The Dissertation Committee for Patrick Alexander Mannon
certifies that this is the approved version of the following dissertation:

Partitioning Methods for NP-Hard Routing Problems

Committee:

Stephen Boyles, Supervisor

John Hasenbein

Erhan Kutanoglu

Joshua Rhodes

Partitioning Methods for NP-Hard Routing Problems

by

Patrick Alexander Mannon,

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Dedicated to my loving and supportive partner Juliana.

Partitioning Methods for NP-Hard Routing Problems

by

Patrick Alexander Mannon, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Stephen Boyles

Routing plays an essential role in modern life. As our civilization grows more reliant upon the efficient movement of goods and people, the mathematical problems underlying routing decisions also grow in complexity. Given their structure, partitioning provides one option for solving routing problems more quickly. This dissertation focuses on developing partitioning frameworks for NP-hard routing problems. Specifically, different partitioning methods are applied to the traveling salesman problem (TSP), resource-constrained shortest path problem (RCSPP), and other related problems. The TSP framework, referred to as convex hull partitioning (CHP), builds on several existing partitioning methods by using a new idea for forming subproblems. CHP relies on the connection between an optimal tour and the convex hull boundary of the input points. The hull points occur in the same order in both the convex hull boundary and the optimal tour. Using this order, CHP forms a set of point

partitions based around consecutive pairs of hull points. This allows Hamiltonian paths through each partition to be connected at shared hull points, leading to a solution to the original TSP. A slightly modified framework uses a similar technique to solve sequential ordering problems (SOP). Adding a global resource constraint makes using a similar geometry-based partitioning method more difficult. Therefore, partitioning when solving an RCSPP uses the resource costs of paths to each node. When combined with an existing dynamic programming algorithm, resource cost partitioning allows all non-dominated paths from a single source to all other nodes to be found more quickly. Computational experiments show the partitioning frameworks allow both TSPs and RCSPPs to be solved more quickly with little to no decrease in solution quality. These results demonstrate the benefits of using partitioning to help solve TSPs and RCSPPs. Ideally, these partitioning techniques will be extended to provide similar benefits when solving other difficult routing problems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Modern life relies on efficient routing. Whether it is package delivery, flying across the country, scheduling a ride-share, or relying on stocked grocery store shelves, routing impacts everyone nearly every day. As the world becomes ever more reliant upon efficient movement of goods and people, the routing problems allowing for this movement become larger and more complicated. Solution methods must continue to evolve to address increasingly difficult problems. One common issue facing routing planners lies in solving problems quickly enough to take advantage of the solutions. Because the computational effort required to solve routing problems typically grows very quickly as the size of the network increases, solving several smaller problems can be faster than solving a single large instance. For this reason, partitioning can be an effective technique to solving routing problems more quickly. This dissertation focuses on developing partitioning frameworks for several NP-hard routing problems, including the Euclidean traveling salesman problem (TSP), the sequential ordering problem (SOP), and resource-constrained shortest path problem (RCSPP). Partitioning frameworks face three challenges. First, the problem must be split into subproblems. Then, each subproblem needs to be

solved. Finally, the subproblem solutions must be combined to form a solution to the original problem. Each of these issues will be addressed, but the vast majority of the work focuses on how to partition the original problem into subproblems.

1.2 Contributions

Partitioning serves as the biggest underlying theme of the methods presented in this dissertation. For the remainder of this work, partitioning refers to dividing the input problem into subproblems. While the idea behind the methods may differ, each attempts to take advantage of implicit divisions of an input set. These divisions lead to an easier or smaller problem. The solutions to these subproblems then combine to form a feasible solution to the original problem. The crux of this work therefore lies in developing partitioning methods that allow each problem to be solved more quickly while maintaining high quality solutions. In some cases, it is possible to solve problems more quickly and still achieve the optimal solution.

The main contribution of this dissertation lies in developing a geometric partitioning framework for the Euclidean TSP. The TSP is an interesting and relevant problem in its own right. It is also related to, or a direct subproblem of, many other routing problems, such as the vehicle routing problem. As an NP-complete problem, many heuristic methods for the TSP have been developed over the years. The framework developed in this dissertation is not the first method to use partitioning to help solve the TSP more quickly. However,

the method developed in this work, referred to as convex hull partitioning, uses new techniques to form partitions. As the name implies, the convex hull of the input points serves as the basis for groups of points. The relationship between the convex hull and the optimal tour helps define a set of point partitions. A Hamiltonian path is found through each of these groups. Two important characteristics of the partitions are central to CHP. First, the partition paths connect consecutive convex hull points. Second, the order in which partition paths occur in the heuristic tour is determined by the convex hull points. This solves some of the main issues of partitioning as using the convex hull as the basis for partitions determines both the order in which subproblem solutions occur and how to connect the subproblem solutions. Several extensions of CHP were also developed. Two of these attempt to directly improve CHP. A final extension modifies CHP to solve symmetric SOPs instead of the Euclidean TSP.

Resource-constrained problems make partitioning even more difficult. For that reason, a new method to solve the RCSPP takes an entirely different approach to partitioning. Instead of grouping points based on their location in the network, the groups are determined by the feasible and relevant resource cost ranges of each node. Forming these groups limits the search required to solve the RCSPP with dynamic programming. This methodology can potentially be expanded to solve other resource-constrained problems, such as the RCTSP or orienteering problem.

1.2.1 Convex Hull Partitioning

By far the largest contribution in this dissertation comes from the convex hull partitioning framework for the TSP. A slight modification of the framework solves the SOP as well. Both frameworks form subsets such that their subproblem solutions are combined in a known order. This first requires finding a subset of points for which the order is known in the optimal solution. Then, a partition is formed for each consecutive pair of points in that subset. This allows for simple recombination of subproblem solutions by joining them at their shared points. Fortunately, both the TSP and SOP have readily available ordered subsets. In the TSP, the points in the convex hull boundary of the TSP point set occur in the same order in both the optimal tour and the convex hull. Thus, a known-order subset is implicit to the problem. The SOP is not quite as simple. However, a known order subset can be formed by solving a smaller SOP instance through only the points included in precedence relationships. Besides providing the subproblem order, this also greatly simplifies solving the problem. Because the precedence relationships only affect the order of precedence points, a smaller SOP and several Hamiltonian path problems replace the original SOP.

Most of the work on these frameworks focused on the TSP, with the SOP being an extension of those techniques. To that end, the largest contributions include the overall convex hull partitioning framework, specific partitioning methods, and experiments demonstrating the efficacy of the new methods. While the idea of using partitioning to solve TSPs is not new, the specific idea

of using sequential partitions based around the convex hull boundary points is, to the best of our knowledge, a novel method. In some cases, the presented partitioning methods use existing ideas but apply them in a new way. Others are entirely new and somewhat specific to the CHP framework. Two newly developed extensions attempt to improve the framework. A recursive implementation aims to decrease the runtime by limiting subproblem size. A second extension augments the convex hull points in order to increase the number of partitions, again as an attempt to provide faster solutions. Finally, a slight modification of the framework led to a new solution method for the SOP.

1.2.2 Resource Cost Partitioning

Behind geometric partitioning, the other area of contribution focuses on dynamic programming algorithms for resource constrained problems. Specifically, the developed partitioning methods divide problems based on the resource consumption of paths leading to each node. For the RCSPP, a group of relevant points is formed for each feasible resource consumption value. What it means to be relevant will be discussed in a later section, but essentially this restricts the search to points that could reasonably occur next in the path being constructed. A dynamic programming method for the RCTSP attempts to use the same ideas. However, because Hamiltonian paths must contain every point, point relevance becomes much harder to define.

Similar to geometric partitioning, resource cost partitioning began with an idea and application to a specific problem, in this case the RCSPP. For that

reason, the biggest contribution from this research vein comes from the work done developing methods for the RCSPP. Extensions of that work led to a method for the RCTSP. The dynamic programming algorithm for the RCSPP is not entirely new, instead it builds on existing methods and ideas. However, both its application and the partitioning method used to limit its search are novel. The resulting algorithm finds all non-dominated paths from a single source to all other nodes in a network.

1.3 Outline

Chapter 2 contains background information on the relevant problems and covers some existing solution methods. The bulk of the discussion focuses on the Euclidean TSP, but constrained problems including the SOP, RCSPP, and RCTSP are also outlined. Topics include the TSP in general, including some interesting historical details, a formal mathematical description, and existing optimal and heuristic solution methods. Sections 2.1.5 and 2.1.6 cover the most relevant background information. Section 2.1.5 discusses the relationship between the convex hull of a point set and the optimal Hamiltonian tour through those points. Section 2.1.6 discusses some existing heuristics that use partitioning to help solve the TSP. Some background on the RCSPP and RCTSP are given in Section 2.2.

Chapter 3 describes convex hull partitioning (CHP), the framework developed for this dissertation. It begins with an outline of the general scheme

being developed. Then, Section 3.3 presents conditions guaranteeing CHP produces an optimal solution and arguing partitioning is the most important phase of the framework. Section 3.4 describes the process of grouping points, including partition initialization, two overarching partitioning schemes, and several new methods for assigning points. Next, Section 3.5 covers the process of going from a set of partitions to a final heuristic tour. This involves discussion of both optimal and heuristic Hamiltonian path methods and how to combine subproblem solutions into a tour. Finally, Section 3.6 builds on a bound originally found for Karp partitioning.

Chapter 4 covers the computational experiments of CHP. The section begins with a discussion of the overall testing procedure. Then, a description is given of the national and VLSI TSP test instances used. In an attempt to find a good CHP variant, testing was conducted to examine the effects of different LKH solver and partitioning method parameter values. Then, insights from this preliminary testing helped determine which CHP variants were tested on the largest instances. Finally, the section concludes with a discussion of the overall results and some areas for future work.

Chapter 5 describes the direct extensions of CHP including a recursive implementation, augmented CHP, and the use of secondary improvement algorithms on the output CHP tour. Experimental results of each extension are also presented.

Chapter 6 discusses partitioning methods for problems besides the TSP. First, the idea behind CHP is applied to symmetric sequential ordering problems. Then, some issues in applying similar geometric partitioning techniques to resource constrained problems are discussed. A new partitioning technique is then presented and an algorithm for the RCSPP is described. The chapter concludes with a discussion of areas for future work.

Chapter 7 contains overall concluding remarks.

Chapter 2

Background

2.1 Traveling Salesman Problems

The Traveling Salesman Problem (TSP) consists of finding the shortest path through a set of points such that each point is visited exactly once, and the path starts and finishes at the same point. Solving the TSP and its variants has served as the focus of many works; several books have been solely dedicated to the topic [Applegate et al., 2007], [Cook, 2011], [Gutin et al., 2002].

2.1.1 Historical Anecdotes

Before it was widely studied academically, the TSP was informally solved through the everyday work of people from who its name derives, traveling salesmen. For example, a handbook from 1832, *Von einem alten Commis-Voyageur*, provides general advice on route choice for salesmen traveling through Germany and presents routes through different regions [Applegate et al., 2007], [Cook, 2011]. Another example of the problem in practice lies in circuit-traveling religious and government officials. As a circuit court judge in Illinois, a young Abraham Lincoln routinely rode his entire circuit and his route comes close to achieving minimal distance traveled [Applegate et al., 2007], [Cook, 2011]. A final, more fun, example is Commercial Traveler, a board

game from 1890 challenging players to build tours to navigate a rail system [Cook, 2011].

A Eulerian walk consists of a closed walk traversing each edge exactly once. Leonhard Euler studied both the Konigsberg bridge problem and the knight's tour through a chessboard in the 18^{th} century [Cook, 2011]. A Hamiltonian tour consists of a closed walk visiting each node exactly once. Sir William Rowan Hamilton studied ways to visits all 20 corners of a dodecahedron. He was another pioneer of touring problems and found his work so fascinating he created a children's game of finding tours through a dodecahedron's points. When the game was criticized as being too easy, Hamilton responded that finding the tours were not at all easy to him. Thus, two fundamentals of touring problems, Eulerian walks and Hamiltonian tours, both have ties to board games.

If these brief anecdotes interest you, then you will surely find the TSP's entire storied past interesting as well. Unfortunately, the full mathematical history of TSP variants is too long to be included in this work but has been recorded in depth elsewhere [Applegate et al., 2007], [Cook, 2011].

2.1.2 Mathematical Description

Consider a graph $G = (N, E)$ where N and E are sets of nodes and edges, respectively. The TSP consists of finding the shortest length Hamiltonian tour of the nodes in N. This work focuses on symmetric, Euclidean TSPs; edge costs consist of the Euclidean distances between nodes and these costs

are assumed to be the same in both directions. As an NP-complete problem, no known method can solve large instances of the Euclidean TSP exactly in polynomial time [Papadimitriou, 1977].

The most straightforward solution method lies in complete enumeration of Hamiltonian tours through the graph. Unfortunately, the number of feasible solutions increases quickly; consider a feasible tour as a permutation of the nodes within a graph. Then, for $|N|$ nodes, $(|N|-1)!$ possible permutations exist [Held and Karp, 1962]. Other exact solution methods include integer programming models and dynamic programming.

A similar routing concept to the TSP is that of Hamiltonian paths. Instead of starting and ending at the same point, an origin and destination are specified. Then, a path beginning at the origin, ending at the destination, and visiting every point exactly once is a Hamiltonian path.

2.1.3 Optimal Methods

TSPs can be solved optimally using integer programming (IP). The Dantzig-Fulkerson-Johnson IP representation of the TSP can be seen in Figure 2.1. Binary decision variables indicate an edge's inclusion in the final tour. The goal of a TSP lies in finding the minimum cost tour as represented by the objective function (2.1.1). Constraint sets (2.1.2) and (2.1.3) ensure each node has exactly one incoming and exactly one outgoing edge, respectively. These two constraints force each node to have degree two but do nothing to prevent subtours. Constraint set (2.1.4) prevents subtours in subsets of 2 to $|N|$ - 1

$G = (N, E)$ = complete graph
N = set of points, $i \in N$
H = set of points on the convex hull boundary of N, $H \subseteq N$
I = set of points not on the convex hull boundary, $I = N \setminus H$
E = set of edges, $(i, j) \in E$
c_{ij} = cost of traveling on link (i, j)
x_{ij} = binary decision variable determining inclusion of edge (i, j)
X = vector of x_{ij} variables with one element for every edge $(i, j) \in E$
$|\cdot|$ = number of elements in a set, e.g. $|T|$ = number of edges in T
$\bar{\cdot}$ = total length of edges in a graph, e.g. $\overline{T}$ = length of tour T

$$\text{Minimize} \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{2.1.1}$$

subject to:

$$\sum_{i:(i,j)\in E} x_{ij} = 1 \qquad \forall j \in N \tag{2.1.2}$$

$$\sum_{j:(i,j)\in E} x_{ij} = 1 \qquad \forall i \in N \tag{2.1.3}$$

$$\sum_{(i,j)\in E, i\in S, j\in S} x_{ij} \leq |S| - 1 \qquad \forall S \subseteq N : 2 \leq |S| \leq |N| - 2 \tag{2.1.4}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E \tag{2.1.5}$$

Figure 2.1: Dantzig-Fulkerson-Johnson TSP IP Formulation

points. Consider a subset S composed of $|S|$ nodes. Then a subtour of S will contain $|S|$ edges. Constraint set (2.1.4) allows at most $|S|-1$ edges connecting points within S, preventing a subtour from occurring. Finally, constraints in (2.1.5) set the binary domain of the decision variables.

A dynamic programming algorithm provides a method for solving a TSP through n points with time complexity $\mathcal{O}(n^2 2^n)$ [Bellman, 1962]. This is the lowest time complexity for an algorithm providing optimal TSP solu-

tions. The algorithm, shown in Figure 1, solves a series of subproblems over increasingly large subsets of points. Let S be a subset of points, point 0 be the origin, point i be another point in S, and $f(S, j)$ be the length of the shortest Hamiltonian path through S starting at point 0 and ending at point j. To solve the TSP, $f(S, j)$ is defined as a function of solutions over smaller point subsets. Subset size is incremented until a solution over the entire input set is found.

Input: N: input set of points; $|N| = n$
Output: $\overline{T^*}$ = length of optimal tour
$f(0, 0)$;
for $s \in [1, n-1]$ **do**
  for $S \subseteq N$: $|S| = s$, $0 \in S$ **do**
    $f(S, 0) = \infty$;
    for $j \in S$: $j \neq 0$ **do**
      $f(S, j) = \min\{f(S - \{j\}, i) + distance(i, j) : i \in S, i \neq j\}$
    end
  end
end
$\overline{T^*} = \min_j\{f(N, j) + distance(j, 0)\}$

Algorithm 1: Bellman's TSP dynamic programming algorithm

One of the most highly regarded optimal solvers for the TSP is Concorde [Mulder and Wunsch, 2003]. The solver was written by David Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook, in ANSI C and is available online [Cook, a].

2.1.4 Heuristic Methods

Because exact methods for the TSP often have a prohibitively long runtime, heuristic methods are frequently used. Two general classes encompass the majority of heuristic methods. Construction methods build a tour directly, whereas improvement methods make changes to existing tours that lead to a lower cost.

2.1.4.1 Construction Methods

Construction methods directly build a heuristic solution from scratch. They can be simple methods able to find a solution quickly but not producing reliably good solutions. The time complexity and maximum heuristic tour length ratio for some common construction methods is summarized in Table A.1. The greedy, or nearest neighbor, method simply adds the next closest point to the end of the path. Beginning with any point as a starting point, the next closest point is added to the end of the solution until no points remain. Then, an edge connects the end of the path back to the starting point. This greedy algorithm guarantees a solution no worse than $(\frac{1}{2}\lceil\log(n)\rceil + \frac{1}{2})\overline{T^*}$ and only requires n^2 computations [Rosenkrantz et al., 1974].

Insertion methods repeatedly insert points into subtours until a full tour has been established. The initial subtour consists of a single, arbitrary point and added to using an insertion rule. Some common rules include nearest, cheapest, arbitrary, and farthest insertion [Rosenkrantz et al., 1974]. These rules work in largely the same way, only differing in how they choose the next

node to be inserted. First, a point is chosen for insertion. Then, the minimum cost edge is found, and the node is inserted between its endpoints. Let w be the next insertion point, then the minimum cost edge is the edge (u, v) such that the $cost(u, w) + cost(w, v) - cost(u, v)$ is less than that of any other edge. Nearest insertion finds the point not already in the subtour that is closest to any point in the subtour. Cheapest insertion directly searches for a point and minimum cost edge pair to make the insertion. Arbitrary, or random, insertion selects a random point to insert next. In farthest insertion, the point not in the subtour furthest from any point in the subtour is chosen. Another possibility when using insertion methods is to use the convex hull boundary as the initial subtour, rather than starting from a random point [Golden et al., 1980].

The Christofides algorithm uses other network characteristics to find a heuristic solution [Christofides, 1976]. It begins with the minimum spanning tree (MST) of the input points. Then, a minimal weight perfect matching is found over the subgraph of odd-degree nodes from the MST. Combining this matching with the MST provides a Eulerian tour. Finally, replacing edges leading to nodes with degree > 2 leads to the final tour [Christofides, 1976].

2.1.4.2 Improvement Methods

Starting from an initial tour, improvement algorithms make decreasing cost edge swaps until no more improvement can be made [Golden et al., 1980]. An exchange of λ edges is known as a λ-change and a λ-opt procedure refers to one using λ-changes to improve the tour. Once no more improvement can be

made using a λ-change, the tour is said to be λ-optimal [Golden et al., 1980]. As the number of edges exchanged per move increases, the algorithm increases in strength [Golden et al., 1980]. For example, a λ-optimal tour is also λ'-optimal, for all values of λ' less than or equal to λ [Helsgaun, 2000]. For example, a 3-optimal solution is also 2-optimal and finds a solution of cost no higher than that found by only a 2-opt algorithm. This improvement in solution quality comes with an increase in time complexity; a λ-opt procedure has complexity $\mathcal{O}(n^{\lambda})$. Additionally, because they converge to local optima, finding the best solution may require multiple runs of a λ-opt algorithms, each starting with different initial tours [Golden et al., 1980]. A tour of n points is guaranteed to be optimal if and only if it is n-optimal [Helsgaun, 2000].

2- and 3-opt are simplest and most common exchange algorithms [Lin, 1965]. In each iteration of 2-opt, two edges are deleted from the incumbent tour and replaced with to new edges, leading to a lower cost. In 3-opt, the iterations consist of swapping three edges. The Lin-Kernighan algorithm provides a more general approach to edge exchange by allowing exchanges of varying size [Lin and Kernighan, 1973]. At each iteration, exchanges of increasing size are checked until stopping criteria are satisfied, the incumbent tour is updated, and the next iteration begins. The overall algorithm terminates when no improving move is found.

The Lin-Kernighan-Helsgaun (LKH) algorithm, makes several improvements to the original Lin-Kernighan procedure [Helsgaun, 2000]. The biggest change lies in the basic move used in the two algorithms. Lin-Kernighan uses

2- and 3-changes as the basis for exchanges. LKH instead uses 5-changes in an attempt to achieve 2-, 3-, 4-, and 5-optimality. This increases the overall runtime of the algorithm, but other changes reduce the search space within each iteration, helping keep this increase small. In addition to the improvement steps, LKH combines many other existing heuristics to form an initial tour [Helsgaun, 2000].

2.1.5 Convex Hull and the TSP

Consider a set points N with $|N| = n$. The convex hull of N is the smallest convex set that contains every point in N [de Berg et al., 2008]. A set of points is convex if and only if the line segment connecting any pair of points in that set is also completely contained by the set [de Berg et al., 2008]. As an intuitive explanation, imagine the points in N as nails; stretch a rubber band around the nails, and let it go. The band will collapse around the nails such that its perimeter is minimized, the area within the rubber band is the convex hull, and the rubber band itself is the convex hull boundary. For the remainder of this work, any point on the boundary of a convex hull is referred to as a hull point, and any point not on this perimeter is called an interior point.

Let n be the number of points in N and k be the number of hull points. Using a two stage approach, the convex hull of N can be determined in $\mathcal{O}(n\ \log(k))$ [Chan, 1996]. The first stage involves finding the convex hulls of point subsets. Then, the second stage finds the overall convex hull more

quickly by considering only the points on the boundary of these subset convex hulls in its search. This approach requires another convex hull method to find both the subset hulls and overall hull. Two methods are the Graham scan and Jarvis march ([Graham, 1972], [Jarvis, 1973]). When considering 2-dimensional points, both methods begin by designating the point with the smallest vertical coordinate value as the hull starting point. Call this point h_0. To find the convex hull, both Graham scan and Jarvis's march find a sequence of edges making right turns from the current point. In a Graham scan, points are sorted based on the angle they make with h_0 relative to the x-axis. Finding the convex hull then consists of searching through this sorted list until finding a sequence of edges making only right turns before returning to h_0 [Graham, 1972]. For each hull point, a Jarvis scan requires iterating through edges to every other point [Jarvis, 1973]. The edge forming the largest right turn is added to the hull, and the process is repeated until returning to h_0.

Let H and T^* be the boundary of the convex hull of the point set N and minimum length Hamiltonian tour through N, respectively. It is important to note that the convex hull discussed here is the geometric convex hull of the input point set, not the convex hull of the set of feasible solutions. Note that $H \subseteq N$ while T^* is an ordering and therefore contains every point in N. One fundamental relationship connects the order of hull points in the convex hull and optimal tour.

Theorem 2.1.1. *The order in which hull points appear in T^* is the same as the order in which they appear in H [Eilon et al., 1971].*

Proof. See [Eilon et al., 1971]. □

Based on the definition of the convex hull and Theorem 2.1.1, $\overline{T^*} \geq \overline{H}$. If every point in N is also in H, then $\overline{T^*} = \overline{H}$ because H is the minimum distance order of all points in N. If not all points in N are hull points, then $\overline{T^*} > \overline{H}$ because T^* consists of sequences of interior points inserted between hull points.

Theorem 2.1.1 allows some special cases of the TSP, notably the N-line, convex-hull-and-line, and 2-convex-polygons cases, to be optimally solved in polynomial time [Cutler, 1980], [Rote, 1992], [Deineko et al., 1994], [García and Tejel, 1997]. The most general of these cases, the N-line TSP, assumes the input points lie on one of N parallel lines. In the 2-line TSP, points lie on two parallel lines. Obviously, all points also lie on the convex hull boundary and thus the optimal tour is the same as that boundary. For the general N-line case, dynamic programming algorithms exist with time complexity $\mathcal{O}(n^N)$ ([Cutler, 1980], [Rote, 1992]). In the convex-hull-and-line TSP the input points consists of hull points on the convex hull boundary and interior points on a single line segment This is an evolution of the 3-line TSP in which the requirement for parallel lines has been relaxed. Solving this special case can be done in time complexity $\mathcal{O}(k(n-k))$, where k is again the number of hull points [Deineko et al., 1994]. The 2-convex-polygons TSP finds an optimal tour of points lying on two nested polygons. This case is a generalization of the 4-line case and solvable in $\mathcal{O}(k^3(n-k)^3)$ [García and Tejel, 1997].

Several heuristic methods for solving the TSP rely upon the convex hull of the input point set. The most straightforward example lies in using the convex hull as an initialization for the insertion heuristics discussed in Section 2.1.4. Theorem 2.1.1 makes the convex hull a good starting point. Instead of starting with a single, arbitrary point and building a tour around it, the convex hull gives more reference points to build a tour around. For example, one algorithm combines the convex hull and cheapest insertion [Golden et al., 1980]. Another example is convex layering, an extension of the N-line and 2-convex-polygons cases. Instead of assuming some layout of the input points, convex layering assigns points into nested convex polygons. Then, tour improvement algorithms connect the polygons to form a tour [Liew, 2012].

2.1.6 Existing Partitioning Methods for the TSP

Partitioning takes the input points and splits them into groups. For the TSP, this technique can be useful for dividing the original problem into smaller subproblems; solving and then combining the solutions to these subproblems then forms a solution to the original problem. Because the complexity of TSP algorithms grows quickly as the number of points increases, using partitioning to form smaller subproblems can lead to a significant decrease in total runtime. However, the quality of these solutions depends on the exact method and problem. Using a given method may lead to an optimal solution for some instances but perform poorly for others.

In the context of the TSP, geometric partitioning consists of splitting

the input point set into subsets based on points' relative locations. This often leads to points nearby one another being included in the same subset. Any partitioning-based method consists of three main phases, partitioning, solving subproblems, and combining partial solutions. Each step impacts the effectiveness of the overall method. Partitioning can be performed in numerous ways, including using problem specific methods or traditional cluster analysis techniques. How points are partitioned affects both runtime improvement and solution quality. For example, partitioning a problem with n points into two subproblems, one with a single point and the other with $n-1$ points, will not lead to a significant time savings. However, dividing the original problem into two subproblems of equal size could lead to a major time improvement. Subproblems typically consist of solving smaller TSP or Hamiltonian path problems. Obviously, using a heuristic solution for the subproblems cannot guarantee an optimal solution to the original TSP. Some combination methods append subproblem solutions together to form the final solution; others use partial solutions as a starting point from which to build the final tour.

