

[9] T. Gouleakis, K. Lakis, and G. Shahkarami, "Learning-augmented algorithms for online tsp on the line," 2022, accessed: Jan. 25, 2024. [Online]. Available: http://arxiv.org/abs/2206.00655

[10] H.-Y. Hu, H.-T. Wei, M.-H. Li, K.-M. Chung, and C.-S. Liao, "Online tsp with predictions," 2022, preprint. Published on arXiv, Jun. 30, 2022. [Online]. Available: http://arxiv.org/abs/2206.15364

[11] E. Bampis, S. Kamali, S. Nazarian, and A. Tigkas, "Learning-augmented online tsp on rings, trees, flowers and (almost) everywhere else," 2023, accessed: Jan. 25, 2024. [Online]. Available: http://arxiv.org/abs/2305.02169

[12] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," *Operational Research Forum*, vol. 3, no. 1, p. 20, Mar. 2022.

[13] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, Apr. 1973.

[14] M. Gendreau, A. Hertz, and G. Laporte, "New insertion and postoptimization procedures for the traveling salesman problem," *Operations Research*, vol. 40, no. 6, pp. 1086–1094, Dec. 1992.

[15] S. Basu, "Tabu search implementation on traveling salesman problem and its variations: A literature survey," *American Journal of Operations Research (AJOR)*, vol. 2, no. 2, pp. 163–173, 2012.

[16] H. Xu, Y. Ge, and G. Zhang, "Genetic algorithm for traveling salesman problem," in *Proceedings of the 2022 5th International Conference on Computational Intelligence and Intelligent Systems (CIIS)*. ACM, Nov. 2022, pp. 33–40.

[17] C. C. Skiścim and B. L. Golden, "Optimization by simulated annealing: A preliminary computational study for the tsp," in *Proceedings of the 15th Conference on Winter Simulation - Volume 2*, ser. WSC '83. IEEE Press, 1983, p. 523–535.

[18] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *Journal of the ACM (JACM)*, vol. 9, no. 1, pp. 61–63, Jan. 1962.

[19] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*, 1st ed. Cambridge University Press, 2011.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, Massachusetts London, England: MIT Press, 2009.