Some heuristic methods propose partitioning points into subregions of a specific shape. Let R be a square region containing the points $\in N$. The spacefilling heuristic constructs a tour by forming a spacefilling curve through R, assigning points to locations on the curve, and then ordering points based on those positions [Platzman and Bartholdi, 1989]. The spacefilling curve is formed by recursively dividing R into identical triangular subregions. Each subregion is assigned a unique, sequential identifier. Two approaches can be

used to determine a tour from this curve. First, R can be divided until each subregion contains at most one point; then the points are ordered based on their subregion identifiers [Platzman and Bartholdi, 1989]. Another method divides R a preset number of times, allowing for subregions to contain more than one point. For any subregion containing multiple points, a partial solution is found. Then, the points or partial solutions are ordered based on the identifier of their subregion to form a tour. A similar approach divides R into a grid of evenly sized squares; partitions then consist of points lying in each square [Campbell, 2006]. The spacefilling heuristic, using square subregions instead of triangles, constructs an initial tour before edge exchange algorithms find the final tour [Campbell, 2006]. Strip-partitioning divides R into $\sqrt{\frac{n}{3}}$ vertical strips [Platzman and Bartholdi, 1989]. Points within each strip are ordered vertically, and these sequences are traversed in alternating directions, either top-to-bottom or bottom-to-top. This type of method makes sense when the input has some underlying geometry. For example, using a grid to form partitions makes sense when points are uniformly distributed within R but may lead to difficulties for input point sets with changing density across R. For an input of roughly vertical, well-separated lines of points, forming and snaking through strips may work well. However, if points form horizontally-oriented lines, then horizontal strips will work better. For general instances, other partitioning methods seem favorable over partitions with a uniform size and shape.

Other partitioning methods do not require partitions of a specific shape

and instead specify the number of partitions or the maximum partition size. As the name implies, fast recursive partitioning (FRP) recursively divides the input point set into smaller groups until none of the partitions exceed an input maximum size [Bentley, 1992]. Then, the nearest neighbor heuristic finds a subtour for each partition. A final tour is found by connecting the closest points in successive partitions. While similar to FRP, Karp partitioning uses optimal subtours to find better heuristic solutions at a higher computational cost. Karp partitioning begins by dividing R into rectangles such that the number of points within any rectangle does not exceed an input maximum size [Karp, 1977]. Two variants of Karp partitioning use different strategies based on the rectangular subregions. The first requires each rectangle to share a point with at least one other rectangle, and optimal tours are found within each rectangle. Because of the points shared between rectangles, the optimal subtours are connected together, forming a spanning walk. A spanning walk occurs when each point has an even degree of at least two [Karp, 1977]. Clearly, any Hamiltonian tour is also a spanning walk, and any spanning walk can be transformed into a tour by removing or replacing edges. Furthermore, there exists a Hamiltonian tour with shorter distance than the spanning walk. To reduce a walk into a tour, begin by removing any loops. Then, for each point with degree greater than two, two of its incoming edges are replaced with a single, shorter edge connecting the other endpoints. For example, if point v is reached by edges (u, v) and (w, v), then those edges are replaced by the edge (u, w). The work outlines these procedures in general but offers no

implementation details.

Unlike most other partitioning heuristics, Karp partitioning offers a known bound on the difference between the heuristic solution and optimal tour. For each rectangle, the difference between its subtour and portion of the overall optimal tour is bounded. This leads to a bound on the length difference between the spanning walk of connected subtours and the optimal tour. Finally, because the walk can be reduced to a shorter distance tour, that tour satisfies the same bound. Let Y_j for $j = 1, ..., k$ be rectangles formed in Karp partitioning. Then the final tour T satisfies $\overline{T} - \overline{T^*} \leq \frac{3}{2} \sum_{j=1}^{k} \overline{Y_j}$.

Fast Wedge Insertion (FWI) combines partitioning with insertion heuristics for the TSP. Instead of splitting based on subregions, FWI assigns points to one of four groups based on their proximity to the edges of the bitmap closest rectangle [Xiang et al., 2015]. The method begins by identifying the corner points and points are iteratively assigned to groups representing the line segments connecting these corners. After this initial assignment, points within each group are reordered using an improvement heuristic.

To help form an initial tour, LKH allows for use of a variety of existing partitioning or clustering methods. There are no specific implementation details, but Karp, k-means, Delauney, Sierpinski, and Rohe partitioning can be used to form subproblems from the original problem [Helsgaun, b]. This integration of partitioning algorithms within LKH will be discussed further in Section 4.9.

2.2 Constrained Routing Problems

Including constraints in routing problems add another dimension of difficulty to the problem but may lead to a more realistic model. Several types of constraints can be added including a resource constraint over the whole tour, capacity constraints on the edges, time windows for visiting each point, or precedence constraints between points. If the unconstrained problem is NP-hard, the constrained version is too. Additionally, adding constraints may make a problem NP-hard, even if the unconstrained version can be solved in polynomial time. Thus, again heuristics are often used when an exact solution takes too long.

2.2.1 Sequential Ordering Problems

The sequential ordering problem (SOP), or precedence constrained TSP (PCTSP), is the problem of solving a traditional TSP through a set of points with some additional constraints specifying the order of certain points [Bianco et al., 1994]. For example, consider a four-point set $\{1, 2, 3, 4\}$ with an optimal tour of $[1, 2, 3, 4, 1]$. If an additional constraint specifies that points 3 must be visited before point 2, then the original optimal tour becomes infeasible.

The SOP arises in many real-world applications, especially in the context of routing. For example, consider the case of pick-up and delivery. Clearly, a pick-up must occur before its corresponding delivery, and this establishes an obvious precedence relationship.

Precedence relationships specify an order between a pair of points.

$$\sum_{i \in S-\{0\}} \sum_{N-S} x_{ij} \geq 1 \; \forall S \subseteq N : 0, q \in S, r \notin S; \forall (q \prec r) \in PC \qquad (2.2.1)$$

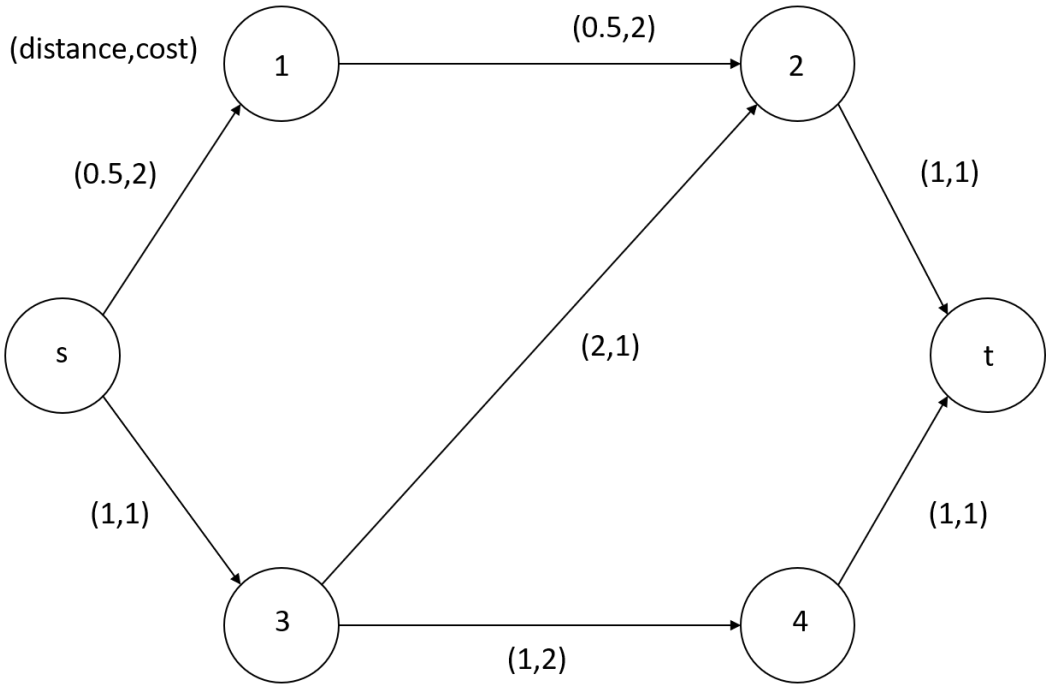
Figure 2.2: Integer Programming Precedence Constraint

Points can be included in multiple precedence relationships, and it is assumed that there are no conflicting relationships [Reinelt, 1995]. For example, again consider the four point set $\{1, 2, 3, 4\}$ with precedence relationships $3 \prec 2$ and $4 \prec 3$. This would lead to the tour $[1, 4, 3, 2, 1]$. However, including the relationship $2 \prec 4$ makes the problem infeasible because it implies $2 \prec 3$ which contradicts $3 \prec 2$. It is also assumed that the origin is not included in any precedence constraints. For the remainder of this dissertation, the term precedence points refers to any points specified in a precedence relationship.

The integer programming model for the SOP is the same as that of the basic TSP with additional constraints to maintain the specified precedence relationships. Let PC be a set of precedence constraints such that $(q \prec r) \in PC$ [Kubo and Kasugai, 1991]. Figure 2.2 shows the additional precedence constraints.

2.2.2 Resource Constrained Shortest Path Problem

A resource constrained shortest path problem (RCSPP), sometimes referred to as delay constrained, seeks to find the lowest cost sequence of edges between a given origin and destination such that additional knapsack constraints are satisfied. Figure 2.3 shows an example network and shows shortest



Knapsack constraint: Total cost $\leq \beta$

$\beta < 3 \rightarrow$ Infeasible

$\beta = 3 \rightarrow P^* = [s, 3, 2, t], \overline{P} = 4$

$\beta = 4 \rightarrow P^* = [s, 3, 4, t], \overline{P} = 3$

$\beta \geq 5 \rightarrow P^* = [s, 1, 2, t], \overline{P} = 2$

Figure 2.3: Optimal Paths from s to t at increasing resource constraints

paths satisfying incremental knapsack bound values. In the example network, edges' associated distances and costs are shown in parentheses, with distance appearing first, followed by cost. As demonstrated in Figure 2.3, if the cost threshold is too low, then the problem becomes infeasible. On the other hand, above the cost of the unconstrained minimum distance path, increasing the cost bound offers no benefit.

For a more formal definition, consider a graph $G = (N, E)$ where N and E are sets of nodes and edges, respectively. Assume an edge (i, j) in E connects nodes i and j and contains two attributes, distance and a resource cost. Additionally, assume positive edge distances and costs. Let s and t be a designated origin and destination, respectively. Then the RCSPP is the problem of finding the path from s to t with the shortest total distance and a total cost less than a specified budget β. Similar to the output of a TSP, the output path is a sequence of nodes to visit in order between s and t. The RCSPP is an NP-hard problem, leading to heuristic solution methods often being used [Zheng Wang and Crowcroft, 1996].

The IP formulation of the RCSPP is shown in Figure 2.4 [Pugliese and Guerriero, 2013]. The objective of the RCSPP (2.2.2) minimizes the cumulative distance of the path from s to t. A feasible path must satisfy the constraints defined by the expressions in (2.2.3), (2.2.4), and (2.2.5). The flow-balance constraints (2.2.3) are shared with the traditional shortest path problem and ensure a continuous path from the origin to the destination. The knapsack constraint (2.2.4) places an upper bound on the cumulative cost of the path. For this constraint to make sense, the cost of an edge must differ from its distance. Otherwise (2.2.4) would be redundant or a feasibility check. Finally, the inclusion of an edge in the final path is a binary decision, which is enforced by the constraint set (2.2.5).

E = set of edges, $(i, j) \in E$
N = set of nodes, $i \in N$
c_{ij} = cost of edge $(i, j) \in E$
d_{ij} = distance of edge $(i, j) \in E$
s = origin of path
t = destination of path
β = upper bound on the total cost of feasible paths
x_{ij} = indicator decision variable if edge (i, j) used in the path

$$\text{Minimize} \sum_{(i,j)\in E} d_{ij}x_{ij} \tag{2.2.2}$$

subject to:

$$\sum_{j\in N} x_{ij} - \sum_{j\in N} x_{ji} = \begin{cases} 1 & \text{if } i == s \\ -1 & \text{if } i == t \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in N \tag{2.2.3}$$

$$\sum_{(i,j)\in E} c_{ij}x_{ij} \leq \beta \tag{2.2.4}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E \tag{2.2.5}$$

Figure 2.4: RCSPP IP Formulation

2.2.3 Dynamic Programming Algorithms for the RCSPP

There have been several methods employed to solve the RCSPP exactly. These include Lagrangian relaxation, branch and bound, and dynamic programming [Pugliese and Guerriero, 2013]. For this dissertation, work into dynamic programming is the most relevant, and thus is the focus of the review. Appendix A.2.2 contains additional information focusing on using Lagrangian relaxation techniques to solve RCSPPs.

One prominent type of RCSPP dynamic programming algorithm is label setting, or label correcting, algorithms ([Aneja et al., 1983], [Desrochers and Soumis, 1988], [Feillet et al., 2004], [Boland et al., 2006], [Righini and Salani, 2006], [Tilk et al., 2017]). These algorithms build and store labels defining paths from the origin to another point. Labels contain information about paths that differentiate them from one another. This includes the distance, resource consumption, and final node. How this information is tracked varies based on the specific algorithm. Additionally, these basic labels may be augmented by storing additional information ([Boland et al., 2006],[Righini and Salani, 2006], [Tilk et al., 2017]). One example of this for the elementary RCSPP, is the generalized state space augmenting algorithm, which tracks which nodes have been previously added to the class [Boland et al., 2006]. Clearly, this prevents violations of the elementary constraint.

The simplest label setting example is appropriately referred to as the general label setting algorithm (GLSA) ([Desrochers and Soumis, 1988], [Boland et al., 2006]). In any label setting algorithm, labels store information describing a path. The GLSA stores a set of labels for each node, and each label contains the distance and resource cost of a path to that node. Index sets track finalized labels and those still requiring work. In the context of label setting algorithms, finalized labels are called treated labels; this means untreated labels still require work. Treatment refers to extending the path defined by a label to all successors of that path's final node. The algorithm begins with a single, trivial label including only the origin. Obviously, the distance and cost are both 0 in the initial

label. Labels are extended through the other nodes in the network. As more nodes are visited, the number of labels grows quickly. Every time a label is treated, a new label will then be created for each of the final node's successors. To prevent maintaining and extending a large number of labels, domination tests eliminate labels that cannot be included in the final path.

The domination criteria are what allow dynamic programming algorithms to be tractable for the RCSPP. Consider simple labels containing only the distance and resource consumption of a path. Let $L_1 = (d_1, c_1)$ and $L_2 = (d_2, c_2)$, where d and c are the distance and cost of the paths, respectively. Then, L_1 dominates L_2 if and only if $c_1 \leq c_2$, $d_1 \leq d_2$, and at least one inequality is strict [Boland et al., 2006]. If a path is not dominated by any other path, it is referred to as efficient or non-dominated.

By default, label setting algorithms find non-dominated paths to all destinations, thus solving the ADRCSPP [Feillet et al., 2004]. However, much of the focus has been on solving RCSPPs from a single origin to a single destination. To that end, bidirectional dynamic programming algorithms simultaneously extend paths forward from the origin and backwards from the destination ([Righini and Salani, 2006], [Tilk et al., 2017]). The most prevalent bidirectional label setting algorithm limits the resource consumption of any subpath to be half of the constraint limit [Righini and Salani, 2006]. The idea is to limit to number of labels created because as the path represented by a label gets longer, more labels had to be created and checked for domination. The premise is that two subpaths of the same length produces fewer overall

labels than one short and one long subpath. By limiting resource consumption to half of the feasible value in both the forward and reverse direction, the goal is that each subpath will reach the limit in roughly the same number of edges. To expand on this idea, a more recent method proposes to update the limit on any subpath's resource consumption as the algorithm progresses [Tilk et al., 2017]. This again helps to reduce the number of labels being generated and handled by the algorithm. For example, consider a network where the resource consumption on edges near the origin is much greater than that of edges near the destination. Subpaths from the destination would therefore search many more nodes and edges before reaching the halfway limit. Clearly, if resource costs are spread relatively evenly across a network's edges, then the rationale behind the halfway limit makes sense. However, if the resource costs are unevenly spread, then adjusting the limit as the algorithm progresses may make more sense.

Another type of dynamic programming algorithm finds the shortest distance path at incremental resource consumption until the constraint value is reached. The dynamic scaling algorithm (DSA) and DAD are two examples of this type of algorithm ([Goel et al., 2001], [Chen et al., 2008]). DSA tracks the constrained shortest path length from the origin node to all other points and allows for cost ϵ above the knapsack bound [Goel et al., 2001]. The method consists of two levels of iteration. First, a multiplier is used to scale the edge costs and determine a new knapsack bound value to use in the inner iterations. Once the edge costs have been scaled, an inner loop iterates through

increasing cost values from 0 to the scaled bound. For each of these resource costs, the shortest distance path from the origin to each point is determined by comparing the path length from the previous cost and the path lengths through feasible incoming edges. With these paths determined, the cost values are compared to the original knapsack bound; if any exceed this value, then the scaling factor of the outer iteration is increased and the process repeats. The steps are outlined in Algorithm 2.

The DAD algorithm finds paths for the inner iterations of DSA; it can also be used as an independent algorithm. Figure 3 outlines the process for finding these paths. Because the algorithm relies on two arrays to track path length, the bound values tracked must be finite, in this case being non-negative integers. If either the edge costs or bound was real-valued, an infinite number of bound values could result. Additionally, calculating paths for the next bound value relies looking back at previously calculated values based on the cost of the edge being compared. Thus, non-integer costs could point back to a bound value that does not exist in the array. If either the bound or costs are non-integer, rounding must be performed. In this case, all values are rounded to the nearest integer below [Goel et al., 2001]. Figure A.1 shows the contents of the length and predecessor tables at various iterations of the algorithm.

Other rounding rules, such as round-to-floor (RTF), round-to-ceiling (RTC), and rounding-randomly (RR), can be integrated into DSA [Chen et al., 2008]. The first two are self-explanatory, but the idea behind random rounding lies in cancelling out the rounding errors from each edge. Each rounded edge cost

Input: N: set of nodes, E: set of edges, β: upper limit on resource consumption, ϵ: error tolerance
Output: Minimum distance path with cost less than $\beta(1+\epsilon)$
$\tau = \tau_0 << \beta$;
$C(n) = \infty, \forall n \in N, n \neq s$;
while $\exists n \in N : C(n) > \beta(1+\epsilon)$ **do**
  $c^\tau = Round - to - Floor(c, \tau, \beta)$;
  $DAD(N, E, \tau)$;
  $C(n) = \sum_{(i,j) \in P(n,t)} c_{ij}^\tau$;
  $\tau = 2\tau$;
end

Algorithm 2: Delay Scaling Algorithm

Input: N: set of nodes, E: set of edges, β: upper limit on resource consumption
Output: L: map of non-dominated path lengths,P : map previous nodes
$L(s,0) = 0, P(s,0) = -1$;
$L(v,0) = \infty, P(n,0) = -1, \forall n \in N, n \neq s$;
$t = 0$;
while $t <= \beta$ **do**
  for $n \in N$ **do**
    $L(n,t) =$
     $\min\{L(n,t-1), \min_{u:(u,n)\in E \text{ and } c_{un}<=t}\{L(u,t-c_{un}) + d_{un}\}\}$;
    Track $P(n,t)$;
    $t+ = 1$;
  end
end

Algorithm 3: DAD

contributes to the total path error. Because the path error is cumulative, rounding some costs up and some down may lead to the positive and negative errors cancelling each other out. Random rounding led to faster runtimes than the original DSA, which rounds values down [Chen et al., 2008]. Interestingly, the tests also showed the last iteration of the algorithm dominated the runtime [Chen et al., 2008]. This means the initial scaling iterations do not contribute much to runtime, but they do also not lead to satisfactory solutions.

DSA uses the costs of incoming edges to choose the best path to a point [Goel et al., 2001]. An alternative algorithm uses the cumulative cost of the whole path through each incoming edge when choosing the best path to a node. This method still requires rounding, but instead of rounding the value at each edge, the whole path cost is rounded [Chen et al., 2008]. This is again an attempt to reduce the error introduced to the solution from rounding; rounding the path cost will lead to less error than rounding each edge. Aside from that change, the mechanics of the algorithm remain the same.

2.2.4 Resource Constrained TSP

The resource constrained TSP (RCTSP) is the problem of finding the shortest Hamiltonian tour through a set of points such that an additional resource constraint is satisfied. The mathematical formulation of the RCTSP is the same as that of the basic TSP with the addition of two things. First, each edge must have an additional cost attribute representing resource consumption for traversing it. The second is the constraint itself. It specifies that the total

resource consumption of all the edges in the final tour is within a specified budget. The applications, either as a standalone model or subproblems of a larger method, can be easily seen. Consider a resource constrained TSP in which travel time is minimized subject to a cost budget. This could apply to a traveler moving between cities with options of varying cost and travel time. How should the traveler spend their budget? When does it make sense to pay for a lower travel time?

Let c_{ij} be the resource cost of using the edge connecting points i and j and β be the cost budget of the edges in the final tour. As of now, it is assumed the resource costs are additive, that is the total resource consumption of a tour is simply the sum of the resource cost of its component edges. In this work, the resource consumption knapsack constraint in the RCTSP is the same as that added for the RCSPP. This constraint is defined by Equation 2.2.4 in Figure 2.4.

One generalization of this problem is to allow multiple resource constraints. This could then be modelled as an IP using multiple knapsack constraints, with one for each resource being limited. Another generalization could use a different type of cost function. For example, the total resource cost could be defined as the product of the component edge costs. However, at this point, only the case of a single additive resource cost is considered.

Chapter 3

Convex Hull Partitioning Framework

Convex hull partitioning (CHP) uses the convex hull boundary as the basis for determining partitions of the input points. Based on both its theoretical foundations and strong empirical results, CHP is an effective partitioning framework for solving TSPs, especially large instances. As outlined in Algorithm 4, CHP consists of three major parts: using the convex hull to initialize partitions, assigning points to partitions, and deriving a tour from the partitions. As a reminder, the convex hull used in CHP is the geometric convex hull of the input point set, not the hull of the set of feasible solutions.

Input: N: input set of points
Output: T: heuristic tour
H = convex hull boundary of N;
Q_0 = initialize partitions for each pair of consecutive hull points;
Q = assign interior points to partition in Q_0;
for *partition* $q \in Q$ **do**
  Solve a Hamiltonian path problem through q;
end
T = join Hamiltonian paths at hull points;

Algorithm 4: Convex Hull Partitioning

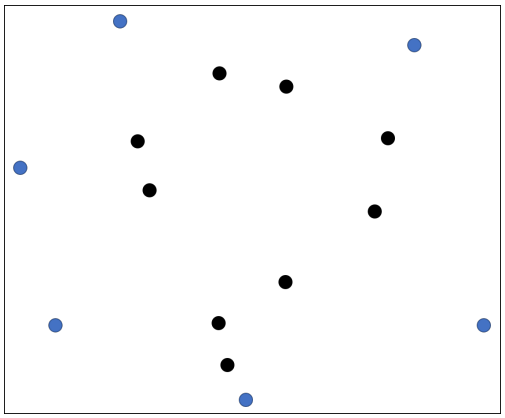
The crux of CHP lies in its use of the convex hull boundary as the basis for partitions of the input point set. The method relies on the relationship

between the convex hull boundary and optimal tour. The order in which hull points occur H is the same order in which hull points occur in the optimal Hamiltonian tour. Because of this shared order, TSPs can be thought of as a sequence of minimum cost Hamiltonian paths connecting consecutive hull points. This idea inspired CHP.

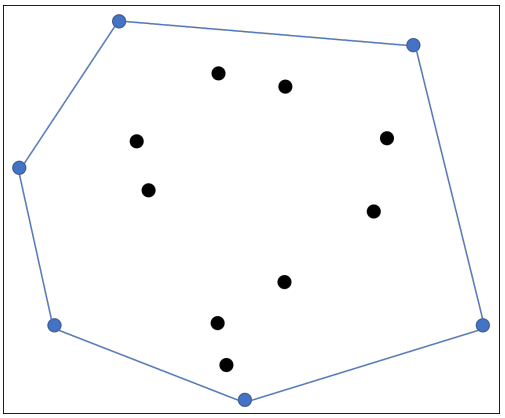
3.1 Illustrative Example

In an attempt to remove any confusion about the general procedure through which CHP solves TSPs, an illustrative example is presented in this section. The various stages of CHP are shown in Figure 3.1. Consider the TSP through the input point set shown in 3.1a. Going through the steps outlined in Algorithm 4, CHP begins with finding the convex hull boundary of the input points. Figure 3.1b shows the same input point set, this time with its convex hull boundary outlined by the solid lines connecting the problem's hull points. After finding the hull points, partitions must be initialized. This step is shown in Figure 3.1c. The empty partitions are represented by the dashed triangles connecting each pair of consecutive hull points. Next, interior points are assigned to a partition, as shown in Figure 3.1d. Now, points lying within the region encompassed by dashed lines connecting two hull points fall within the partition defined by those hull points. The last significant step is to find a Hamiltonian path through each partition. The paths must start and end at the two hull points defining the partition. To determine which is the origin and which is the destination, look at their order in the convex hull boundary.

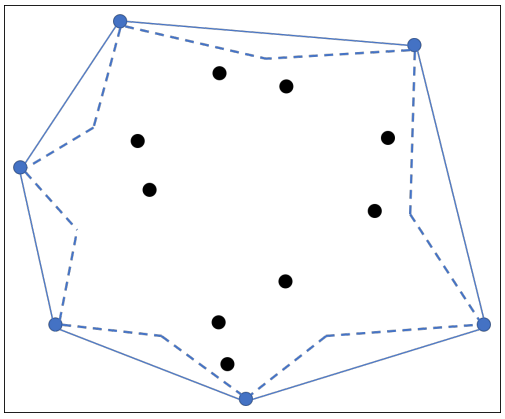
The preceding point in the convex hull serves as the origin of the Hamiltonian path. This way, the order of those hull points is preserved in the final tour. Figure 3.1e shows the Hamiltonian paths within their respective partitions. Finally, joining these paths at their shared hull points gives the final tour. For this example, the final CHP tour is shown in Figure 3.1f.



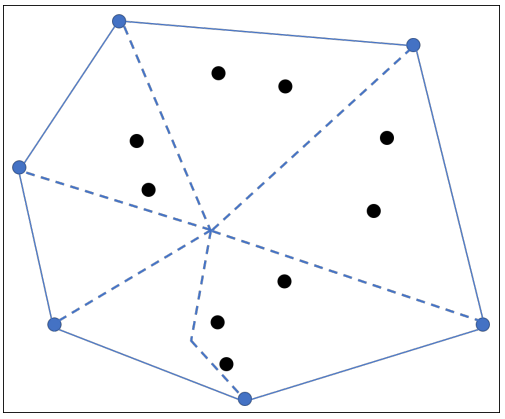
(a) Input points



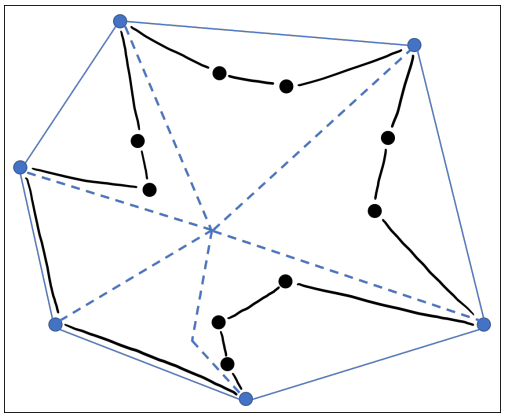
(b) Find convex hull boundary



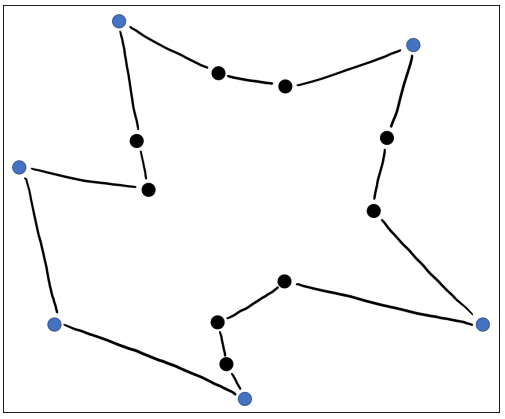
(c) Initialize partitions



(d) Assign points to partitions



(e) Find partition Hamiltonian paths



(f) Final CHP tour

Figure 3.1: Illustrative Example of CHP

3.2 Time Complexity

Before describing each phase of CHP in detail, a discussion of its time complexity is warranted. The whole purpose of this partitioning framework is to divide and conquer the original TSP, leading to a faster solution time. To that end, the time complexity of CHP depends largely on the Hamiltonian path method. Given a set of k partitions $Q = \{Q_1, Q_2, \cdots, Q_k\}$, solving subproblems and forming a tour has overall time complexity $\sum_{i=1}^{k} \mathcal{O}(HP(s, t, |Q_i|))$, where $\mathcal{O}(HP(\cdot))$ is the complexity of the Hamiltonian path method. Because the computation time of solving a TSP or Hamiltonian path grows quickly as the number of points increases, solving the subproblem for the largest partition dominates the overall complexity. As a quick example, consider solving a TSP through $n = 100$ points using Bellman's dynamic programming algorithm, which has complexity $\mathcal{O}(n^2 2^n)$. This would require roughly $100^2 2^{100} = 10^{34}$ operations. Now consider using CHP. Assume points are split into 10 equal partitions of 10 points each. Then using the same dynamic programming algorithm to optimally solve the subproblems leads to approximately $10^3 2^{10} = 1024000$ operations. As evidenced from this example, using a partitioning approach can greatly improve the expected solution time of a TSP. It is important to note, however, that as the size of the original TSP increases, the partition size will also increase and the corresponding subproblems can themselves require a long runtime.

Faster solution time is only one aspect of a successful heuristic. The other is solution quality. In the case of the TSP, that means finding a heuristic

tour with length close to optimal. In this case, solution quality depends on two factors: how the points are partitioned and the subproblem method. To examine the relative importance of each of these factors, optimality conditions for the CHP framework are discussed in the next section.

3.3 Optimality Requirements

With a few conditions, CHP is guaranteed to reproduce an optimal tour. Lemma 3.3.1 gives the conditions for finding an optimal solution and proves it is guaranteed for the case when that solution is uniquely optimal. The next step is a Theorem that generalizes this idea by removing the requirement for a unique optimal tour. Both of these results assume subproblems are solved optimally.

Lemma 3.3.1. *Assume T^* is the unique optimal Hamiltonian tour through N. Let $H \subseteq N = \{h_1, h_2, \cdots, h_k\}$ be the convex hull boundary of the input set, let $|H| = k$ points, and $Q = \{Q_1, Q_2, \cdots, Q_k : Q_i \subseteq N\}$ be a set of subsets of N. Additionally, define $HP(s, t, P)$ as the optimal Hamiltonian path from s to t through points in P.*

If H and Q satisfy:

1. *$h_i, h_{i+1} \in Q_i,\ \forall i \in \{1, 2, \cdots, k-1\}$ and $h_k, h_1 \in Q_k$*
2. *$j \in Q_i \iff j$ lies between h_i and h_{i+1} in T^**

then the optimal tour T^* is formed by joining $HP(h_i, h_{i+1}, Q_i)$ for all i. That

is, $T = HP(h_1, h_2, Q_1) + HP(h_2, h_3, Q_2) + \cdots + HP(h_{k-1}, h_k, Q_{k-1}) + HP(h_k, h_1, Q_k) = T^*$.

Proof. Let T be the tour formed in the above procedure and satisfying the two criteria given. Assume there is a unique optimal and let T^* be that optimal tour. Additionally, suppose $\overline{T^*} < \overline{T}$, implying T is suboptimal. Based on Theorem 2.1.1 and the tour construction procedure described in the Lemma, the points $\{h_1, \cdots, h_m\}$ occur in the same order in both T and T^*. Additionally, the points in Q_i lie between points h_i and h_{i+1} for all $i \in \{1, \cdots, m\}$ in both T and T^*. In order for $\overline{T^*} < \overline{T}$, there is at least one i such that $HP(h_i, h_{i+1}, Q_i)$ in T^* with strictly lower distance than $HP(h_i, h_{i+1}, Q_i)$ in T. This provides a contradiction, because $HP(h_i, h_{i+1}, Q_i)$ is defined as the optimal Hamiltonian path through Q_i in T. Therefore, $\overline{T^*} \geq \overline{T}$. Because T^* is defined as the uniquely optimal tour, $\overline{T} = \overline{T^*}$, and $T = T^*$. Thus, the tour construction procedure outlined using point partitions satisfying the two criteria will find the uniquely optimal tour. □

Lemma 3.3.1 assumes a unique optimal tour and proves this tour is reproduced if the partitions satisfy two criteria. Clearly, the procedure outlined in this Lemma mirrors CHP in both partition formation and tour construction using Hamiltonian paths. The only difference is the condition on the component points of each partition. This result is important because it proves that if given some knowledge of the optimal tour, CHP always recreates the optimal tour as its solution. Clearly, if CHP could not reproduce the optimal tour

given the information provided in Lemma 3.3.1, then it would not be a viable method. Additionally, while the given information on the optimal tour is generous, it is not perfect knowledge. In practice, the first criterion is guaranteed due to the procedure CHP uses to construct tours. The second criterion is not guaranteed without knowing the optimal tour. In spite of this, the result supports the idea that how points are partitioned determines the quality of the final solution. In fact, it is the sole factor affecting solution quality when subproblems are solved to optimality. Thus, much of the effort of developing CHP lies in devising good ways to group points with the correct hull points.

Lemma 3.3.1 assumes the TSP being solved has a unique optimal tour. To generalize the result, the uniqueness assumption is removed. There are two ways optimal tours could differ from one another. First, points may lie between different pairs of consecutive hull points in different optimal tours. Second, the subsets between consecutive hull points remains the same but at least one subproblem has a non-unique optimal solution. Corollary 3.3.2 shows optimality is reached for the case of non-unique subproblem Hamiltonian paths. The desired final result is a Theorem for the general case. That is, a result giving optimality requirements for a TSP with multiple optimal tours.

Corollary 3.3.2. *If a Hamiltonian path subproblem has a more than one optimal solution, then using any of the optimal solutions to construct a tour will lead to same final tour length.*

Proof. $\overline{T} = \sum_{Q_i \in Q} \overline{HP}(h_i, h_{i+1}, Q_i)$, where $\overline{HP}(s, t, P)$ is the optimal length

of a Hamiltonian path from s to t through points in P. Assume p_1^* and p_2^* are two optimal solutions to $HP(h_k, h_{k+1}, Q_k)$. Because they are both optimal, $\overline{p_1^*} = \overline{p_1^*} = \overline{HP}(h_k, h_{k+1}, Q_k)$. Then, either p_1^* or p_2^* will lead to the same tour length: $\overline{T} = \sum_{Q_i \in Q/Q_k} \overline{HP}(h_i, h_{i+1}, Q_i) + \overline{HP}(h_k, h_{k+1}, Q_k) = \sum_{Q_i \in Q/Q_k} \overline{HP}(h_i, h_{i+1}, Q_i) + \overline{p_1^*} = \sum_{Q_i \in Q/Q_k} \overline{HP}(h_i, h_{i+1}, Q_i) + \overline{p_2^*}$. Clearly, the same argument could be made for any number of paths with length $\overline{HP}(h_k, h_{k+1}, Q_k)$. Therefore, using any optimal solution of a subproblem leads to the same final tour length. Additionally, this result is easily extended for the case when multiple subproblems have more than one optimal solution. □

Theorem 3.3.3. *Assume $\overline{T^*}$ is the optimal Hamiltonian tour length through N. Let $\hat{T}$ be any optimal tour with length $\overline{T^*}$, $H \subseteq N = \{h_1, h_2, \cdots, h_k\}$ be the convex hull boundary of the input set, let $|H| = k$ points, and $Q = \{Q_1, Q_2, \cdots, Q_k : Q_i \subseteq N\}$ be a set of subsets of N. Additionally, define $HP(s, t, P)$ as the optimal Hamiltonian path from s to t through points in P. If H and Q satisfy:*

1. *$h_i, h_{i+1} \in Q_i,\ \forall i \in \{1, 2, \cdots, k-1\}$ and $h_k, h_1 \in Q_k$*
2. *$j \in Q_i \iff j$ lies between h_i and h_{i+1} in $\hat{T}$*

then an optimal length tour is formed by joining $HP(h_i, h_{i+1}, Q_i)$ for all i. That is, $T = HP(h_1, h_2, Q_1) + HP(h_2, h_3, Q_2) + \cdots + HP(h_{k-1}, h_k, Q_{k-1}) + HP(h_k, h_1, Q_k)$, and $\overline{T} = \overline{T^*}$.

Proof. To show that $\overline{T} = \overline{T^*}$, it is enough to show that $\overline{T} = \overline{\hat{T}}$. There are two cases for the subproblems through the partitions in Q. Either every subproblem has a unique solution or at least one has non-unique optimal Hamiltonian paths. If every subproblem solution is uniquely optimal, then $T = \hat{T}$ and clearly, $\overline{T} = \overline{\hat{T}}$. When at least one subproblem has multiple optimal solutions, Corollary 3.3.2 shows that $\overline{T} = \overline{\hat{T}}$. Therefore, in either case, $\overline{T} = \overline{T^*}$. □

There are several takeaways from these results. First, they highlight the benefits of the specific way partitions are formed around hull points. The overlap of partitions at the hull points allows the subpaths to be combined in a very straightforward way. It also determines the order in which to combine subpaths without any additional work. Finally, the hull points provide an obvious origin and destination for the Hamiltonian path within each group. If partitions were not built around hull points, extra work would be required to choose partition order and subpath end points.

The second takeaway is the importance of partitioning on the output tour. These results assume perfect partitions, where points are contained in the partition defined by their surrounding pair of hull points in an optimal length tour. Unfortunately, correctly forming groups of points meeting this criterion is more difficult than simply stating the conditions. Consider the two requirements of Theorem 3.3.3. The first is guaranteed through partition initialization in CHP, but the second is impractical to guarantee, primarily because validation requires knowing the optimal tour in advance. With an

optimal Hamiltonian path solver, any error in the final solution is a result of partitioning the points incorrectly. Even when a heuristic subproblem method is used, poorly grouping points drives much of the error. For this reason, much of the effort of this work lies in finding partitioning methods that get close enough to perfect partitions to form good final tours.

3.4 Forming Partitions

Partitioning points in CHP differs from traditional cluster analysis. While clustering is often used as an early, purely exploratory step in data analysis, it is the definitive step of CHP as partitions determine the quality of the final solution. Another difference lies in using consecutive hull points as partition bases. From the beginning, two points belonging to each partition are known and immutable. Finally, the partitions in CHP are not disjoint subsets. Instead, they share a point with two other groups. These differences require entirely new partitioning methods or modification of existing algorithms.

The basis of each partition must be two consecutive hull points but beyond that, interior points can be assigned in many ways. The simplest example lies in initializing the partitions and then randomly assigning points. Two overarching categories exist for partitioning methods: iteratively assigning points to groups and splitting heuristic tours at hull points. Iterative point assignment begins with groups containing only two hull points and assigns points based on some criteria. Tour splitting an initial heuristic solution forms partitions consisting of points lying between hull points in the initial

tour. The latter approach depends on the quality of the first TSP heuristic. If a bad heuristic tour is used to form partitions, then the final tour is also likely to be bad. For either approach, consecutive pairs of hull points serve as the basis of the partitions. Thus, initializing the partitions with two hull points is the first step in either case.

3.4.1 Partition Initialization

The algorithm begins with the convex hull, $H \subseteq N$. Let hull points be the points in H, interior points be points not in H, and $|H|$ be the number of points in H. Assume H is ordered $\{h_1, h_2, \cdots, h_{|H|}\}$ and this is also the order of hull points in the optimal tour. Then, $|H|$ clusters are initialized, each initially containing only a pair of consecutive hull points. Clearly, each hull point is contained in exactly two clusters; this serves the important role of connecting the partitions together. The shared hull points dictate how and in what order to combine the subproblem solutions into a tour. Algorithm 5 outlines the steps of the initialization step.

Input: N: set of points
Output: Q_0: set of initialized convex hull partitions, I: set of interior points
$H = ConvexHull(N)$;
for $i \in [0, |H| - 1]$ **do**
  Add cluster $\{H[i], H[i+1]\}$ to Q_0;
end
Add cluster $\{H[-1], H[0]\}$ to Q_0;
Return Q_0 and $I = N \setminus H$;

Algorithm 5: Initialize Partitions

3.4.2 Insertion Partitioning

Insertion partitioning methods use many of the same ideas as insertion construction heuristics for the TSP, as described in Section 2.1.4. After initializing clusters to contain the requisite two hull points, an insertion criterion determines which group to assign each interior point. Algorithm 6 outlines the general format for any insertion partitioning method.

Input: N: set of points
Output: Q: set of final partitions
$Q, I =$ initialize partitions;
for $p \in I$ **do**
  Assign p to cluster in Q;
end
Return Q;

Algorithm 6: Insertion Partitioning

The specific insertion scheme determines the difficulty and effectiveness of partitioning. As mentioned earlier, the insertion criteria for partitioning can be the same as those used to form heuristic solutions to a TSP. These include nearest, cheapest, furthest, and random. Because the goal lies in assigning interior points to the cluster containing their preceding and succeeding hull point from the final tour, only the nearest and cheapest insertion schemes seem to make sense for CHP. Nearest and cheapest partitioning rely on the same ideas as the construction heuristics bearing the same names. Nearest partitioning adds points to the group containing the minimum distance (and previously assigned) point. Similarly, cheapest partitioning uses the same idea as the cheapest insertion heuristic, which relies on finding the lowest cost insertion

of each new point. Cheapest insertion searches for the lowest cost insertion between pairs of consecutive points in the existing tour. Cheapest partitioning conducts a larger search by considering every pair of points already assigned to each group. Thus, the number of comparisons in cheapest partitioning is much higher than that of cheapest insertion. This leads to better solutions at the cost of a longer runtime, especially as the number of points grows. In an attempt to further balance solution quality and runtime, cheapest-m partitioning searches the m nearest points within each group. If groups do not contain m points, then cheapest-m partitioning acts exactly the same as cheapest partitioning for that group. Algorithms 7, 8, 9 contain details of the nearest, cheapest, and cheapest-m partitioning subroutines, respectively.

```
Input: i: interior point, i ∈ I; Q: set of partitions
k* = −1;
minDist = ∞;
for k ∈ [0, |Q|] do
    for p ∈ Q_k do
        if distance(i, p) < minDist then
            minDist = distance(i, p);
            k* = k;
        end
    end
end
Add i to Q_{k*}
```

Algorithm 7: Nearest Insertion Subroutine

In addition to more traditional insertion schemes, hull insertion is another approach to grouping points. Cheapest, cheapest-m, and nearest insertion rely on comparisons between partition interior points. Hull insertion, on

Input: i: interior point, $i \in I$; Q: set of partitions
$k^* = -1$;
$minCost = \infty$;
for $k \in [0, |Q|]$ **do**
  for $index \in \{-1, 0, \cdots, |Q_k| - 1\}$ **do**
    $h = Q_k[index]$;
    $j = Q_k[index + 1]$;
    $curCost = distance(h, i) + distance(i, j) - distance(h, j)$;
    if $curCost < minCost$ **then**
      $minCost = curCost$;
      $k^* = k$;
    end
  end
end
Add i to Q_{k^*}

Algorithm 8: Cheapest Insertion Subroutine

Input: i: interior point, $i \in I$; Q: set of partitions; m: number of comparison points
$k^* = -1$;
$minCost = \infty$;
for $k \in [0, |Q|]$ **do**
  $Q^m = \min\{10, |Q_k|\}$ closest points to i in Q_k;
  for $index \in \{-1, 0, \cdots, |Q_k| - 1\}$ **do**
    $h = Q_k[index]$;
    $j = Q_k[index + 1]$;
    $curCost = distance(h, i) + distance(i, j) - distance(h, j)$;
    if $curCost < minCost$ **then**
      $minCost = curCost$;
      $k^* = k$;
    end
  end
end
Add i to Q_{k^*}

Algorithm 9: Cheapest-m Insertion Subroutine

the other hand, attempts to use the relationship between the convex hull and optimal tour even more. Hull insertion attempts to find partitions such that the sum of the partition convex hull perimeters is minimized. An outline of the hull insertion scheme is shown in 10. Let the partition perimeter be the sum of the perimeter of the convex hull of each group, $\sum_{j=1}^{k} \overline{H(Y_j)}$. The partition convex hulls decompose into two portions, an edge of the overall convex hull and edges through interior points. Let $H^I(Y)$ be the latter portion of partition Y's convex hull. Then, $H^I(Y) \subset H(Y)$ and $\sum_{j=1}^{k} \overline{H(Y_j)} = \sum_{j=1}^{k} \overline{H^I(Y_j)} + \overline{H}$. The idea is that minimizing this sum leads to more compact partitions, which in turn leads to a shorter heuristic solution.

Input: i: interior point, $i \in I$; Q: set of partitions
$k^* = -1$;
$minIncrease = \infty$;
for $k \in [0, |Q|]$ **do**
  if $(\overline{H(Q_k + i)} - \overline{H(Q_k)} < minIncrease)$ **then**
    $k^* = k$;
    $minIncrease = \overline{H(Q_k + i)} - \overline{H(Q_k)}$;
  end
end
Add i to Q_{k^*}

Algorithm 10: Hull Insertion Subroutine

Finally, a hybrid partitioning method was developed. It attempts to combine the best features of cheapest-m and hull insertion to give better final partitions. The first step to assigning a new point is to determine if it lies within the convex hull of any partition. If it is within a single convex hull, assign it to that partition. If it is within the convex hulls of multiple groups,

run cheapest-m through just the encompassing partitions. If the new point lies outside every partition convex hull, use hull insertion to determine the minimum increase hull perimeter, and assign it to that group. The rationale behind hybrid insertion is that if a point being newly inserted lies within an existing partition hull, then that partition will trivially give the minimum hull perimeter increase. Given the nature of the insertion methods, it is possible that some hull perimeters may overlap. In this case, multiple partitions give the minimum perimeter increase, and cheapest-m is used to choose between them. For clarity when using hybrid partitioning with different m values, let hybrid-m partitioning be hybrid partitioning using cheapest-m as one of its sub-methods. For example, hybrid-5 combines hull and cheapest-5 partitioning.

Input: i: interior point, $i \in I$; Q: set of partitions; m: number of comparison points
if *i is outside all hulls* **then**
  Run hull partitioning;
else
  Q^1 = set of partitions with hulls encompassing i;
  Run cheapest-m partitioning on Q^1;
end

Algorithm 11: Hybrid-m Insertion Subroutine

Another dimension of insertion partitioning is the order in which points are added to groups. Clearly, this order can greatly impact the final partitions. There are many ways to sort the input points prior to insertion. In this case, sorting is based on proximity to hull points. Three ordering schemes are used

to sort points before adding them to partitions. Two are based on ascending and descending order of distance to the nearest hull point. The third sorting method finds the same descending order but shifts it such that the median distance point is added first, followed by points in descending distance order until the closest point, and finishing with the rest of the points in descending order back towards the median. This is referred to as shifted ordering. For an example of the proximity-based orders, consider the points $\{1, 3, 5, 7, 9\}$, where the point name also indicates its distance to the nearest hull point. Then, the ascending, descending, and shifted orders are $\{1, 3, 5, 7, 9\}$, $\{9, 7, 5, 3, 1\}$, and $\{5, 3, 1, 9, 7\}$, respectively. There are many other sorting methods that could be used. For example, points can be ordered randomly, or the points can be inserted in the order in which they appear in the input file. However, only ascending, descending, and shifted are included in CHP experiments at this time. The effects of these insertion orders on CHP as a whole will be discussed in a later section. To combat the variation from different ordering methods, another option lies in iterating the insertion method. Iterative insertion repeatedly runs an insertion method on the input points until the partitions stabilize.

As discussed earlier, most of the solution time is likely to be spent solving Hamiltonian path subproblems. However, forming partitions can still be a time intensive process, especially on larger problems. The contributing factors to computational cost of insertion partitioning remain consistent across the different subroutines. Finding the minimum hull point distance takes time

$\mathcal{O}(|I| * |A|)$, with $|I| = |A| = \frac{|N|}{2}$ in the worst case. Sorting with *heapsort* adds time $\mathcal{O}(|I| \log(|I|))$. The nearest insertion scheme contributes time $\mathcal{O}(n_I^2)$. Thus, nearest insertion partitioning takes time $\mathcal{O}(|I||A| + |I| \log(|I|) + n_I^2) = \mathcal{O}(n_I^2)$. When used for partitioning, cheapest insertion checks every combination of two points within each partition leading to $\mathcal{O}(\binom{n}{2})$ comparisons. Using cheapest-m limits this to at most $\binom{m}{2}$ for each partition. Thus cheapest-m requires $\mathcal{O}(k\binom{m}{2})$ comparisons for each point and has overall time complexity $\mathcal{O}(kn\binom{m}{2})$. As a reminder, $k = |H|$, $n = |N|$, and m is the number of comparison points.

3.4.3 Heuristic Tour Splitting

The second general approach to forming partitions lies in splitting a heuristic tour for the original TSP. Algorithm 12 shows the steps for this approach in more detail. Similar to insertion partitioning, tour splitting begins by initializing clusters with consecutive hull points. Then a TSP heuristic finds an initial tour, T_0. For each partition q, the indices of the two defining hull points in T_0 are found. Then, replace the initial q with the portion of the tour between the two indices. Both hull points need to be included in the final partition. Depending on the implementation, this may require adjusting the indexing selection from T_0.

Tour splitting can obtain its initial tour from any TSP heuristic. However, some methods make more sense than others. Factors to consider when choosing a heuristic method for initialization include speed and performance.

Input: N: set of points
Output: Q: set of final partitions
$Q, I =$ initialize partitions;
$T_0 =$ heuristic tour;
Rearrange T_0;
for $k \in [0, |Q|]$ **do**
    $start = where(T_0 == Q_k[0])$;
    $end = where(T_0 == Q_k[1])$;
    $Q_k = T_0[start, end]$ (Make sure to include end points) Q_k);
end
Return Q;

Algorithm 12: Heuristic Tour Splitting

Because it serves as only an initialization method, faster methods make more sense. On the other hand, a bad initial solution can have a very negative impact on the final solution. After partitioning, interior points do not shift between groups. Thus, if splitting leads to many points in the wrong partitions, the final solution will be bad. At the other end of the spectrum, if all interior points fall between the correct pair of hull points, an optimal final solution will be found, assuming an exact subproblem method is also used. One very important implementation requirement for heuristic splitting is ensuring hull points occur in the correct order in the heuristic solution. Because most heuristics do not force an order of hull points in their solution, it is possible that they may be out of order. One potential method of handling this is flipping portions of the tour until the hull points occur in the correct order, but such a method is not addressed in this work.

3.5 Hamiltonian Path Tours

The second phase of CHP consists of finding and connecting Hamiltonian paths through each partition. After finding these paths, reconstructing a solution to the original TSP requires simply connecting paths at the hull points. Because each hull point belongs to two groups and each group contains a consecutive pair, joining the incoming and outgoing paths of each hull point leads to a feasible tour. Algorithm 13 shows an outline of this path connection procedure. It consists of solving a minimum Hamiltonian path subproblem through the points in each partition and then appending the solution to the final solution.

Input: Q: set of partitions, H: set of hull points
Output: T: heuristic tour
tour $T = [\ ]$;
for *Consecutive pair* $(h_1, h_2) \in H$ **do**
    P_q = points in cluster $q = Q[h_1, h_2]$;
    T.append($HP(P_q, h_1, h_2)$;
end
Return T;

Algorithm 13: Hamiltonian Path Tour

To solve the subproblems, many TSP methods can be modified to solve for Hamiltonian paths instead with only small modifications. For example, the heuristic LKH or the optimal solver Concorde can be used to find Hamiltonian paths by forcing the edge connecting the desired start and end points to be included in the output tour. Fortunately, the file format used as input for both Concorde and LKH allow for fixed edges to be easily specified. More

straightforward methods can also typically be modified. Insertion heuristics simply add points to an initial tour list. To find a tour, the initial list starts and ends with the origin. Therefore, to find a Hamiltonian path, the initial list would need to start with the origin and end with the destination.

Using an exact subproblem method leads to the best possible solution given a set of partitions. Because an optimal solution gives the best path through a group's points, no additional improvement can be made by rearranging points within a partition. In this case, any difference between the CHP tour T and T^* comes from points belonging to incorrect partitions. Three possible exact methods include integer programming, a modification of Bellman's dynamic programming algorithm, or Concorde [Bellman, 1962], [Cook, a]. Because of its acceptance as one of the premier optimal solvers, Concorde is the logical choice of exact subproblem solver.

For many TSPs, achieving a significant runtime improvement through CHP still requires using a heuristic method to solve the Hamiltonian path subproblems. Again, there seems to be an obvious choice, this time in the form of LKH. It is also widely regarded as one of the best solvers, even though it does not guarantee optimality. For many of the National TSP test set, the optimal solution is attributed to LKH [Cook, b],[Cook, c]. Beyond LKH, other TSP heuristics can, such as insertion heuristics or the Christofides algorithm, can also be used.

By modifying the optimality conditions discussed earlier, an upper bound on CHP with a heuristic subproblem method can be established. In-

stead of an exact subproblem method, now assume the heuristic method used produces tours with a known upper bound on the ratio of its length to the optimal length. That is, there is a known bound on the ratio of the heuristic Hamiltonian path length to the optimal length. Then, Corollary 3.5.1 provides a bound on the ratio of the CHP tour length to the optimal tour length.

Corollary 3.5.1. *If a heuristic Hamiltonian path method producing paths* $HP(s,t,P)$ *with a known bound* $\overline{HP}(s,t,P) \leq \alpha\overline{HP^*}(s,t,P)$ *is used to solve subproblems in CHP, then the final tour* T *found has the same bound. That is,* $\overline{T} \leq \alpha\overline{T^*}$.

Proof. Assume Q is a set of partitions satisfying the conditions in Theorem 3.3.3: each contains a pair of consecutive convex hull points and the interior points lying between them in an optimal tour. Then, the given information and the procedure for constructing a tour from partition subtours leads to the tour bound. This begins by finding the length of T by adding the lengths of the heuristic Hamiltonian paths, $\overline{T} = \sum_{Q_i \in Q} \overline{HP}(h_i, h_{i+1}, Q_i)$. Using the heuristic bound, the heuristic lengths can be replaced with an inequality with the optimal path lengths, giving $\overline{T} \leq \sum_{Q_i \in Q} \alpha\overline{HP^*}(h_i, h_{i+1}, Q_i)$. Finally, rearranging gives the final bound of $\overline{T} \leq \alpha\overline{T^*}$. □

3.6 Worst-Case CHP Tour Length

The quality of the final solution depends on both partitioning and subproblem methods. Clearly, using a method giving optimal paths through par-

titions leads to the best possible solution. However, using a faster subproblem heuristic may lead to a further improvement in runtime with only a small decrease in tour quality.

For exact subproblem methods, deriving a bound on the worst-case performance closely follows the process used for the Karp partitioning bound [Karp, 1977]. First, Theorem 3.6.1 defines a bound on the length difference between subproblem solutions and the optimal tour portions intersecting each partition. Then, Theorem 3.6.2 bounds the difference between the heuristic solution length and that of the optimal tour.

The proofs for Theorems 3.6.1 and 3.6.2 closely follow that written by Karp [Karp, 1977]. One major difference in the results here is the use of convex hull perimeters versus Karp's use of rectangular perimeters. Additionally, partition subproblem solutions, $T(Y)$, differ. Karp partitioning uses subtours through grouped points and CHP uses a Hamiltonian path connecting hull points. One final important difference in Theorem 3.6.2 is the known relationship between $\overline{W}$ and $\overline{T}$. Karp partitioning shows a shorter distance tour exists but does not define a specific difference. However, in CHP, $W = T + H$. This allows an additional term, $\overline{H}$, to be subtracted from the left-hand side, improving the bound.

Let Y be a partition, $H(Y)$ be the convex hull of Y, and $T^* \cap Y$ be the intersection of the optimal tour and Y. Furthermore, let h_s and h_t be the two consecutive hull points in Y with $HP(Y)$ as the Hamiltonian path through Y connecting h_s and h_t. Finally, let $T(Y)$ be the tour consisting

of the Hamiltonian path between h_s and h_t along with the convex hull edge connecting h_t to h_s, $T(Y) = [HP(Y), h_t h_s]$. Then, the difference between the length of the partition tour, $\overline{T(Y)}$, and the length of the intersection of the optimal tour and the partition, $\overline{T^* \cap Y}$, is at most $\frac{3}{2}$ the perimeter of the partition. That is,

Theorem 3.6.1. $\overline{T(Y)} - \overline{T^* \cap Y} \leq \frac{3}{2} \cdot \overline{H(Y)}$.

Proof. Consider Y where $T^* \cap Y$ consists of m curves, $\{c_1, c_2, \cdots, c_m\}$. Define the $2m$ endpoints of these curves on $H(Y)$ as $\{y_1, y_2, y_3, \cdots, y_{2m-1}, y_{2m}\}$, in clockwise order around $H(Y)$. Clearly, $H(Y) = \{y_1y_2, y_2y_3, y_3y_4, \cdots, y_{2m-1}y_{2m}, y_{2m}y_1\}$. Without loss of generality, assume $\overline{y_1y_2} + \overline{y_3y_4} + \cdots + \overline{y_{2m-1}y_{2m}} \leq \overline{y_2y_3} + \overline{y_4y_5} + \cdots + \overline{y_{2m}y_1}$. Let $W(Y)$ be the spanning walk through Y consisting of $H(Y)$, $T^* \cap Y$, and $H(Y) \setminus T^* \cap Y$. Then, $W(Y)$ contains two instances of edges $\{y_2y_3, y_4y_5, \cdots, y_{2m}y_1\}$, a single instance of edges $\{y_1y_2, y_3y_4, \cdots, y_{2m-1}y_{2m}\}$, and the curves $\{c_1, c_2, \cdots, c_m\}$. Adding length of these portions gives the length of the walk overall, $\overline{W(Y)} = 2 \cdot (\overline{y_2y_3} + \overline{y_4y_5} + \cdots \overline{y_{2m}y_1}) + (\overline{y_1y_2} + \overline{y_3y_4} + \cdots \overline{y_{2m-1}y_{2m}}) + \sum_{j=1}^{m} \overline{c_i}$. Replacing the known lengths, the length of the walk is $\overline{W(Y)} = \overline{T^* \cap Y} + \overline{H(Y)} + (\overline{y_2y_3} + \overline{y_4y_5} + \cdots \overline{y_{2m}y_1})$. Based on its definition, $(\overline{y_2y_3} + \overline{y_4y_5} + \cdots \overline{y_{2m}y_1}) \leq \frac{1}{2}\overline{H(Y)}$. Additionally, the tour through the partition is shorter than the described walk. Thus, $\overline{T(Y)} - \overline{T^* \cap Y} \leq \frac{3}{2} \cdot \overline{H(Y)}$. □

Using Theorem 3.6.1 as a bound for each partition, a bound on the difference between the final heuristic and optimal tours can be derived. Because of the way each partition tour was defined, joining these tours at the

hull points leads to a walk consisting of the overall convex hull boundary and a tour through the interior points. Thus, removing the edges forming the hull boundary results in the heuristic solution to the TSP. This procedure for forming a tour helps improve the bound as well. The difference in the length of the heuristic and optimal tours is at most $\frac{3}{2}$ times the sum of the partition convex hull perimeters minus the overall convex hull boundary length,

Theorem 3.6.2. $\overline{T} - \overline{T^*} \leq \frac{3}{2} \sum_{j=1}^{k} \overline{H(Y_j)} - \overline{H}$,

where T is the heuristic tour, T^* is the optimal tour, $H(Y_j)$ is the convex hull of partition Y_j, and H is the overall convex hull of the original problem.

Proof. Let walk W consist of union of the partition Hamiltonian tours. The Hamiltonian tour $T(Y)$ through partition Y of points has two parts. The first is the edge from overall convex hull connecting the two hull points in Y. The second part is the Hamiltonian path starting and ending at the same two hull points. Clearly, this forms a tour as the Hamiltonian path visits every point in the partition, and the convex hull edge connects the start and end points of the path. This means $W = \bigcup_{j=1}^{k} T(Y_j)$ and $\overline{W} = \sum_{j=1}^{k} \overline{T(Y_j)}$. Using the results from Theorem 3.6.1, $\overline{W} \leq \sum_{j=1}^{k} \overline{T^*} + \frac{3}{2} \sum_{j=1}^{k} \overline{H(Y_j)}$. Finally, the CHP tour T can be found by removing edges of the overall convex hull from the walk W. The length of the CHP tour can then be defined in terms of W and H: $\overline{T} = \overline{W} - \overline{H}$. Therefore, $\overline{T} + \overline{H} \leq \overline{T^*} + \frac{3}{2} \sum_{j=1}^{k} \overline{H(Y_j)}$, and rearranging, $\overline{T} - \overline{T^*} \leq \frac{3}{2} \sum_{j=1}^{k} \overline{H(Y_j)} - \overline{H}$. □

Using tour splitting to form partitions also leads to performance bounds. Let T and T^* be defined as before, with T being a tour found through CHP and T^* being the optimal tour. Additionally, let T_0 be another heuristic tour used to form partitions through tour splitting. Then the length of T is no more than that of T_0.

Lemma 3.6.3. *When partitions are formed using tour splitting of initial tour T_0, the tour resulting from CHP, T, has length no longer than that of T_0. That is, $\overline{T} \leq \overline{T_0}$.*

Proof. The partitions used to form T come from T_0. The only changes from T_0 to T is rearranging interior points, not changing which hull points they come between. Assuming CHP uses an exact Hamiltonian path method for subproblems, any rearranging of interior points either improves or maintains the solution. □

Additionally, if a bound on $\frac{\overline{T_0}}{\overline{T^*}}$ exists, then it can be improved for $\frac{\overline{T}}{\overline{T^*}}$.

Theorem 3.6.4. *Assume the length of the heuristic tour T_0 is at most α times the length of the optimal tour. Then, when partitions are formed by splitting T_0 at the hull points, the ratio of the length of the CHP and optimal tours is at most α times the ratio of T and T_0, $\frac{\overline{T}}{\overline{T^*}} \leq \alpha \frac{\overline{T}}{\overline{T_0}}$.*

Proof. Because $\overline{T_0} \leq \alpha \overline{T^*}$, $\frac{\overline{T}}{\overline{T^*}} \leq \alpha \frac{\overline{T}}{\overline{T_0}}$. The bound on the ratio between the initial tour and optimal lengths is α. From Lemma 3.6.3, the length of the

CHP tour is less than that of the original heuristic tour, $\overline{T} \leq \overline{T_0}$. Therefore, if T is shorter than T_0, the bound on the ratio $\frac{\overline{T}}{\overline{T^*}}$ is strictly less than α. $\square$

A good example of the use of Theorem 3.6.4 is splitting the Christofides tour, T_c, to form partitions. Then, $\frac{\overline{T_c}}{\overline{T^*}} \leq \frac{3}{2}$. After running CHP, this bound can be improved to $\frac{\overline{T}}{\overline{T^*}} \leq \frac{3}{2}\frac{\overline{T}}{\overline{T_c}}$. Thus, Theorem 3.6.4 improves a bound on the optimal tour using information found in the running of CHP.

The results in this section assume CHP with an optimal subproblem method. In practice, a heuristic subproblem method may be used and these bounds are no longer be guaranteed. However, empirical results show many CHP variants produce tours well within these bounds. The experimental results also show partitioning plays a bigger role than subproblem method in determining the length of the final tour, assuming a good TSP heuristic is used.

Chapter 4

CHP Experiments

The purpose of this work is developing a partitioning framework capable of producing good solutions for TSPs in a reasonable time. The first step in that development process is the CHP framework explained in the previous chapter. However, that was only the one piece. CHP consists of two main stages, partitioning and solving subproblems. Several partitioning methods and insertion orders were presented in Section 3.4. Each combination of partitioning and order potentially leads to different point groups. CHP also relies on a subproblem solver to create the subpaths of the final tour before combining them. Again, several potential subproblem solvers were discussed. In summary, CHP can consist of many combinations of partitioning method, point order, and subproblem solver. Given that fact, a series of experiments attempts to find the best CHP variant.

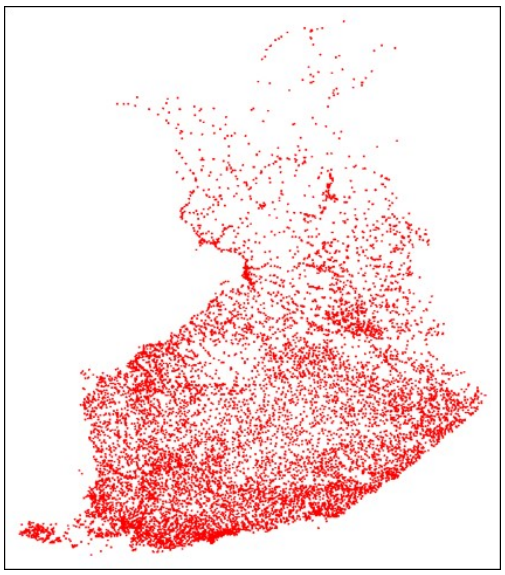
The first phase of testing consisted of running many CHP variants on a large number of small problems to immediately identify poorly performing combinations to disregard in future testing. Through this testing, LKH was identified as the best subproblem solver. However, limitations in its default settings became apparent. This led to testing of LKH using different param-

eter sets. Specifically, varying the LKH time limit shows how CHP and LKH progress over time. Next, different point insertion orders were tested to limit those included in later tests. In general, partitioning methods were developed to incorporate as few input parameters as possible. The exceptions are cheapest$-m$ and hybrid$-m$, which require the specification of an m value. To that end, testing attempts to identify m values balancing runtime and solution quality. The final experiments use promising CHP variants to solve all national and VLSI instances.

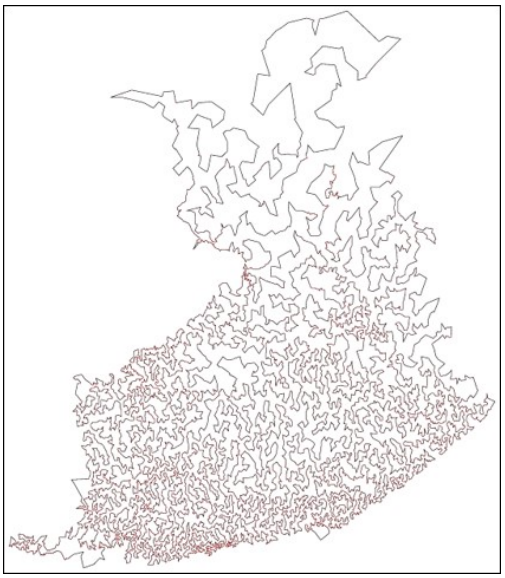
4.1 Test Problems

In total, 118 TSP instances were tested from the national and VLSI test sets [Cook, b], [Cook, d]. The problem sizes were between 29 and 35000 points. These two test sets were chosen for their accessibility and well documented optimal, or best-known, tour lengths. This provides easy benchmarks for comparison of several CHP variants.

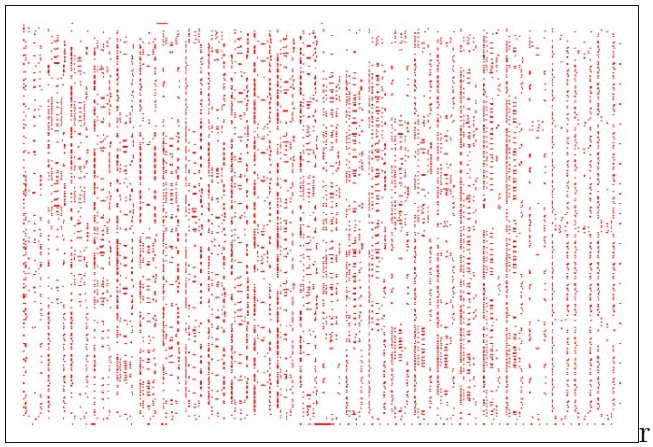
The included national TSP test problems consist of 26 problems ranging from 29 to 33708 points and are derived from cities or towns in a variety of countries [Cook, b]. In general, national TSP instances do not follow any regular shape as their points mirror the shape of their corresponding country. For example, the points and optimal tour for the TSP instance for Finland, fi10639, are shown in Figures 4.1a and 4.1b, respectively. This irregularity allows for comparison with the more regular instances of the VLSI test set, which all share a rectangular outline.



(a) Finland $fi10639$ Point Set



(b) Finland $fi10639$ Optimal Tour



(c) $xmc10150$ VLSI Point Set

Figure 4.1: Example National and VLSI TSPs

The included VLSI TSP problems consist of 92 problems with between 131 and 35000 points [Cook, d]. VLSI stands for very-large-scale integration and refers the process of creating an integrated circuit. In general, VLSI instances conform to a more regular shape than national problems. This is due to their origin in circuit planning. For that reason, they all have a rectangular outline. For comparison, Figure 4.1c shows VLSI instance xmc10150, which consists of 10150 points. This is the closest size to the Finland national instance, which has 10639 points. The difference in their layout is obvious. Because of the rectangular shape of their convex hull boundaries, different variants of CHP may perform better or worse for VLSI instances versus national instances.

4.2 Test Metrics

The metrics of interest for CHP are the length of the heuristic tour, the total runtime of the algorithm, and how these values compare to some benchmark. The optimal tour length is known for the majority of test problems. For those without a known optimal tour length, the best-known value is used. Thus, the optimal value serves as one benchmark of solution quality. Specifically, the ratio of heuristic tour length to the optimal value shows relative performance of a heuristic method. Let $\overline{T}$ and $\overline{T^*}$ be a heuristic tour length and the optimal tour length, respectively. Then the ratio of these two values, $\frac{\overline{T}}{\overline{T^*}}$, is referred to as the tour length ratio or ratio to optimal. This ratio serves as one of the main metrics used to judge the performance of different CHP

variants.

Because the TSP is an NP-hard problem, finding optimal solutions can be very time consuming. Additionally, CHP does not guarantee optimal solutions in general. It therefore makes sense to compare CHP to other heuristics, and the LKH heuristic serves as the primary heuristic benchmark, both in terms of solution accuracy and runtime. More details on LKH's performance is discussed in the next section. The following tests attempt to show the capability of the CHP framework overall and determine specific variants that perform well relative to these benchmarks.

4.3 LKH Parameter Testing

LKH itself uses many parameters affecting both runtime and solution quality [Helsgaun, b]. Much of the initial testing done on CHP used the default settings of LKH. However, changing some LKH settings causes the algorithm to terminate more quickly. Specifically, the settings controlling the number of runs and time limit on each run can force a faster runtime. By default, LKH performs 10 runs and returns the lowest distance tour. Any difference in these ten tours occurs due to randomness in construction of the initial tour or in selection of edges to swap throughout the improvement phase. LKH prioritizes solution accuracy over runtime in its default settings. The time limit of each run is set to the maximum double value; this essentially places no limit on the runtime of each iteration of LKH. Based on the parameters presented in the LKH user guide, the number of runs and time limit are the best ways to

control the runtime of the algorithm as a whole. Because CHP is a heuristic, both its accuracy and runtime are important and need to be tested. LKH finds tours with length very near optimal length, making it a good heuristic benchmark for solution quality. By setting its parameters to terminate more quickly, it also provides a speed benchmark.

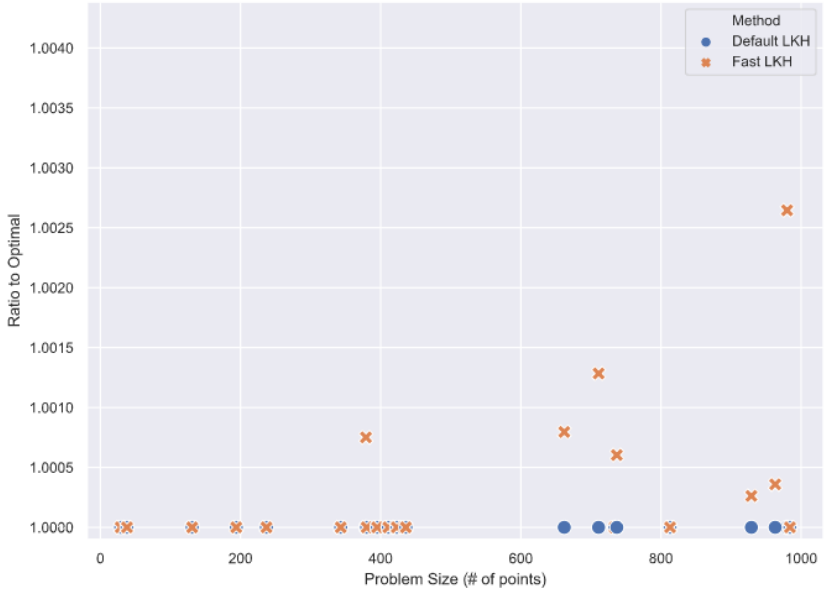
One of the biggest factors determining the runtime of LKH is the time limit parameter. Consequently, the time limit also affects the runtime of CHP, as each subproblem is solved with a separate instance of LKH. Because the current implementation solves subproblems sequentially, the time limit parameter affects CHP differently than LKH. The effective time limit in CHP is the LKH time limit parameter multiplied by the number of partitions (number of hull points). However, the LKH time limit parameter has no effect on this portion of CHP runtime spent partitioning points. To gain a better understanding of how the time limit affects the solutions from both LKH and CHP, two parameter sets were tested on TSP instances with less than 1000 points. First, LKH was run using its default parameters to set a baseline. The same problems were then solved with LKH using only a single run and a time limit of 1 second. Because this parameter set was intended to produce a solution as fast as possible, it is referred to as fast LKH in the results.

The results of the default and fast LKH parameter tests are shown in Figure 4.2. In these charts, the blue circles and orange x markers show results for the default and fast parameter sets, respectively. The runtime of default LKH, represented by the blue circles in Figure 4.2b, increases much

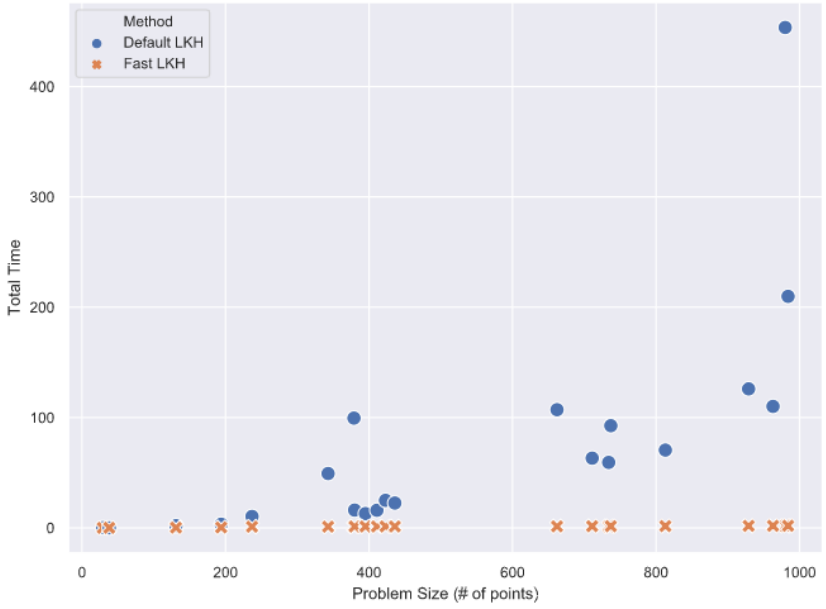
more quickly than that of fast LKH. On the other hand, the tour length ratio increases changes only very slightly between the two parameter sets. For an extreme example, consider the instance lu980, which has 980 points. Fast LKH solves the problem within 0.3% of optimal in 1.84 seconds. Default LKH solves it optimally but takes around 450 seconds. Not every instance is this extreme, but the general trend is the same. Default LKH produces a slightly shorter distance tour but takes much longer than fast LKH, especially as the size of the problem grows.

Another factor when choosing LKH parameters is their impact on CHP. Figure 4.3 shows the results of solving TSP instances with less than 1000 points using three CHP variants, cheapest-15, hull, and hybrid-15 partitioning. Each variant solves the TSPs using both default and fast LKH to solve subproblems. As above, blue and orange markers represent default and fast LKH, respectively. Additionally, different markers indicate different partitioning methods used by CHP. These tests attempt to demonstrate the impact of the LKH parameter sets relative to one another, rather than the capability of the CHP variant itself. As seen in Figure 4.3b, using default LKH to solve subproblems greatly increases the runtime of CHP. On the other hand, Figure 4.3a, which shows the tour length ratios from the tests, seems to indicate that any difference in solution quality between the two parameter sets is small, if any exists at all.

Using fast LKH leads to significant runtime improvements in both LKH and CHP subproblems. More importantly, it leads to only slightly worse tours.

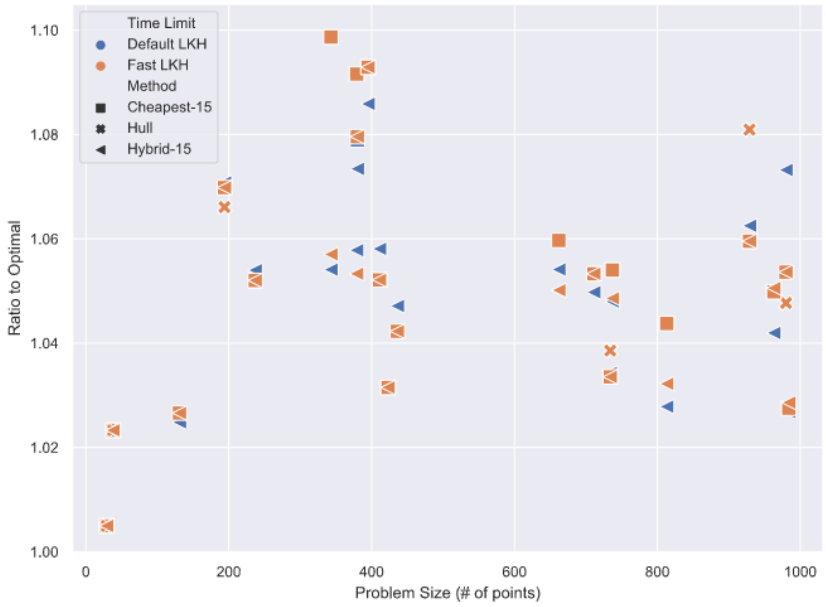


(a) LKH Tour Length

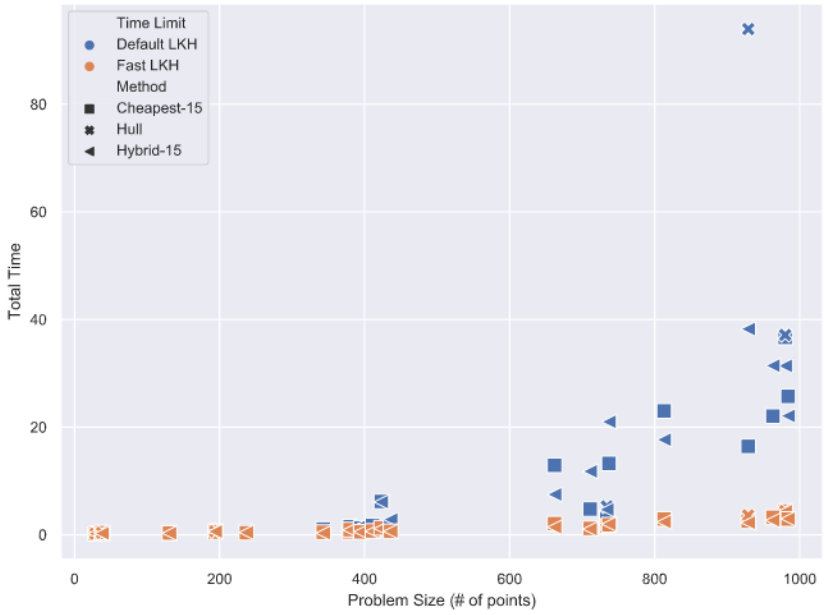


(b) LKH Runtime

Figure 4.2: Comparison of Fast and Default LKH Parameters



(a) CHP Tour Length



(b) CHP Runtime

Figure 4.3: Comparison of LKH Subproblem Parameters

For these two reasons, fast LKH is used as the heuristic benchmark and as the subproblem solver in CHP for the remainder of the tests.

4.4 Algorithm Progress Over Time

To further examine each algorithm's progress as over time, the TSP instances bm33708 and bby34656 were solved using CHP and LKH at increasing time limit values. These instances were chosen because they are two of the largest included instances at just under 35000 points each. Additionally, one each belongs to the national and VLSI test sets. The results are shown in Figure 4.4. The time limit parameter is the only obvious way to end LKH before its internal termination criteria are met. According to the LKH user guide, the time limit parameter "specifies a time limit in seconds for each run" [Helsgaun, b]. The parameter works slightly differently in CHP because a separate instance of LKH solves each subproblem. This means that the effective time limit is the input value multiplied by the number of subproblems. In Figure 4.4, runtime is shown on the x-axes, with the tour length ratio shown on the y-axes. Points in each series show the results for increasing time limit values. The charts show the results from time limit values of 1, 2, 5, 10, 30, and 60 seconds. Although increasing the time limit does typically increase the runtime, the actual runtimes were much longer than the input parameter. This is especially true for LKH. The CHP tests exhibit more noticeable changes as the parameter value increases, as runtime increases are compounded by multiple subproblems. The longer runtimes also lead to tours closer to the optimal

length. It appears the time limit parameter does not solely determine the algorithm's runtime. Instead, it seems as though some portion of the algorithm is unaffected by the time limit parameter. There appears to be a minimum runtime regardless of time limit value. This makes sense for CHP because partitioning the points contributes to the overall runtime. On the other hand, LKH is called directly using an input point file and there is no obvious external runtime contributor.

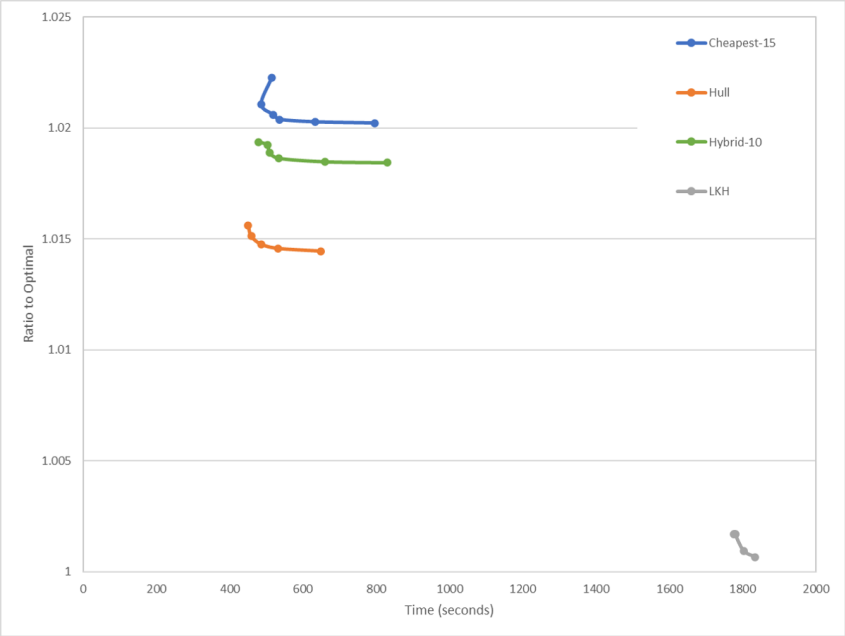
Too see these trends, consider Figure 4.4b. The orange markers show the results of solving bby34656 CHP with hull partitioning. No time limit provides a runtime less than approximately 600 seconds. For small time limits, the overall runtime fluctuates around this apparent minimum time. When the time limit increases to 30 and 60 seconds, the overall runtime increases by around 100 and 200 seconds, respectively. This larger increase demonstrates the time limit compounding due to multiple subproblems. Again, the seeming minimum time for CHP can be explained by partitioning the points, as this is not affected by the LKH time limit in any way. Now consider the yellow markers showing the LKH tests on bby34656. Again, there seems to be a minimum time; for LKH it is much higher. No time limit restricts LKH to a runtime less than around 1700 seconds. Increasing the parameter value does lead to longer solution time, but the compounding exhibited by CHP is not seen in the LKH tests. The reason for the consistent minimum time is less clear in this case. LKH constructs a tour from an input point set read directly from a TSP file. One potential explanation is the time limit only restricting the time

spent on tour improvement and not on the initial construction procedure. This would explain the consistency of the minimum time and why tours improve with a higher time limit. LKH was also run with 3600 second and the default time limits. For both tested TSP instances, the 3600 second time limit led to an overall runtime of approximately 5200 seconds. LKH using the default setting ran for 10 hours without terminating.

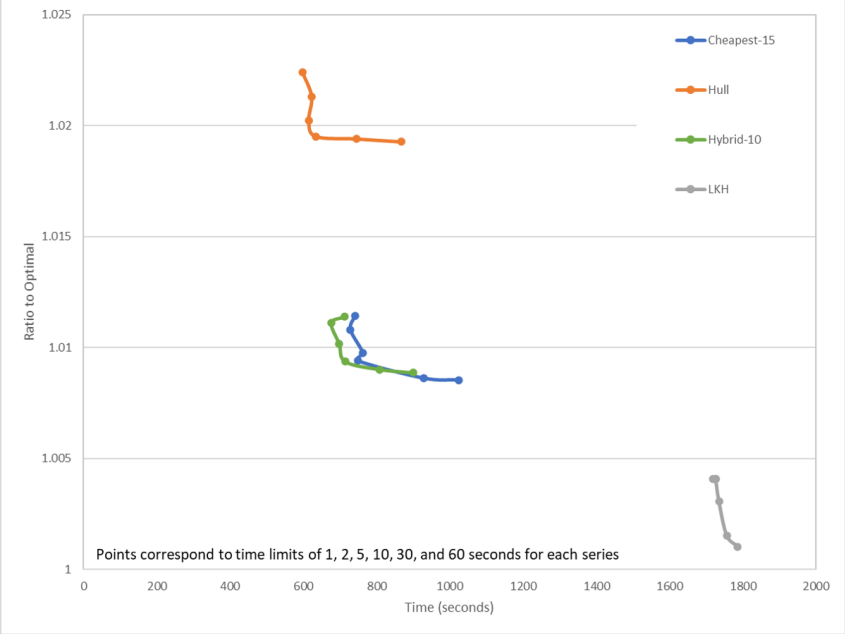
These tests further show the efficacy of CHP in solving TSPs quickly. Again, the time limit parameter is the only obvious way to terminate LKH early. The time limit does not restrict the runtime of CHP and LKH in exactly the same manner; it is less restrictive for CHP because it only limits each subproblem instead of the overall algorithm. Despite this, CHP runs significantly faster than LKH for the same time limit value. Even when using the largest tested time limit, CHP ends well before the fastest LKH test completes.

4.5 Insertion Order Testing

As mentioned in Section 3.4.2, the order in which insertion partitioning assigns points impacts both the runtime and solution quality of CHP. For this reason, three different insertion orders were tested. The insertion orders described in Section 3.4.2 include interior points in ascending order of proximity to their nearest hull point, descending order of proximity to their nearest hull point, and a shifted descending order. These orders are aptly referred to as ascending, descending, and shifted in the test results. To assess the effects of each ordering method, tested variants consist of combinations of each inser-



(a) bm33708



(b) bby34656

Figure 4.4: CHP and LKH Algorithm Progress Over Time

tion order with different partitioning methods. Due to the long runtimes seen during testing and the number of CHP variants being tested, the tests include only national and VLSI instances with less than 20000 points. The results of the insertion order testing are shown in Figures 4.5 and 4.6. More extensive summary statistics of tour quality using the different orders are shown in Table 4.2.

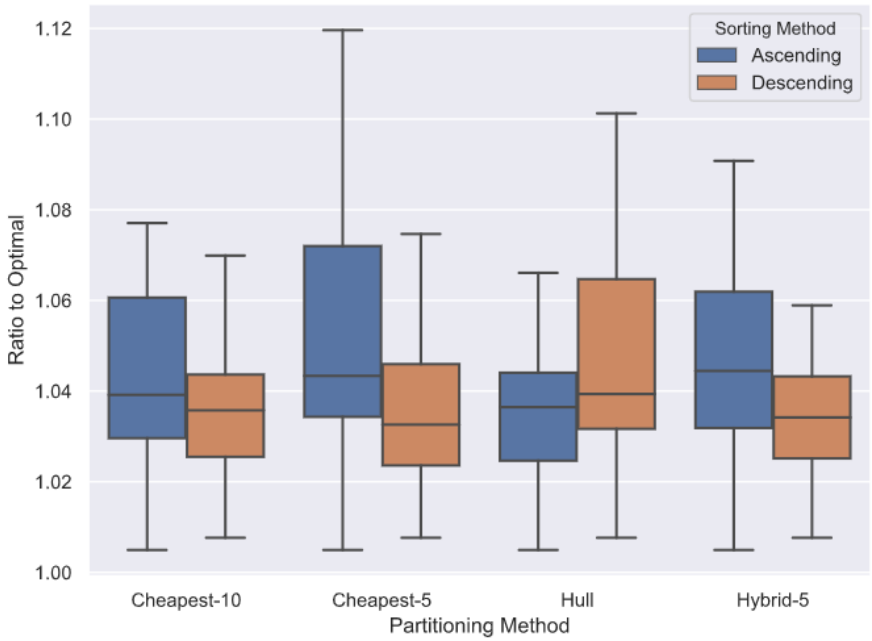
Ascending and descending ordering were tested on national TSP problems using cheapest-5, cheapest-10, hull, and hybrid-5 partitioning. A summary of these test results is shown in Figure 4.5a. From the figure and Table 4.2, it is difficult to determine which insertion order leads to the best overall solutions. The worst-case performance is worse when using a descending order, which leads to a higher average performance as well. This is especially true for hull and hybrid-5 partitioning. Aside from the worst cases, a descending order seems to lead to better solutions. The median and 75^{th} percentile values are better for all but hull partitioning. However, both ascending and descending insertion orders lead to tours with length within 4.2% in at least half of the tested instances, regardless of partitioning method.

In addition to ascending and descending orders, a shifted descending order was tested on the VLSI TSPs. This means that in total, 3 insertion orders and 4 partitioning methods were used. In the end, 11 CHP variants were tested, as shifted order insertion was not tested with cheapest-5 partitioning. Summarized solution quality results are shown in Figure 4.5b. To some degree, the results of the VLSI instance testing highlight the difference between the

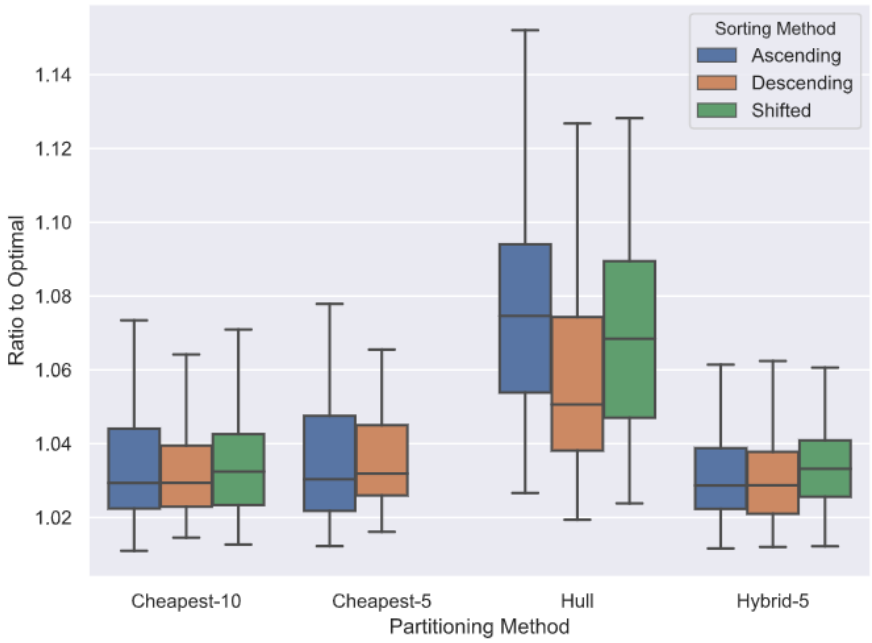
two test sets. For example, hull partitioning using ascending order was one of the best performing methods for national TSPs, but, for VLSI problems, it is the worst method across every recorded summary statistic. Hull partitioning in general does not seem effective for solving VLSI instances. On the other hand, the cheapest$-m$ and hybrid$-m$ partitioning methods seem to perform better on VLSI instances. The mean and median tour lengths are less than 4% of optimal for any m value and insertion order. With the exception of hull partitioning, the shifted insertion order seems to be the worst performing insertion order. Ascending and descending had roughly the same mean and median tour length ratios for cheapest-5, cheapest-10, and hybrid-5.

The performance test results do not show a clear distinction between ascending and descending insertion orders. The runtimes for the national and VLSI TSP insertion order tests are shown in Figure 4.6. The figure shows the runtime of CHP variants as problem size grows for both national and VLSI instances. The partitioning methods are represented with different markers and insertion orders are shown in different colors, with blue, orange, and green indicating ascending, descending, and shifted orders, respectively. For example, the orange x in the top right corner of Figure 4.6 represents hull partitioning using descending order.

Ascending and descending orders were tested on all TSP instances with less than 20000 points. In addition to ascending and descending orders, a shifted order was also tested for VLSI instances. For instances with less than 5000 nodes, no clear distinction is shown in the chart. However, as problem size



(a) National TSPs



(b) VLSI TSPs

Figure 4.5: Insertion Order Tour Length Ratio Summary

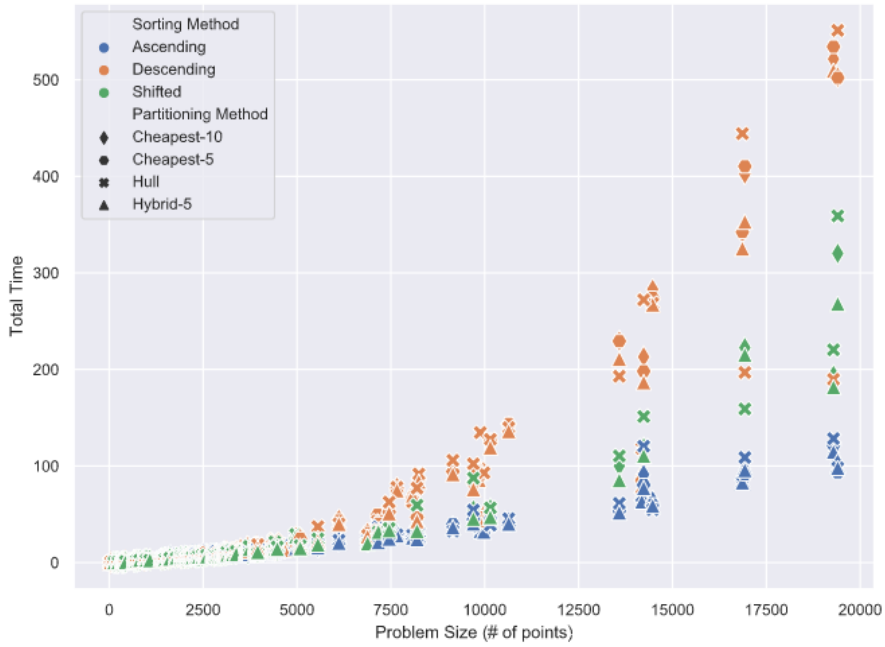


Figure 4.6: Runtime of CHP with Different Insertion Orders

grows, runtime trends become evident. Descending order variants take much longer than those using the same partitioning method with ascending order. A pattern is evident for shifted order variants as well. For a given instance and partitioning method, using a shifted insertion order typically leads to a runtime faster than descending order but slower than ascending order. For example, look at the vertical row of symbols near 17000 points. These represent different variants tested on the same instance. For each partitioning method, the runtime of the shifted order variant (green) is higher than the ascending order method (blue) and lower than that using a descending order (orange).

Based on these tests, the remaining experiments test CHP using only

an ascending insertion order. In general, using a shifted order takes more time and seems to produce worse results. A descending insertion order typically produces as good, if not better, tours. However, it led to runtimes significantly higher than using ascending order. The likely reason behind this is points being assigned to a fewer number of partitions, leading to larger subproblems. As a reminder, assigning points in a descending order means the furthest points are placed into partitions first. This immediately leads to partitions encompassing large areas of the point set but being defined by a few points. This differs from ascending order, in which partitions are built outwards and have boundaries that grow more slowly. The developed partitioning methods assign points based on their proximity to previously inserted points. When assigning points to one of a few partitions with large areas but few points, the partitioning methods have fewer reference points in each partition. This leads to points distributed to partitions differently than when using an ascending order. With a descending insertion order, partitions tend to be more variable in size, some being very large and others containing a relatively few number of points. These large groups lead to large subproblems, which could explain the longer runtimes.

4.6 Cheapest$-m$ Testing

Cheapest$-m$ and hybrid$-m$ partitioning require an input parameter, m. Because hybrid partitioning is essentially a wrapper that uses cheapest$-m$ partitioning for some points, the parameter functions similarly in both meth-

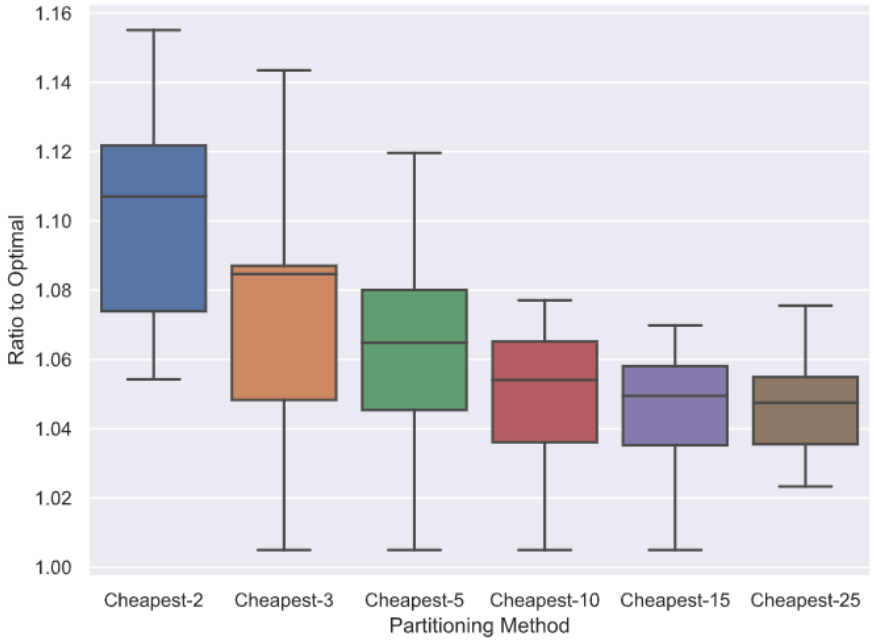
ods. The value of the m parameter sets the number of points within each partition checked to determine where to assign a new point. For example, if m is set to 5, then combinations of the 5 nearest points in each partition are checked for the minimum cost insertion. Then, the group with the global minimum insertion is chosen to add the point. To test the effects of the parameter, instances with less than from 5000 nodes from both the national and VLSI problem sets were tested with m values of 2, 3, 5, 10, 15, and 25 points. In theory, increasing the m value leads to a better final tour because the likelihood of finding a lower cost insertion. However, increasing the number of comparison points increases the runtime of the algorithm, as more combinations must be checked. To examine the effectiveness of cheapest$-m$ CHP variants, both the ratio of heuristic to optimal tour lengths and a comparison of runtimes are considered.

As mentioned earlier, problems from the two test sets exhibit different general characteristics. To recap, national TSPs do not have the same shape outline, in general. They are instead shaped like the countries from which the problems are derived. Additionally, they tend to have more varied points densities, with some very dense regions, while other regions are very sparsely populated with points. In contrast, VLSI problems have a rectangular outline and tend to be densely packed with points throughout. Because of these differences in general structure, different methods may be more or less effective on problems of different types. Knowing and understanding these differences may allow for recommendations of a specific CHP variant based on the layout

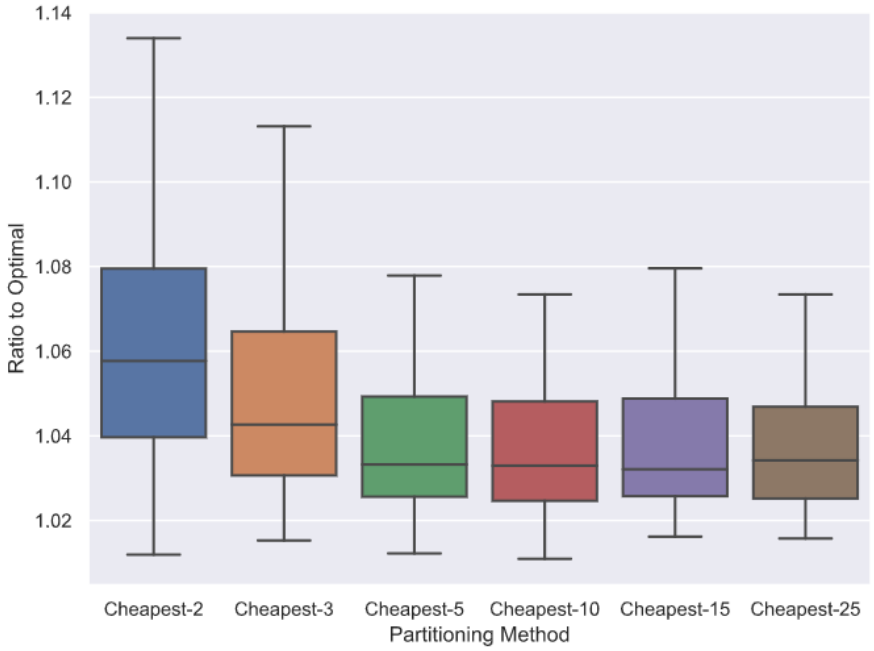
of the input point set.

Figure 4.7 and Table 4.1 show summaries of the ratio of CHP tour length to optimal tour length broken down by test set. The results show subtle differences in performance between the two problem types. The tested cheapest$-m$ variants have better best-case performance for the national TSPs. This can be seen in Figure 4.7a, as the minimum ratios are lower and more consistent across the different m values. On the other hand, the mean and median ratios were lower and more consistent for VLSI instances. As shown in Figure 4.7b, at least half of test problems were solved to within 3.5% of optimal for cheapest-5, 10, 15, and 25; the average optimality gap across all VLSI instances was also around 4% for the same m values. Cheapest-10, 15, and 25 are only slightly less consistent for the national TSPs. However, the median and mean ratios were above 4% for all m values.

Solution quality is only on factor in heuristic performance; the other is runtime. Figure 4.8 shows the solution time for the cheapest$-m$ variants as the number of points increases. Each cheapest-m variant is shown with a different color and marker, with the colors corresponding to the box plots in Figure 4.7. While the previous discussion of solution quality focused on aggregate performance, these results examine how the runtime of cheapest$-m$ grows for each instance. As seen in Figure 4.8, all cheapest$-m$ variants follow the same general trend. The linear nature of the runtime increase makes sense given the time complexity of cheapest$-m$ partitioning when $m << |N|$, $\mathcal{O}(|H||N|\binom{m}{2})$. In general, $|H| << |N|$, and the complexity becomes $\mathcal{O}(|N|\binom{m}{2})$. At the two



(a) National TSPs



(b) VLSI TSPs

Figure 4.7: Cheapest$-m$ Tour Length Ratio Summary

extremes, the time complexity of cheapest-2 and cheapest-25 are $\mathcal{O}(|N|)$ and $\mathcal{O}(300|N|)$, respectively. This represents only a portion of the total time as much of CHP's overall runtime consists of time spent solving Hamiltonian path subproblems. Figure 4.8 shows how each variant impacts the overall runtime of CHP. Therefore, the absolute difference in runtime between cheapest$-m$ variants may not clearly reflect the magnitude of their respective time complexities. Instead, the figure shows how they impact the runtime from start to finish of CHP, which makes more sense when determining the overall effectiveness of each cheapest$-m$ variant.

Cheapest-25 is noticeably the most time-consuming variant, especially for the largest test problems. Similarly, cheapest-2 and cheapest-3 are the least time consuming for most instances. Cheapest-5, 10, and 15 tend to be grouped closely together. Looking back at the summary of tour length ratios shown in Figure 4.7, this grouping makes sense. The summary shows that, for the tested problems, m values of 5, 10, and 15 usually find solutions of nearly the same length. This in turn indicates similar, if not the same, partitions formed from cheapest-5, 10, and 15. Because solving subproblems requires a significant portion of the total runtime, it makes sense that variants finding similar partitions are grouped together in Figure 4.8.

Based on these tests, 10 and 15 appear to be the best of the tested m values. Cheapest-2, 3, and 5 do not as reliably lead to high quality tours. While cheapest-25 offers similar quality solutions in general, it does so at a consistently higher runtime. In summary, cheapest-10 and cheapest-15 seem

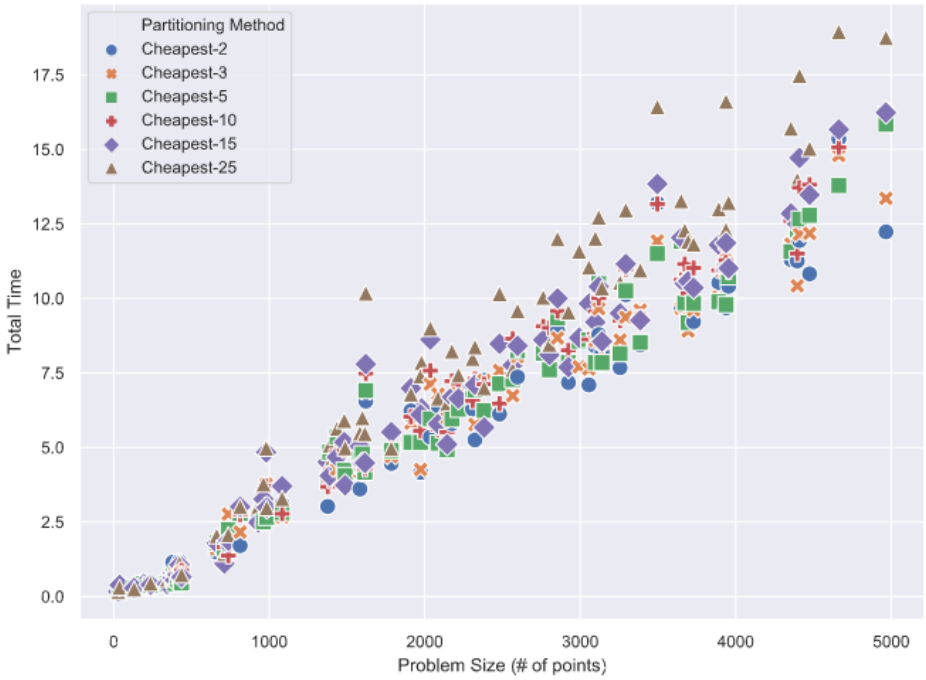


Figure 4.8: Cheapest$-m$ Runtime as Problem Size Increases

to balance runtime and solution quality. Therefore, m values of 10 and 15 are used in the majority of the remaining cheapest$-m$ and hybrid$-m$ testing.

4.7 Large Instance Testing

After performing exploratory tests of relevant parameters, a somewhat more limited set of tests were conducted on the remainder of the large national and VLSI instances. For national instances, cheapest-5, cheapest-10, cheapest-15, hull, hybrid-5, hybrid-10, and hybrid-15 partitioning were tested. Because it contains more TSP instances, testing on the VLSI problem set included only cheapest-10, cheapest-15, hybrid-10, and hybrid-15 partitioning. The results from testing insertion orders show hull partitioning does not work as well for VLSI instances, and thus it was not included for those tests. Additionally, all partitioning assigned points in ascending order. As a reminder, fast LKH solved subproblems for all the following test results.

The results presented here resemble those from previous sections. Again, both an examination of tour length ratio summary statistics and runtimes allow for an assessment of CHP performance. In addition to comparing CHP variants with one another, the comparison includes LKH results as a benchmark. National TSP results are covered first, followed by the VLSI results. Two types of charts are discussed in the following sections. As seen previously, box plots summarize the tour length ratio results for the different methods. Additionally, scatter plots show tour length ratio and runtime as problems grow in size. In both cases, different methods are shown in different colors. In

the scatter plots, different markers also help differentiate the various methods.

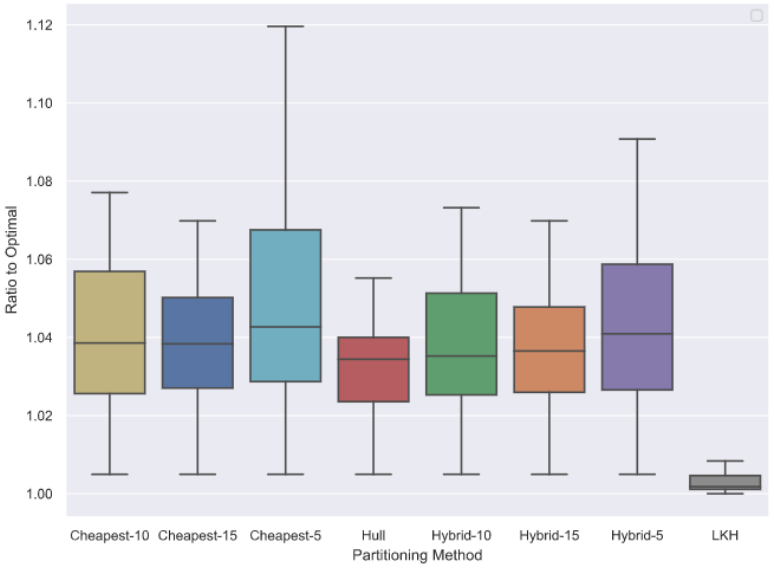
4.7.1 National TSPs

Figure 4.9 shows summary statistics for the ratio of CHP tour length to the optimal (or best-known) tour length for national TSP instances. As before, the box plots show the median, upper and lower quartiles, and overall range of the tour length ratios. Figure 4.9a shows these values when calculated across all 26 instances tested. The same statistics when only instances with at least 10000 points are shown in Figure 4.9b. These two charts allow for some insights into how CHP's performance scales as problem size grows. More detailed statistics are shown in Table 4.3. Across all instances, hull partitioning exhibits the best mean and median performance, with both values around 3.5% above the optimal tour length. When aggregated over only instances with at least 10000 points, every tested CHP variant sees an improvement in their aggregated tour length ratios. For these larger instances, hull partitioning again shows the most consistency in average and median ratios. However, three other variants seem to perform nearly as well as hull partitioning on larger problems. Cheapest-15, hybrid-10, and hybrid-15 all exhibit very similar mean and median ratios. On average, they find tours just under 3% above optimal. At least half of the larger instances were solved to within 2.5% of the optimal length for all three variants. This is only slightly higher than the median optimality gap of hull partitioning, which was just under 2.3%. Given these results, hull partitioning appears to produce tours with length closest to

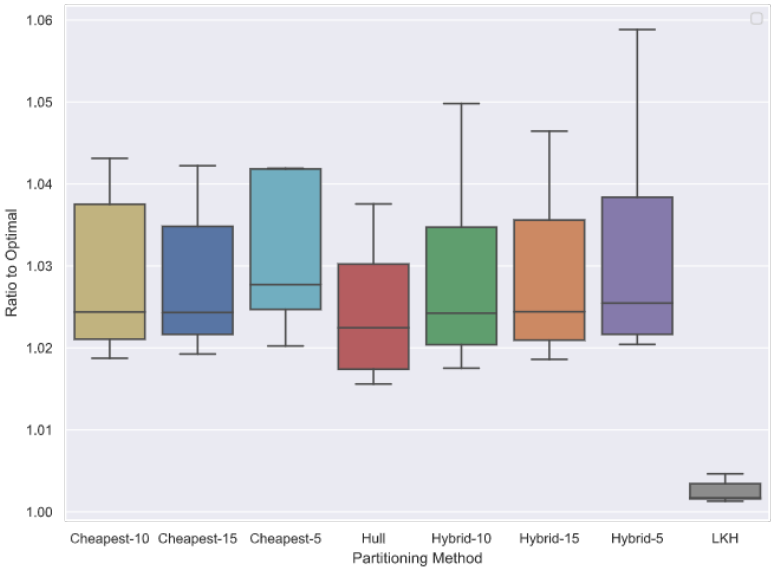
optimal for national TSPs.

As expected, LKH finds very good tours. Overall, at least half of tested instances were solved to within 0.2% of the optimal length. Additionally, LKH found the optimal tour for several instances with less than 10000 points. While LKH still finds tours well within 1% of optimal for most national TSP instances larger than 10000 points, the summary statistics show a slight decrease in performance. Based on Table 4.3, the worst-case for LKH comes from a problem with more than 10000 points, which may cause the higher average and median values.

The charts in Figure 4.10 show runtime and tour length ratios for each national TSP instance. The two plots allow trends to be identified as problems grow larger. The top chart shows tour length ratio as problem size increases; the bottom shows runtimes. By examining the results in this way, two trends become clear. First, as problem size increases CHP finds tours increasingly close to the optimal length. Second, the difference between CHP and LKH runtime increases as problems grow larger. For problems much smaller than 10000 points, the runtime improvement may not make up for a higher length tour produced by CHP. However, as problems grow large, significant time savings are experienced by CHP. A comparison of CHP runtimes to LKH runtimes is shown in Table 4.4. The table contains the ratio of CHP time to LKH time, again split into all instances versus only instances with more than 10000 points. Many of the CHP variants have similar runtime improvements. It is hard to judge the fastest CHP variant based on these results. However,



(a) All Instances



(b) Instances with ≥ 10000 points

Figure 4.9: National TSP Tour Length Ratio Summary

they all run much more quickly than LKH. Except for cheapest-15 and hybrid-15, all of the tested variants solve at least 75% of the large national instances in less than a quarter of the time as LKH. When considering runtime, CHP also performs well in the worst-case, as no CHP variant takes more than 30% of the LKH time to solve any instance.

Based on Figures 4.9 and Tables 4.3 and 4.4, hull partitioning appears to be the best of the tested CHP variants for solving national TSP instances. It solved most large problems more than 4 times faster than LKH and typically found tours with length about 2.4% above optimal, on average. In general, the tested CHP variants run in around the same time as LKH for smaller instances. As problems grow in size, CHP becomes a more attractive alternative. It finds tours with length closer to optimal and solves TSPs upwards of 4 times faster than LKH. For these reasons, CHP makes the most sense for large national TSP instances, specifically those with more than 10000 points in this case.

4.7.2 VLSI TSPs

Following the format of the national TSP results, the discussion of overall VLSI testing first focuses on the ratio of the CHP tour length to optimal tour length. Then, the solution quality and runtime of the CHP variants are compared to one another and that of LKH as problems grow in size.

The box plots in Figure 4.11 show summary statistics on the ratio between the heuristic and optimal tour lengths for each CHP variant and LKH. Additional statistics are also shown in Table 4.5. In both the figure

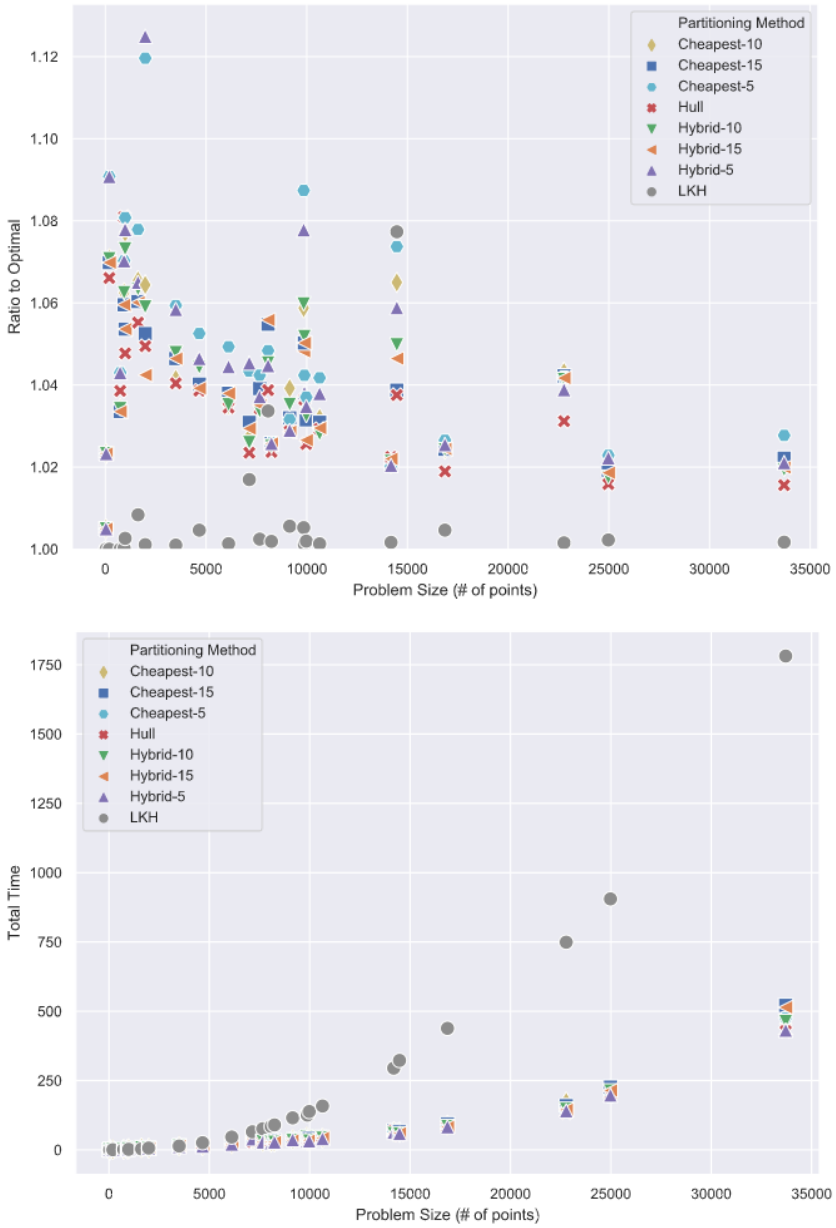


Figure 4.10: National TSP Results

and table, the values are split into two groups. One aggregates the results of tests across every VLSI instance. The other only considers instances with more than 20000 points.

The top chart, Figure4.11a, shows summary statistics aggregated from all VLSI instances. Figure 4.11b shows the same metrics when aggregated over only instances with at least 20000 points. As expected, LKH finds very good tours, with lengths typically within 0.5% of the optimal length, regardless of problem size. However, unlike the national TSP tests, the difference in CHP performance is very clear. CHP finds tours much closer to optimal when used to solve larger VLSI instances.

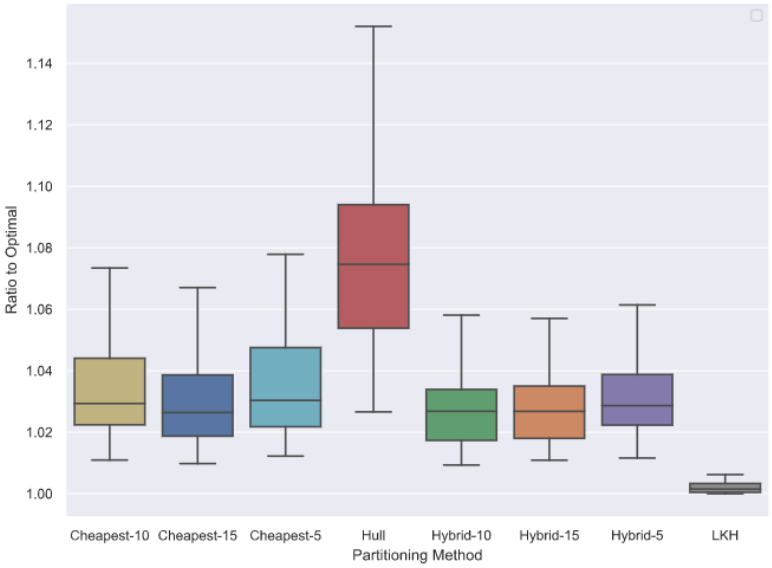
Based on the results of tests for insertion order, cheapest$-m$, and smaller VLSI instances, only cheapest-15, hybrid-10, and hybrid-15 were used to solve VLSI instances over 20000 points. Therefore, the discussion of summary statistics focuses on those three CHP methods. The most notable improvement is in worst-case performance. When aggregating across all instances, each variant demonstrated a worst-case of between 8.5% and 10%. However, when only instances with at least 20000 points are considered, the maximum optimality gaps are all less than 2.4%. In both cases, cheapest-15 has the highest of the maximum ratio to optimal length, and hybrid-10 has the lowest.

For instances with at least 20000 points, Cheapest-15 has the highest mean but the lowest median ratio to optimal length. This seems to show that while it finds good tours in many cases, it may find longer tours than the

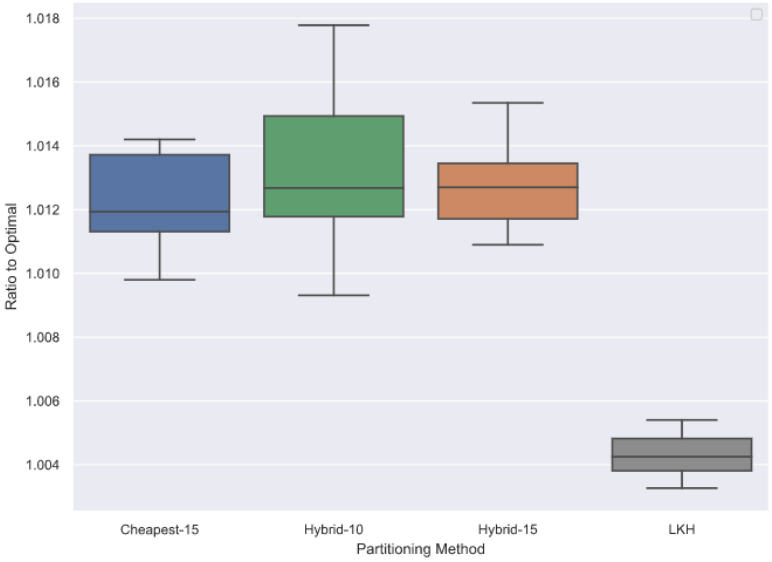
other variants for some instances. The two hybrid$-m$ variants perform very similarly. The only real way to differentiate the two is by the ranges of their results. Hybrid-15 has an overall higher range, with both higher minimum and maximum ratios. This means that it does not find solutions as good as those found through hybrid-10 in the best case and finds worse solutions in the worst case. However, hybrid-15 produces ratios with a smaller interquartile range. Both hybrid-10 and hybrid-15 produce tours with length within 1.5% of optimal for most large VLSI instances, and their summary statistics do not indicate one being clearly better than the other.

Looking at the LKH statistics, an opposite trend can be seen. While it still performs very well in all instances, both the average and median optimality gap increase when only considering larger instances. Additionally, the maximum ratio remains the same, indicating this was found on one of the larger instances. These results show that while the tours found through CHP seem to generally improve on larger instances, LKH seems to perform slightly worse. That said, LKH still finds better tours than any of the CHP variants. However, when considering only larger problems, the difference in performance between CHP and LKH decreases.

The length ratios and runtimes of the CHP variants and LKH for the VLSI test set are shown in Figure 4.12. The two charts on the left were again included to show overarching trends in performance and runtime as problem size increases. Interestingly, around 5000 nodes seems to be where the trend in both solution quality and runtime noticeably change to be more favorable



(a) All Instances



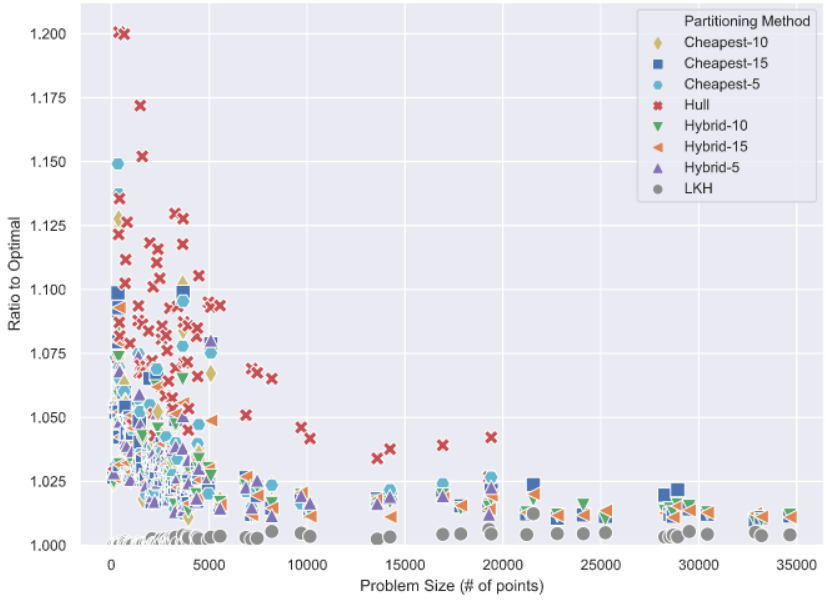
(b) Instances with ≥ 20000 points

Figure 4.11: VLSI TSP Tour Length Ratio Summary

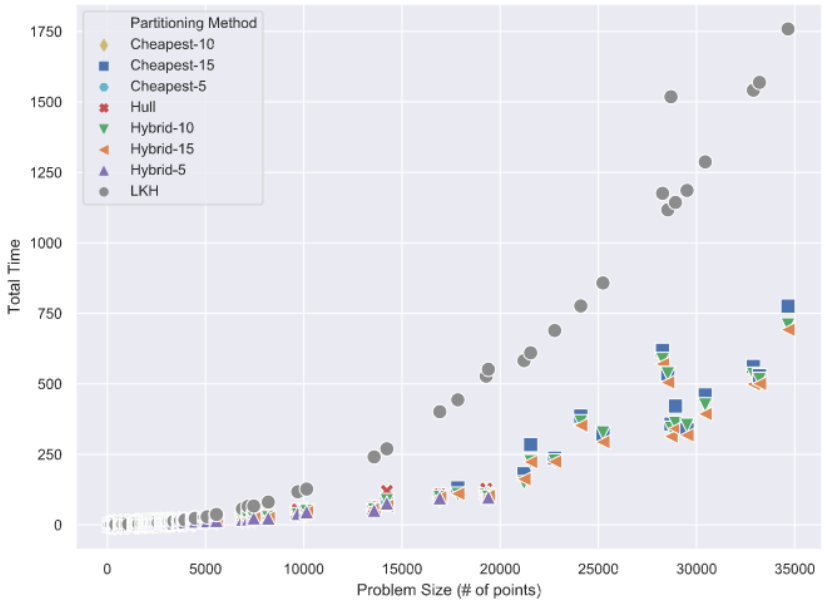
for CHP. Below 5000 nodes, using CHP probably does not save enough time to justify the decrease in solution quality. However, as instances get larger, the CHP variants run much faster than LKH. Again, the discussion focuses on cheapest-15, hybrid-10, and hybrid-15. Summary statistics for the ratio of CHP time to LKH time are shown in Table 4.6. Across all instances, the runtime improvements do not seem that great. Over half of the VLSI instances are less than 5000 nodes, which skews the overall statistics.

For instances larger than 20000 points, the CHP variants run 2-4 times faster than LKH. Of the three CHP variants, cheapest-15 is generally the slowest, hybrid-15 is generally the fastest, and hybrid-10 falls somewhere in the middle. All CHP variants run much faster than LKH. In the worst case, each variant runs about twice as fast as LKH. In the best case, they all run between 4 and 5 times faster. Finally, one of the best results is the 75^{th} percentiles for hybrid-10 and hybrid-15, which show that for at least 75% of instances with more than 20000 nodes, these CHP methods run in less than 40% of the time taken by LKH. Cheapest-15 is only slightly slower, as it solves most instances in less than 46% of the LKH runtime.

Looking at Figure 4.13, the performance and runtime of CHP and LKH are shown for only VLSI instances with at least 20000 nodes. The general trends remain the same, although maybe less pronounced. As expected from the aggregated results, Cheapest-15 produces the most variable tour length ratios, with hybrid-10 and hybrid-15 being more consistent across the tests. Additionally, the time chart shows that all three CHP variants run much



(a) Tour Length Ratio



(b) Runtime

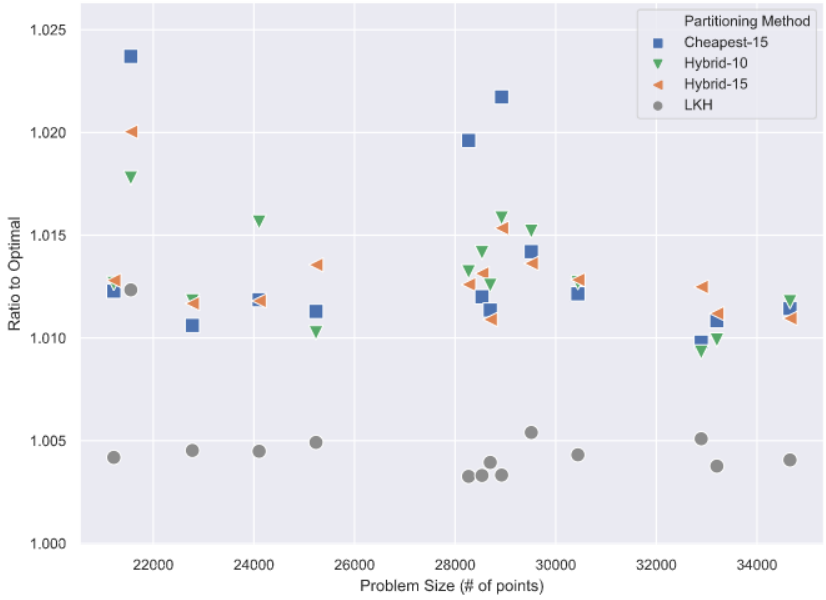
Figure 4.12: Results for all VLSI Instances

faster than LKH and match the expectations from the summarized results. Cheapest-15 and hybrid-15 usually have the slowest and fastest runtime, respectively.

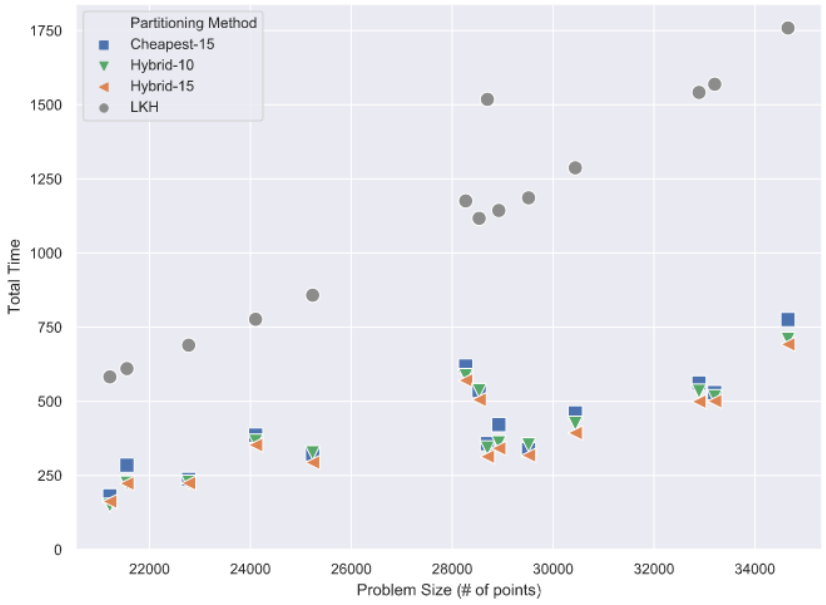
Based on the VLSI tour length ratio and runtime results, CHP performs well relative to the two outlined benchmarks, the best-known lengths and LKH. This is especially true as problem size grows. When looking at instances with at least 20000 points, the runtime improvement may justify the slight decrease in solution quality. Especially for large instances, solving a problem 2-3 times faster can be a significant amount of absolute time. Consider the largest VLSI instance tested. From Figure 4.13b, the LKH runtime is around 1750 seconds and the CHP variants solve the problem in around 750 seconds. The LKH tour has an optimality gap of just under 0.5% and the CHP variants find tours around 1.2% longer than the optimal tour. This means CHP found a solution less than 1% worse than LKH and took 1000 seconds, or roughly 16 minutes, less. Depending on the application, LKH may be the better tool, especially when trying to find the absolute best tour. However, if runtime is of any concern, these results make an argument for considering CHP.

4.8 Discussion

To recap the results, preliminary tests were conducted to examine the effects of LKH parameter values on both LKH and CHP, different insertion orders, and cheapest$-m$ (and hybrid$-m$) parameter values. Insights from these tests limited the number of tests for the largest instances. Testing included



(a) Tour Length Ratio



(b) Runtime

Figure 4.13: Results for VLSI Instances with more than 20000 Points

both national and VLSI instances, ranging in size from 29 to just under 35000 points. Based on LKH parameter testing, all further testing used fast LKH. After testing cheapest$-m$ variants, cheapest$-m$ or hybrid$-m$ used m values of 5, 10, and 15 in any later test. Finally, partitioning methods used an ascending insertion order. Using a descending produced better tours in some cases but led to a prohibitively long runtime. These results helped determine which CHP variants to test on larger problems. Because there are fewer national TSP instances, cheapest-5, cheapest-10, cheapest-15, hull, hybrid-5, hybrid-10, and hybrid-15 partitioning were tested. The same set of CHP variants were used for VLSI instances with less than 20000 points. The VLSI test set contains many more large instances. For that reason, tests on instances with more than 20000 points included only cheapest-15, hybrid-10, and hybrid-15 partitioning. Again, all CHP variants used an ascending insertion order and fast LKH to solve subproblems in the final testing stage.

Several CHP variants performed very well relative to both the best-known tour lengths and LKH, especially for large problems. For the tested national TSPs, hull partitioning seems to perform the best. For VLSI instances, cheapest-15, hybrid-10, and hybrid-15 all seem to produce very good solutions. The difference in the best methods between the two problem sets is likely due to the difference in the shape of their convex hull. Unexpectedly, hull partitioning appears to lead to better solutions when the convex hull of a TSP's point set is more irregular, such as in national TSPs. This may be caused by the large areas of low point density seen in many national problems.

Without very many points, the proximity-based assignment methods do not have as many nearby points for comparison. The opposite effect may be seen for the VLSI instances, which typically consist of very densely packed points. In this type of problem, cheapest$-m$ and hybrid$-m$ perform better.

As noted in the recaps of national and VLSI testing, CHP performs well relative to both benchmarks, especially on large problems. For problems with more than 20000 points, CHP seems to find better tours for VLSI instances rather than national problems. However, there were not as many large national instances, which may affect this conclusion. As noted when discussing both problem sets, two trends are seen in CHP performance as problem size increases. First, CHP produces tours closer to the optimal (or best-known) length. Second, the difference between LKH and CHP runtime grows. This means that CHP becomes a more effective alternative to LKH as problems grow large. These trends are most likely because the number of partitions does not grow proportionally to the number of points. This means partitions contain more points. When points are distributed to fewer partitions, they are more likely to be assigned to the correct one. Because LKH is used to solve subproblems, they are solved to a high degree of accuracy. With many points assigned to the correct partition and an accurate subproblem solver, CHP produces a high-quality final tour. One downside of larger subproblems is that they take longer to solve. However, because the runtime of LKH grows very quickly with the number of points, solving several moderately large subproblems is still faster than a single much larger problem.

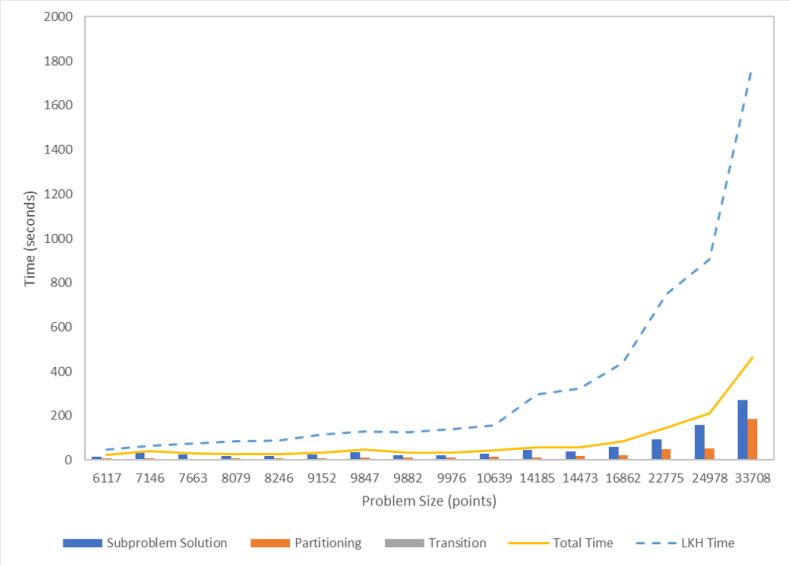
4.9 Limitations and Future Work

The bulk of the CHP framework is implemented in Python 3.6.2. The LKH and Concorde solvers used to solve subproblems are written in C, with binaries available online through the authors' websites [Cook, a],[Helsgaun, a]. The interplay between the Python and C portions exists as a major limitation of this implementation but building an integrated C program was beyond the scope of this work. As it stands, all partitioning functionality is implemented in Python. For each partition, a TSP file containing that partition's points is saved and used as input for one of the solvers. An instance of the solver runs for each partition TSP file, producing the subproblem Hamiltonian paths. The solver saves these paths as output files before to be read by Python, appended together, and returned as a final tour. A runtime breakdown of each portion of CHP using hybrid-10 partitioning is shown in Figure 4.14. The charts show the total CHP time (solid line), fast LKH time (dashed line), and time taken by different stages of CHP (columns) for different size instances. The different columns show the time taken for each portion of CHP. Subproblem solution refers to the time taken to solve subproblems using LKH. Partitioning refers to the time taken to initialize and assign points to groups. Finally, transition is the time required for saving and writing files to use with LKH. As shown in the figure, most of the time is taken by solving subproblems. The next biggest contributor is partitioning. Transition time is not a significant portion of the total solution time of CHP. However, any additional time required to move between Python and C is time not spent actively moving towards a

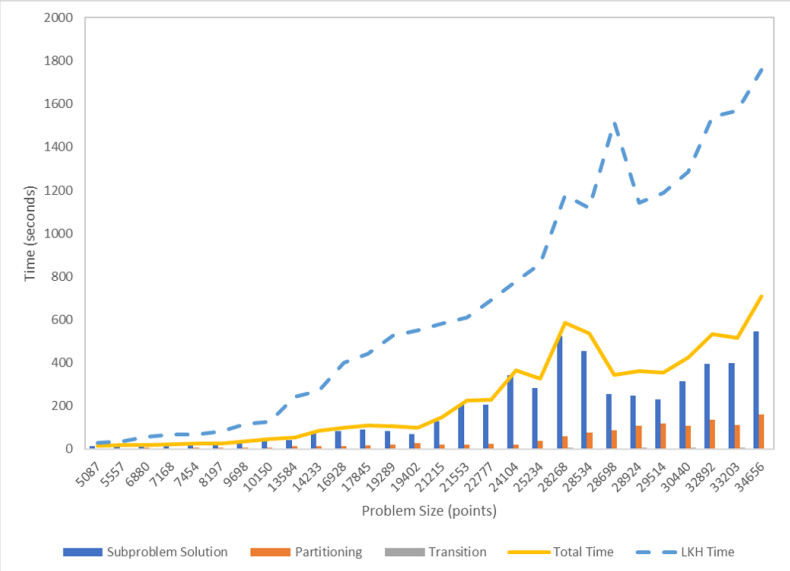
solution. Additionally, Python code runs more slowly than programs written and compiled in C. The speed difference between Python and C and the time spent on file input and output seem to be low-hanging fruit in terms of runtime improvement. While it was beyond the scope of this dissertation, a more integrated implementation of CHP and LKH all written in C would lead to further runtime improvements. Given the partitioning heuristics already built into LKH, integrating CHP seems like a good avenue for future work.

Beyond making it more integrated, a more efficient implementation of CHP could probably be written. The current version performs well when compared to existing heuristics, but further improving the implementation could only help performance. Another implementation-based area of future work lies in a parallel version of CHP. Because they do not rely on results from one another, the subproblems can be solved in parallel before combining them to find a final tour. Again, programming CHP as efficiently and effectively as possible was not the primary focus, and therefore, making implementation improvements exists as another major area of future work.

Another factor limiting the runtime improvement seen through using CHP is the number of partitions and its dependence on the size of the convex hull boundary of a given problem. As problems grow larger, the number of interior points typically grows much more quickly than the number of hull points. To better demonstrate this, a chart of the number of hull points as problem size increases is shown in Figure 4.15. In this chart, blue circles and orange x markers represent national and VLSI TSPs, respectively. Clearly, the



(a) National TSPs



(b) VLSI TSPs

Figure 4.14: CHP Runtime Breakdown

number of hull points does not scale with the number of points overall. Additionally, national TSPs tend to have more hull points than VLSI instances. This trend could be another factor leading to national TSPs being generally faster to solve with CHP than VLSI instances. CHP forms the same number of partitions as the number of hull points. This means that larger TSP instances have more points in each partition. For example, consider the TSP instances $dj38$, $ar9152$, and $bm33708$ which consist of 38, 9152, and 33708 nodes, respectively. Similarly, $dj38$ has 8 hull points, $ar9152$ has 18, and $bm33708$ has 22. If points were evenly distributed across their partitions, $dj38$ would have partitions containing either 4 or 5 points and $bm33708$ would have partitions containing around 1532 points. Scaling the number of partitions more closely with the number of points in a TSP instance would lead to smaller partitions, faster subproblem solutions, and a lower overall runtime. However, as CHP is based on using the convex hull points as the basis for partitions, it does not provide an obvious way to increase the number of groups. Thus, extending CHP to generate more partitions is another potential area for future research. The premise of any such extension is to determine a larger set of points than the convex hull boundary that still has the benefit of a known order in the final tour. Augmented CHP, described in Chapter 5, attempts to increase the number of partitions, but the results were mixed and more work remains to be done.

More work could also be done developing other partitioning methods for use with CHP. These could include more traditional clustering methods,

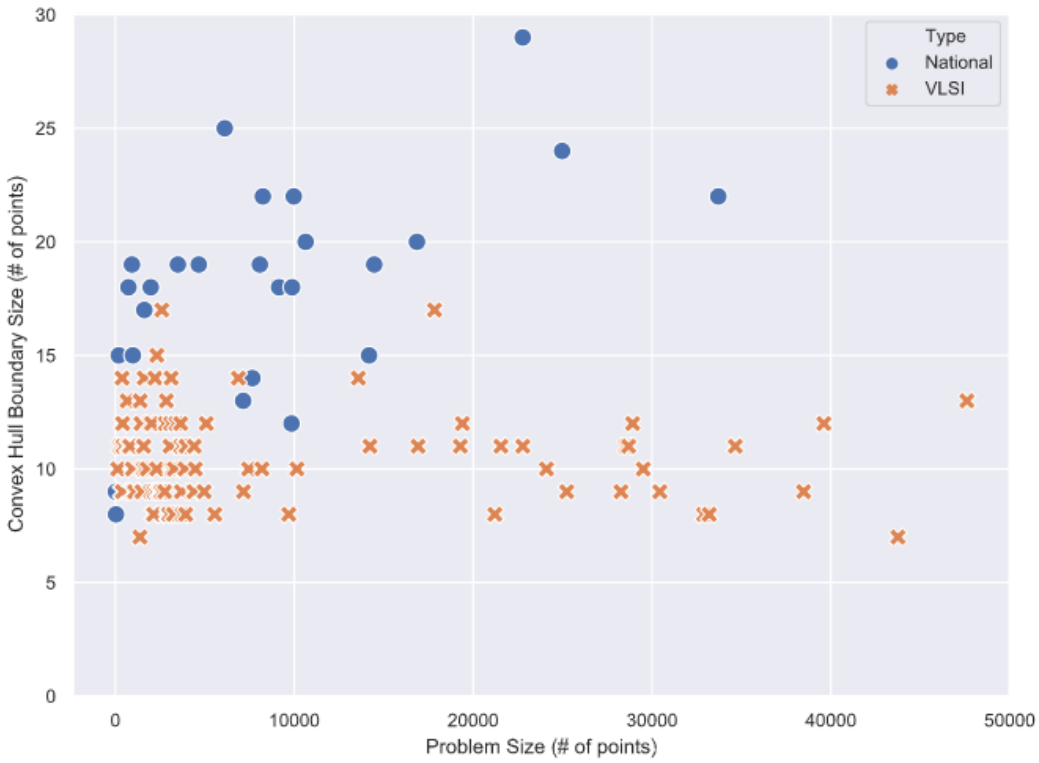


Figure 4.15: Convex Hull Size as Problem Size Increases

such as k-means or DBSCAN [Tan et al., 2019]. In early testing, CHP using k-means did not perform as well as with other partitioning methods. For that reason, it was not tested further. DBSCAN uses several ambiguous parameters, which are difficult to choose without running several iterations of the algorithm. Additionally, the results of DBSCAN parameters may vary widely across different problem instances. This work focuses on finding methods that worked reasonably well across many TSP instances; for that reason, testing DBSCAN was not pursued. However, partitioning methods other than those presented in this work may lead to a better final tour. As was shown through the results in Section 3.3, partitioning plays an incredibly important role in determining the quality of the output tour. Developing partitioning methods that get increasingly close to assigning points to the correct groups will lead to increasingly better tours.

One final area of potential work lies in extending CHP to other TSP variants. As implemented, CHP only solves Euclidean TSPs. Ideally, this could be extended to include asymmetric TSPs or TSPs through incomplete networks. The current implementation requires input point coordinates and calculates distance between each pair of points. Modifying CHP to use an input distance matrix would allow for asymmetric distances. To accommodate incomplete graphs, place-holder values could be used to indicate edges that do not exist.

4.10 Results Tables

The tables in this section contain the results of testing on the CHP extensions described in Chapter 4. The tables come in two forms, one containing summary statistics and another containing all test results.

The summary statistic tables aggregate the tour length ratios of the tests up to the relevant level. For example, Table 4.3 groups the tests by problem set and partitioning method before aggregating the results. This allows examination of the effectiveness of each partitioning method for different instance types. The summary statistics recorded include count, mean, standard deviation, minimum, 25^{th} percentile, median, 75^{th} percentile, and maximum value of the tour length ratio.

The second table type contains raw test results, with each row representing an individual test. Each column represents either a characteristic of the CHP variant used in the test or a metric used to examine the results.

Type	Partitioning Method	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
National	Cheapest-2	1.091189	0.038198	1.035063	1.063331	1.079214	1.115424	1.155091
	Cheapest-3	1.068462	0.038369	1.004963	1.043802	1.081011	1.085916	1.14349
	Cheapest-5	1.058122	0.031802	1.004963	1.040099	1.055951	1.078608	1.119621
	Cheapest-10	1.046971	0.021509	1.004963	1.034614	1.043514	1.064647	1.077072
	Cheapest-15	1.042607	0.018153	1.004963	1.033149	1.043329	1.055097	1.069825
	Cheapest-25	1.043423	0.019344	1.004963	1.034433	1.041337	1.052711	1.075544
VLSI	Cheapest-2	1.064157	0.03407	1.011938	1.039239	1.057378	1.079476	1.167398
	Cheapest-3	1.050576	0.028365	1.015288	1.029963	1.041957	1.064156	1.156156
	Cheapest-5	1.041045	0.025829	1.012227	1.025265	1.032674	1.04918	1.149123
	Cheapest-10	1.039794	0.023886	1.010944	1.024264	1.032607	1.047651	1.127628
	Cheapest-15	1.038909	0.021421	1.015646	1.025531	1.032088	1.04866	1.098844
	Cheapest-25	1.039226	0.02084	1.015749	1.024719	1.034089	1.046722	1.11038

Table 4.1: Cheapest$-m$ Testing Tour Length Ratio Summary

Type	Partitioning Method	Sorting Method	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
National	Cheapest-10	Ascending	1.041186	0.018656	1.004963	1.025618	1.038571	1.056893	1.077072
		Descending	1.036117	0.020573	1.007662	1.023854	1.032534	1.043436	1.113393
	Cheapest-5	Ascending	1.04942	0.026486	1.004963	1.028692	1.042689	1.067527	1.119621
		Descending	1.038139	0.026995	1.007662	1.022251	1.031847	1.044799	1.14894
	Hull	Ascending	1.035077	0.016494	1.004963	1.023554	1.034399	1.040006	1.080938
		Descending	1.163679	0.584803	1.007662	1.030801	1.040799	1.066192	4.027742
	Hybrid-5	Ascending	1.046419	0.026143	1.004963	1.02656	1.040921	1.058732	1.124938
		Descending	1.197102	0.82045	1.007662	1.024879	1.033866	1.0435	5.218511
VLSI	Cheapest-10	Ascending	1.037185	0.023367	1.010944	1.022405	1.029338	1.044051	1.127628
		Descending	1.035543	0.0178	1.014519	1.022901	1.029388	1.039479	1.091266
		Shifted	1.03477	0.015228	1.012653	1.023352	1.032416	1.042573	1.074561
	Cheapest-5	Ascending	1.038316	0.025326	1.012227	1.021777	1.030348	1.047526	1.149123
		Descending	1.03777	0.01983	1.016101	1.02596	1.031888	1.044999	1.115616
	Hull	Ascending	1.081401	0.035384	1.026614	1.05384	1.074608	1.094015	1.200625
		Descending	1.057326	0.02565	1.019373	1.038109	1.050612	1.074301	1.126758
		Shifted	1.069798	0.028619	1.023816	1.047005	1.068419	1.089452	1.17394
	Hybrid-5	Ascending	1.032756	0.015831	1.011595	1.02231	1.028659	1.03878	1.080206
		Descending	1.032676	0.015985	1.011988	1.020989	1.028705	1.037769	1.107949
		Shifted	1.035883	0.015507	1.012187	1.025589	1.033184	1.040885	1.088645

Table 4.2: Insertion Order Tour Length Ratio Summary

	Method	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
All Instances	Cheapest-10	1.041186	0.018656	1.004963	1.025618	1.038571	1.056893	1.077072
	Cheapest-15	1.038286	0.015274	1.004963	1.027033	1.038353	1.050201	1.069825
	Cheapest-5	1.04942	0.026486	1.004963	1.028692	1.042689	1.067527	1.119621
	Hull	1.035077	0.016494	1.004963	1.023554	1.034399	1.040006	1.080938
	Hybrid-10	1.039619	0.017852	1.004963	1.025288	1.035241	1.051321	1.073192
	Hybrid-15	1.03743	0.015426	1.004963	1.025948	1.036518	1.047783	1.069825
	Hybrid-5	1.046419	0.026143	1.004963	1.02656	1.040921	1.058732	1.124938
	LKH	1.006872	0.015982	1	1.001122	1.0018	1.004637	1.077333
$\geq$ 10000 points	Cheapest-10	1.032186	0.016759	1.01874	1.021068	1.024381	1.037525	1.064994
	Cheapest-15	1.028393	0.009092	1.019253	1.021653	1.024324	1.034824	1.042223
	Cheapest-5	1.036391	0.01857	1.020237	1.0247	1.027722	1.041826	1.073724
	Hull	1.024412	0.008434	1.0156	1.017405	1.022456	1.030227	1.037562
	Hybrid-10	1.02883	0.012183	1.017526	1.020392	1.024236	1.034728	1.04981
	Hybrid-15	1.028938	0.011005	1.018606	1.020951	1.024413	1.035598	1.046445
	Hybrid-5	1.032117	0.014103	1.020427	1.02166	1.025465	1.038385	1.058839
	LKH	1.012915	0.028428	1.001301	1.001595	1.001708	1.003437	1.077333

Table 4.3: National TSP Tour Length Ratio Summary

	Method	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
All Instances	Cheapest-10	1.020668	1.587578	0.197901	0.26291	0.36986	0.952536	7.565797
	Cheapest-15	1.044192	1.42567	0.206821	0.293868	0.382647	1.059974	5.993854
	Cheapest-5	0.884996	1.171526	0.186241	0.246829	0.367992	0.970407	5.266605
	Hull	1.04677	1.445117	0.169943	0.258339	0.361881	1.125991	6.364203
	Hybrid-10	0.978889	1.461294	0.17955	0.262125	0.379157	0.978219	6.915064
	Hybrid-15	0.954922	1.349047	0.185441	0.272508	0.352767	0.970509	6.436971
	Hybrid-5	1.064255	1.751422	0.182562	0.245385	0.362286	0.875882	6.88121
$\geq$ 10000 points	Cheapest-10	0.231735	0.025457	0.197901	0.214545	0.230513	0.24887	0.266902
	Cheapest-15	0.241982	0.03539	0.206821	0.213479	0.231732	0.267873	0.29262
	Cheapest-5	0.216305	0.029656	0.186241	0.193635	0.211561	0.231217	0.266631
	Hull	0.217102	0.041132	0.169943	0.184044	0.221407	0.236666	0.286943
	Hybrid-10	0.218911	0.035323	0.17955	0.194171	0.201231	0.247397	0.268461
	Hybrid-15	0.222895	0.042782	0.185441	0.190439	0.197745	0.253848	0.288509
	Hybrid-5	0.212802	0.028114	0.182562	0.188596	0.214659	0.23027	0.254664

Table 4.4: National TSP Time Summary Statistics

	Method	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
All Instances	Cheapest-10	1.037185	0.023367	1.010944	1.022405	1.029338	1.044051	1.127628
	Cheapest-15	1.032705	0.021501	1.009798	1.018716	1.026422	1.038641	1.098844
	Cheapest-5	1.038316	0.025326	1.012227	1.021777	1.030348	1.047526	1.149123
	Hull	1.081401	0.035384	1.026614	1.05384	1.074608	1.094015	1.200625
	Hybrid-10	1.029287	0.015457	1.009312	1.017345	1.026814	1.033907	1.08587
	Hybrid-15	1.029297	0.015859	1.010898	1.018005	1.026794	1.035009	1.092896
	Hybrid-5	1.032756	0.015831	1.011595	1.02231	1.028659	1.03878	1.080206
	LKH	1.002074	0.002018	1	1.000438	1.00152	1.003319	1.012341
$\geq$ 20000 points	Cheapest-15	1.013777	0.004469	1.009798	1.01131	1.011933	1.01372	1.023705
	Hybrid-10	1.013057	0.002452	1.009312	1.011779	1.012676	1.014935	1.017782
	Hybrid-15	1.013067	0.002341	1.010898	1.011709	1.012701	1.013452	1.020037
	LKH	1.004782	0.002274	1.003264	1.003809	1.004251	1.00482	1.012341

Table 4.5: VLSI TSP Tour Length Ratio Summary

	Method	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
All Instances	Cheapest-10	0.863883	0.397004	0.184232	0.59205	0.850054	1.153819	1.855708
	Cheapest-15	0.812852	0.439144	0.174568	0.42438	0.729908	1.153831	1.826261
	Cheapest-5	0.821678	0.392392	0.16932	0.546907	0.772005	1.09762	1.743561
	Hull	0.949595	0.420837	0.177663	0.632992	0.911028	1.270457	1.990936
	Hybrid-10	0.776813	0.422414	0.178167	0.376185	0.66188	1.081566	1.770218
	Hybrid-15	0.741827	0.411472	0.186958	0.380672	0.660022	1.043354	1.888237
	Hybrid-5	0.798618	0.369081	0.177687	0.505896	0.773647	1.112434	1.538877
$\geq$ 20000 points	Cheapest-15	0.384945	0.085423	0.235317	0.338761	0.366494	0.46008	0.526377
	Hybrid-10	0.359009	0.081275	0.225493	0.317993	0.338406	0.397266	0.498538
	Hybrid-15	0.343978	0.078723	0.206485	0.300095	0.324222	0.386352	0.484926

Table 4.6: VLSI TSP Time Summary Statistics

Chapter 5

Extensions of CHP

The standard CHP implementation is designed to produce a high quality TSP solution in a reasonable amount of time. Based on testing, CHP performs well in that capacity. However, improvements can still be made. Three extensions to the standard CHP implementation are described in this section. First, a recursive implementation attempts to solve TSPs even more quickly. Second, augmented CHP uses the same general framework but increases the number of partitions by using more points as partition bases. Finally, tests are run using an improvement heuristic on the output CHP tour to lower the final tour length.

5.1 Recursive Implementation

A recursive implementation of the CHP framework attempts to provide a further improvement in solution time. Algorithm 14 shows an outline of recursive CHP. The implementation begins with a specified recursion limit. This value sets the maximum size of any partition. This also limits the maximum size of any Hamiltonian path problem solved by the subproblem method. This is useful because as problem instances grow, the partition size also grows,

leading to the subproblems themselves being large. The trade-off of recursive CHP is a drop in solution quality as partitioning plays a larger role in determining the overall solution. Recursive CHP initially partitions points exactly the same as non-recursive CHP. Then, a recursive call of CHP is performed on any partition larger than the recursion limit. For any group smaller than the limit, the subproblem is solved and its solution is incorporated into the output tour.

Input: N: set of points, γ: recursion limit
Output: T: heuristic tour
$T = [\,]$;
Q = partitioning of N;
for *partition* $q \in Q$ **do**
    N_q = points $\in q$;
    if $|N_q| \leq \gamma$ **then**
        p_q = Hamiltonian path through N_q;
    else
        $p_q = RecursiveCHP(N_q, \gamma)$;
    end
    Append p_q to T;
end
Return T;

Algorithm 14: Recursive CHP

5.1.1 Recursive CHP Experiments

Recursive CHP places an upper limit on the number of points in a subproblem being solved by LKH. A recursive call of CHP is performed on any partition with more points than the recursion limit. The recursion level dictating that upper limit is an input parameter and affects the final solution.

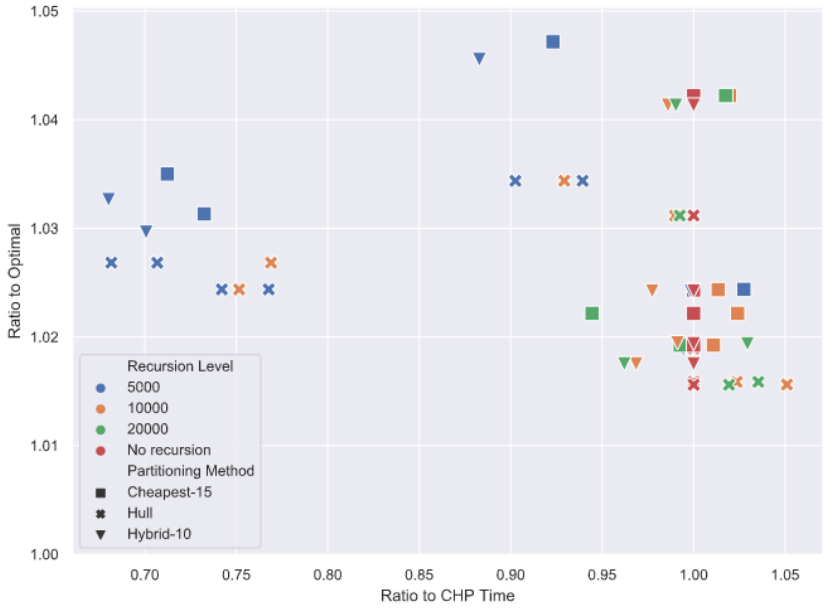
Because the heuristic nature of CHP largely stems from imperfect partitioning, a lower recursion tends to lead to longer final tours. This is because partitioning plays a larger role in recursive CHP. Every time CHP is called within a group of points, an additional layer of partitioning occurs and the likelihood of error increases. However, solving large subproblems with LKH is the biggest contributor to the overall runtime of CHP. In theory, the recursive version helps decrease runtime by shifting work to the faster partitioning algorithms and decreasing the size of subproblems solved by LKH.

LKH is capable of quickly solving small TSP instances, and CHP runs significantly faster only as problems grow larger. Recursive CHP is expected to exhibit the same trend because there is more room for runtime improvement on larger instances. For that reason, tests consist of a comparison between standard CHP, recursive CHP, and LKH for instances with more than 15000 points. Testing included recursion levels of 5000, 10000, and 20000 points and both national and VLSI problems. If the recursion level is set higher than the number of points in the TSP instance being solved, then recursive CHP functions exactly as the standard implementation. For example, consider solving the Finland instance (fi10639), which has 10639 points. If a recursion level of at least 11000 is used, then recursive CHP runs exactly as standard CHP. However, if a recursion level of 10000 is used, then a recursive call may occur.

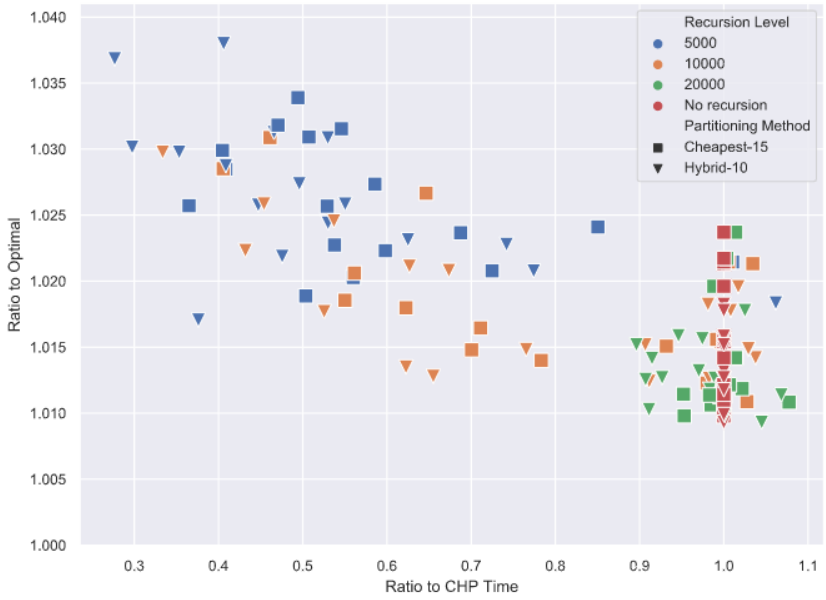
Summary statistics of the recursion level tests are shown in Tables 5.1 and 5.2. These tables contain tour length ratio and runtime comparisons,

respectively. Additionally, a summary of the recursive CHP results is shown in Figure 5.1. The results are split into national and VLSI instances, which are shown in Figures 5.1a and 5.1b, respectively. These figures show recursive CHP's relative performance to standard CHP. The horizontal axis shows the ratio to CHP time. The vertical axis again shows the ratio to the optimal tour length. In this case, different colors represent the tested recursion levels, with markers indicating a different solution method. Plotting these metrics for each test instance helps identify trends in relative performance between the two methods. Because the primary interest is examining the effects of recursion limit, these are shown in different colors.

In both cases, recursive CHP leads to a runtime improvement over traditional CHP when using a recursion limit of 5000. When used to solve VLSI instances, recursive CHP leads to shorter runtimes for recursion limits of both 5000 and 10000. As expected, using lower recursion limits leads to bigger runtime improvements. In most cases, a 20000-point recursion limit does not lead to any significantly lower runtime compared to traditional CHP; the best case of any variant with a 20000 limit ran in around 90% of the traditional CHP runtime. The biggest improvement comes from using a recursion level of 5000. For either type of problem, recursive CHP with a limit of 5000 solves TSPs in less than 80% of the time of traditional CHP, on average. Recursive CHP runs even faster for VLSI problems. With a limit of 5000 points, all VLSI instances were solved in at most 72.5% of CHP time regardless of partitioning method. On average, they were solved in around half of traditional CHP's



(a) National TSPs



(b) VLSI TSPs

Figure 5.1: Recursive CHP Summary Results

total time. When using a limit of 10000 points, the runtime improvement is less; recursive CHP solves VLSI problems in an average of 75% and 68% of CHP time for cheapest-15 and hybrid-10 partitioning, respectively. However, this can still be a large improvement in absolute time for large instances.

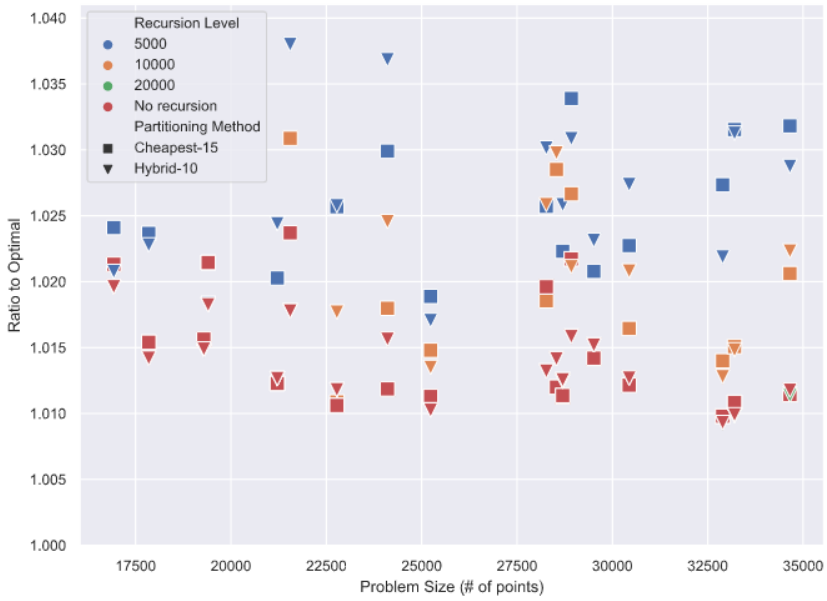
The increased solution speed comes at the cost of solution accuracy, albeit a small amount in most cases. Again, for national TSPs, only a recursion level of 5000 points made an impact on the results. In this case, the average optimality gap increased by around 1% for the tested partitioning methods. The tour length ratios for VLSI instances were changed for recursion levels 5000 and 10000. For the 5000-point case, average and median optimality gaps increased by about 1.5% over the no recursion case. As expected, this increase was less when using a recursion level of 10000 points. Across all VLSI instances, partitioning methods, and recursion levels, the worst-case solution was 3.8% above optimal. For at least $\frac{3}{4}$ of VLSI instances, the optimality gap was just over 2% when using a recursion level of 10000 points.

In summary, lower recursion limits lead to faster runtimes but longer tours. These trends can be seen in Figure 5.1. The red markers show the results of standard CHP on each instance and serve as the benchmark. The blue and orange markers, showing 5000- and 10000-point limits, tend to have x-values less than 1, indicating a lower runtime than CHP. However, they also typically have a higher y-value, which means a tour further from the optimal length. On the other hand, the green markers showing tests with a recursion limit of 20000 points typically appear near the baseline CHP markers. This

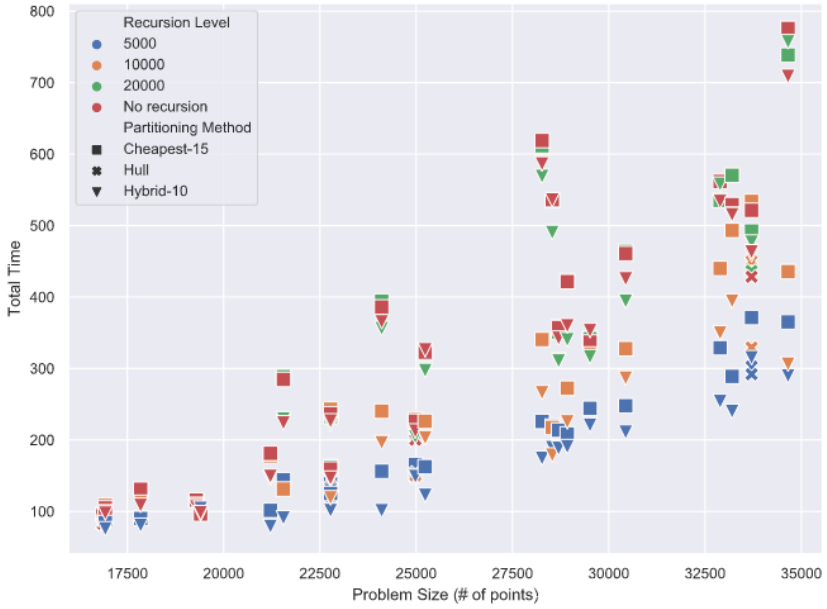
shows a recursion limit of 20000 finds tours of about the same length as CHP in about the same time.

The tour length and runtime trends described can be seen in Figure 5.2, which shows a summary of the VLSI test results. Clearly, the tour length ratio decreases as the recursion level increases. On the other hand, the ratio to CHP time is lower for smaller recursion levels as these variants run faster.

The charts in Figure 5.2 show a comparison of CHP, recursive CHP, and LKH. Clearly, LKH finds the best tours. However, as problems grow in size, CHP and recursive CHP solve problems significantly faster. The trends shown in Figure 5.1 can be seen here as well. In Figure 5.2a, markers indicating a lower recursion limit are higher on the chart, corresponding to a tour length further from optimal. However, the 5000- and 10000-point limit markers in Figure 5.2b have the lowest y-values, meaning recursive CHP using smaller limits run faster than standard CHP and LKH. Consider the instance with just below 35000 points. LKH finds the tour closest to optimal, followed by standard CHP and recursive CHP with a limit of 20000 points. A limit of 10000 leads to solutions just over 2% above optimal, and a recursion limit of 5000 results in the longest tours, about 3% above optimal. The trend reverses in the runtime chart. LKH is the slowest method by far. Standard CHP and recursive CHP with a 20000-point limit run in about 1000 seconds less than LKH. Recursive CHP with 5000- and 10000- point limits run the fastest. They take almost 1500 fewer seconds less than LKH and run in $\frac{1}{3}$ to $\frac{1}{2}$ the time of standard CHP.



(a) Tour Length Ratio



(b) Ratio to CHP Time

Figure 5.2: Recursive CHP Results as Problem Size Increases

Depending on the recursion limit, recursive CHP solves TSPs more quickly but results in slightly longer tours. Based on the conducted tests, a recursion limit set to no more than half the total number of points typically leads to recursive calls. Decreasing the recursion level leads to faster solution times but finds longer tours. These results show that recursive CHP can be a useful tool for solving TSPs. Whether or not to use recursive CHP depends on the priorities of the user. If speed is most important, recursive CHP seems to solve TSP more quickly than CHP and thus also LKH. If accuracy is top priority, then LKH is hard to beat. Finally, CHP seems to be a happy medium of the two.

5.2 Augmented Convex Hull Partitioning

As discussed in Section 4.9, one limitation of CHP is its reliance on the convex hull boundary to determine the number of partitions used to form subproblems. Because the size of the convex hull boundary does not scale proportionally to the number of points in a TSP instance, even the Hamiltonian path subproblems can themselves be large. This means that while CHP is relatively fast compared to other methods, it can still take a large amount of time in an absolute sense. In theory, increasing the number of partitions leads to solving smaller subproblems and thus a lower overall solution time. The method presented here, augmented CHP, is one way to produce more partitions. As the name implies, augmented CHP picks a subset of interior points to augment the convex hull boundary when initializing partitions. An outline of

the method is shown in Algorithm 15. The procedure is the same as standard CHP except for two additional steps occurring between finding the convex hull and assigning points to partitions. Once the convex hull has been found, a subset of interior points is chosen. Then, a Hamiltonian tour through the hull points and interior point subset is found. This tour then replaces the convex hull boundary as the basis for forming partitions. Instead of a partition for each pair of consecutive hull points, in augmented CHP, a partition is formed for each consecutive of points in the augmented subset tour. Once partitions have been initialized, the process for assigning points, solving subproblems, and forming a final tour is the same standard CHP.

Input: N: set of points
Output: T: heuristic tour
H = convex hull boundary of N;
I = points in N not in H;
$\hat{I}$ = subset of I chosen to augment H;
$\hat{H}$ = Hamiltonian tour through points in $H \bigcup \hat{I}$;
Initialize partitions by forming one for each pair of consecutive points in $\hat{H}$;
Assign points to partitions;
Solve Hamiltonian path through each partition;
T = join partition paths using points in $\hat{H}$; Return T;

Algorithm 15: Augmented CHP

How to choose points to augment the convex hull is another factor in augmented CHP. Four potential methods for making this selection are ascending order, descending order, random, and iterative convex hulls. Ascending and descending order are very similar to the insertion orders when forming

partitions. Ascending selection adds the nearest points to a hull point until the desired number is reached. Descending selection adds the furthest points first. Unsurprisingly, random is a random selection of interior points. Iterative convex hulls is the most intensive. It begins by finding the overall convex hull. Then, the convex hull of the interior points is found. Points in the boundary of this inner convex hull are used to augment the overall set of hull points. If the desired number has been reached, then the algorithm moves forward to find an order. If too few points have been selected, the process continues by finding convex hulls of successively smaller interior point sets until enough points have been chosen.

5.2.1 Augmented CHP Experiments

For each national and VLSI instance, augmented CHP was run using 20, 30, and 50 partitions. This testing includes only ascending selection. Iterative, descending order, and random selection were not included because they either found significantly worse tours or took much longer to run in preliminary testing. Augmented CHP performs relatively well for instances with more than 10000 points. For problems smaller than that threshold, augmented CHP still finds solutions within 10% of optimal but typically takes more time than CHP. Standard CHP itself does not lead to significant runtime improvement until problems become larger. For this reason, the augmented CHP results discussed include only problems with more than 10000 points. Aggregations of the tests are shown in Tables 5.3 and 5.4. These tables show summary

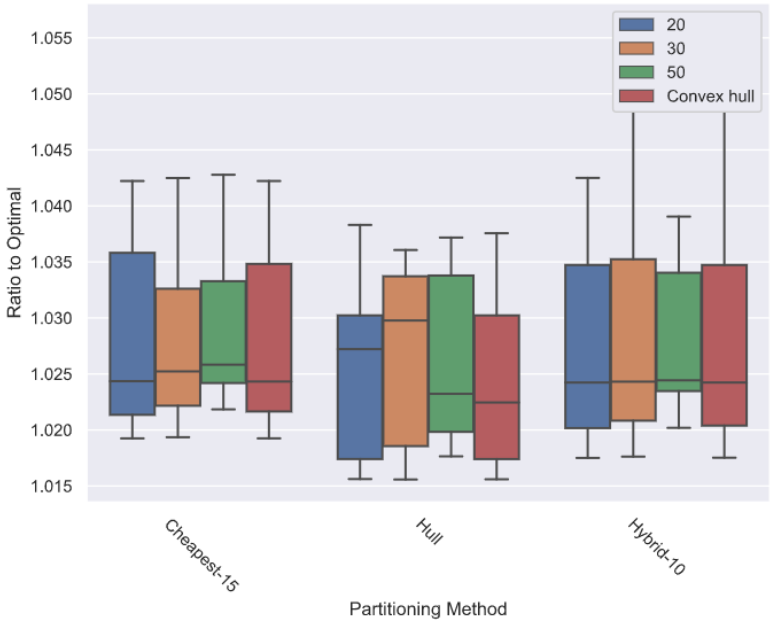
statistics of the tour length ratio and ratio to CHP time, respectively. When determining the usefulness of augmented CHP, the optimal (or best known) tour length still makes sense as a benchmark. However, in this case CHP replaces LKH as the time benchmark. This allows for comparison between augmented and traditional CHP. Clearly, the hope was that augmented CHP finds better tours in less time than CHP. Unfortunately, that does not seem to be the case in general. Charts comparing the tour length ratio of different augmentation levels are shown in Figure 5.3. In this case, different colors indicate a different number of partitions used to solve the TSP. Results are again split into the national and VLSI categories. Augmented CHP seems most effective for VLSI instances. Specifically, augmented CHP finds better solutions than standard CHP when using hybrid-10 partitioning. For national instances, augmented CHP finds solutions comparable to standard CHP for most variants.

A few trends can be clearly seen in the results. First, the variants with 50 partitions usually found longer tours, although not significantly. The median optimality gap is still within 2.5% for most variants. For VLSI instances, hybrid-10 partitioning seems like the obvious partitioning method to pair with augmented CHP. For all three partition numbers, it found better tours than standard CHP in at least half of the instances. Additionally, augmented CHP typically takes about the same time as standard CHP to solve VLSI instances. On the other hand, augmented CHP performs less well on national instances. It typically takes slightly longer than standard CHP and does not seem to find

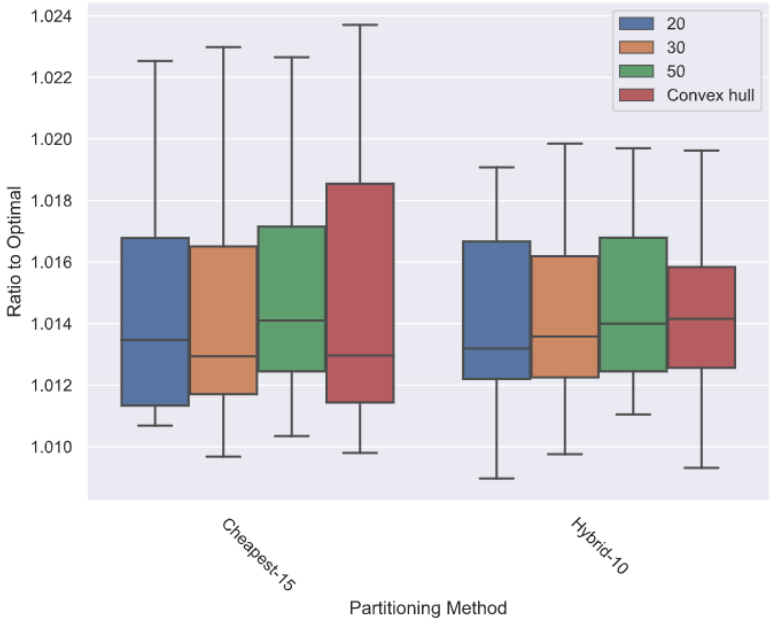
better tours in general. Specifically, hull partitioning does not seem to pair well with augmented CHP. The number of partitions does not seem to make a major difference in tour quality. However, increasing the number or partitions too much seems to decrease the quality of the final solution. Typically, using 20 or 30 partitions lead to better tours than 50 partitions. Another interesting trend is that the interquartile range is lower for some augmented CHP variants than standard CHP.

Figures 5.4 and 5.4 show the tour length ratio and runtime of augmented and standard CHP as problems grow in size. Figures 5.4a and 5.5a show the tour length ratio of national and VLSI instances. Figures 5.4b and 5.5b show runtimes of the various methods. As in the box plots, different colors represent different augmentation levels. In the scatter plots, different marker shapes indicate different partitioning methods. Again, only instances with more than 10000 points are included. Based on the tests, there is no augmented CHP variant that is clearly better than standard CHP. The tour length comparisons do not show any augmented CHP variant that produces consistently better tours. However, as problems continue to grow larger, standard CHP seems to take slightly longer than augmented CHP in many of instances.

Based on the test results, augmented CHP performs comparably with CHP in many cases. As problems grow increasingly large, augmented CHP may run slightly faster. One potential limitation is the way partition basis points are ordered. Choosing which points to include with the convex hull

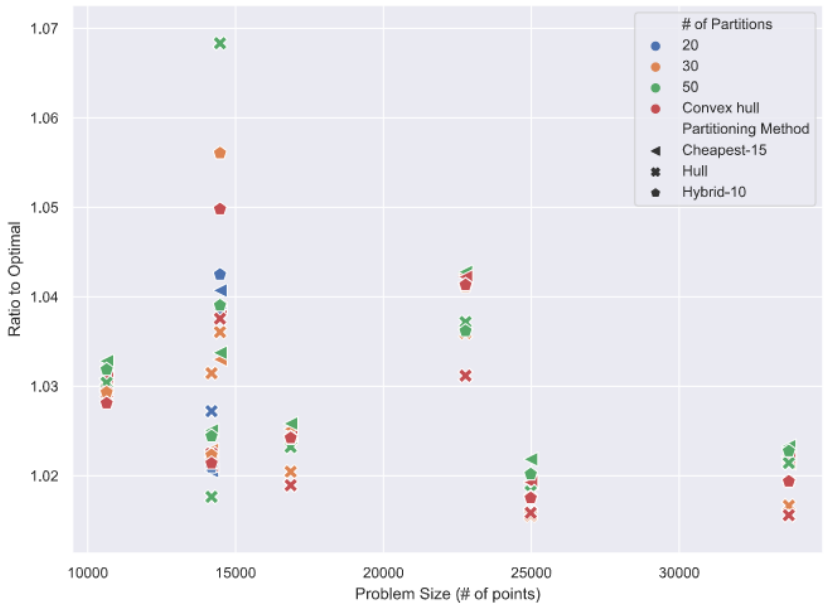


(a) National Instances

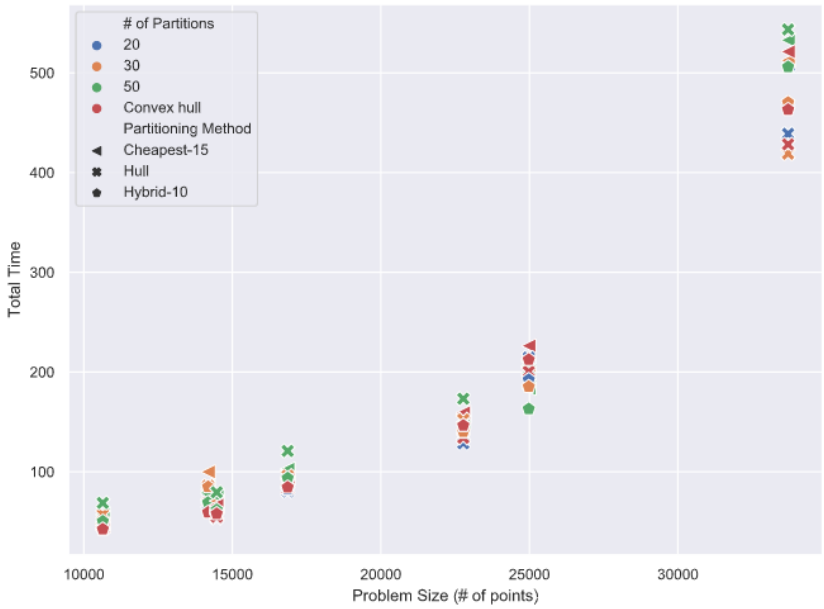


(b) VLSI Instances

Figure 5.3: Summary of Augmented CHP using Ascending Selection

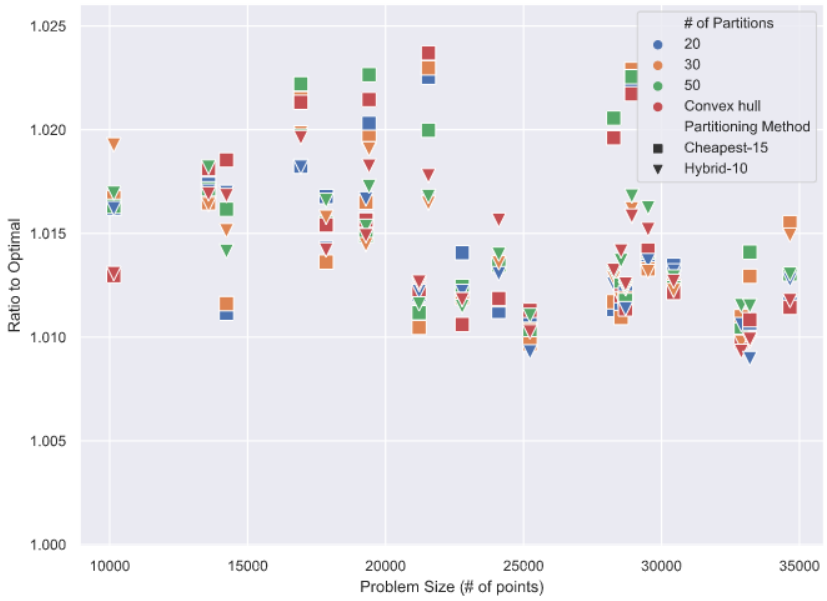


(a) Tour Length Ratio as Problem Size Increases

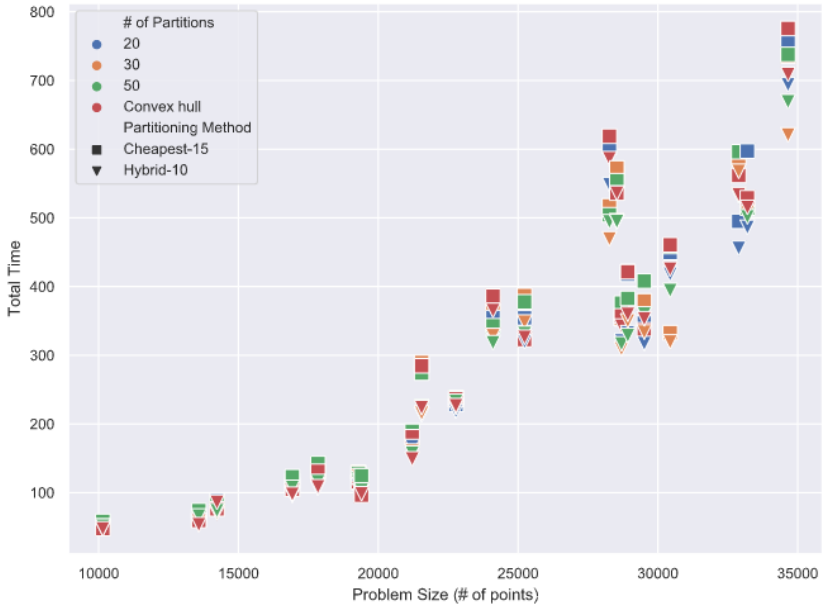


(b) Runtime as Problem Size Increases

Figure 5.4: Augmented CHP Results for National Instances



(a) Tour Length Ratio as Problem Size Increases



(b) Runtime as Problem Size Increases

Figure 5.5: Augmented CHP Results for VLSI Instances

points and finding the Hamiltonian tour through this augmented hull both add time. Additionally, using more partitions typically leads to smaller partitions. This means more time is spent forming partitions and less is spent solving subproblems. As discussed in Section 4.9, augmented CHP would likely improve with a more integrated implementation. Partitioning is performed in the Python portion of the code. Additionally, using more partitions requires more reading and writing of files to use with LKH. With a more integrated version of augmented CHP written in C, partitioning would take less time, and the file reading and writing would not be required. Another limitation with using a Hamiltonian tour of the augmented point set is that the order of points in the smaller tour is not guaranteed to be the same order in which those points occur in the overall tour.

Ideally, augmented CHP would have been an outright improvement on traditional CHP. However, it seems to highlight how effective CHP is as a TSP heuristic and underlines the usefulness of the convex hull of the input points. The major limitation of augmented CHP is not knowing the order of the augmented point set in the final tour. This order is found heuristically, which both adds time and leads to error in the final tour. These initial tests show augmented CHP can produce better tours and run more quickly than CHP. More work into the method could lead to consistently better solutions. One main area of potential work is developing better selection methods, ideally with an inherent order of the points. This would more closely mirror CHP but would ideally contain more points to build partitions around.

5.3 Secondary Improvement Algorithms on CHP Tours

As outlined in Section 2.1.4, TSP improvement methods take an initial tour and make changes, typically edge swaps, that lead to a shorter distance tour. While capable of generating a strong initial tour, LKH is at its most basic level a very strong implementation of the Lin-Kernighan algorithm, allowing it to be used as an improvement method [Helsgaun, 2000]. To that end, any feasible tour, including one generated through CHP, can be used as an initial tour in LKH. Beyond LKH, other improvement methods, such as 2- or 3-opt can be used to improve a tour found through CHP.

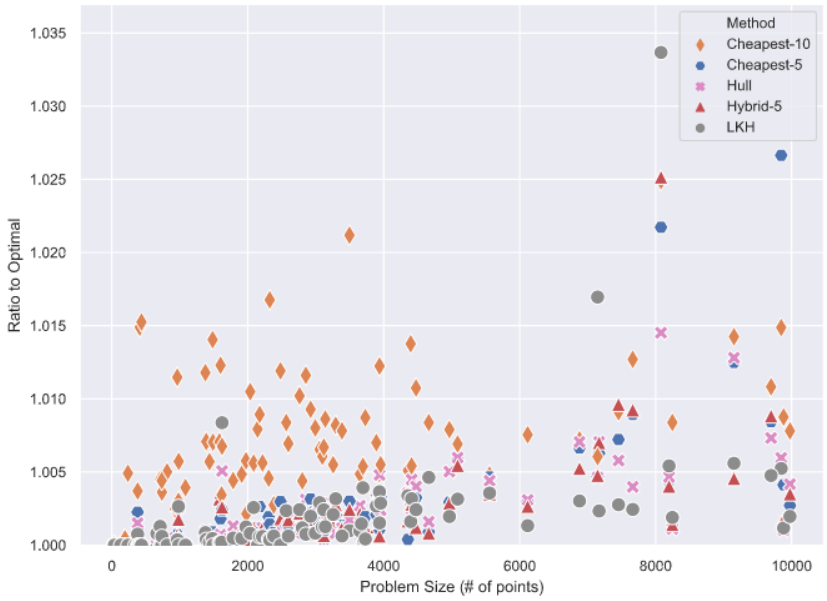
The results of preliminary testing are shown in Figure 5.6. These tests used CHP tours as the initial tour for LKH. Both national and VLSI instances with less than 10000 points were tested. The initial tours came from CHP using cheapest-5, cheapest-10, hull, and hybrid partitioning. A comparison of tour length ratios for the CHP variants and fast LKH is shown in Figure 5.6a. As shown in the chart, using the CHP tour an initial tour for LKH typically leads to worse solutions. Some partitioning methods lead to better tours than others. For example, cheapest-10 clearly performs the worst, as most of its tour length ratios are higher than that of any other method. The other CHP variants find tours much closer to LKH, especially for smaller instances. The good news is that the secondary LKH algorithm did improve the tours in all instances. This shows that improving the CHP tours can be done by using them as input to a secondary algorithm. Unfortunately, using an initial or input tour did not seem to improve the runtime of LKH. Runtimes are shown in Figure 5.6b.

The LKH runtimes are from fast LKH and the CHP variant runtimes are the total runtime including CHP and the secondary LKH algorithm. As shown in the chart, using CHP to initialize LKH leads to consistently longer runtimes than running LKH on its own.

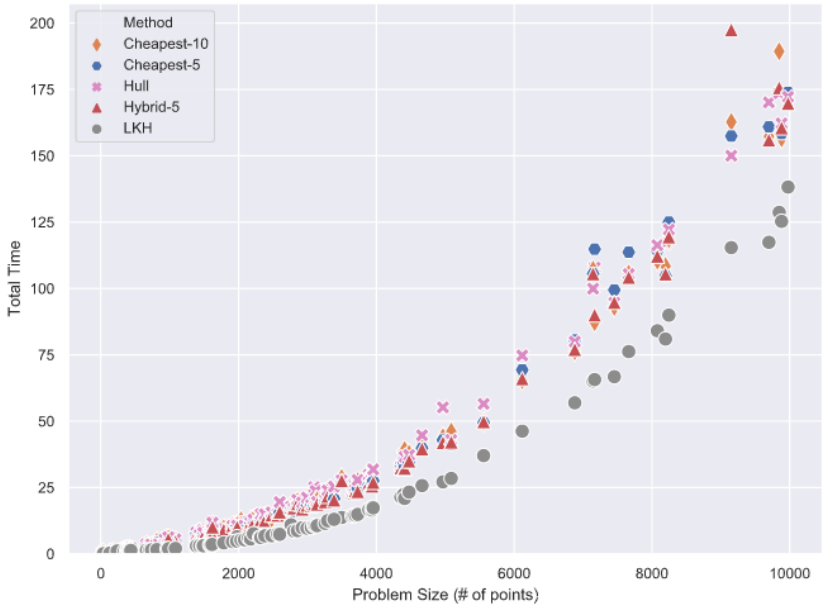
In theory, using a tour found through CHP to give LKH a good starting point, would lead to a decrease in the time spend running LKH. However, testing showed the opposite was true. The reason behind this is unclear. It is possible that the initial CHP tours are worse than those initialized by LKH on its own. This would lead to more work to reach the eventual output tour. Other preliminary tests using $2-opt$ as the improvement algorithm showed little to no decrease in tour length over CHP on its own. Running a secondary improvement algorithm does lead to a decrease in CHP tour length. However, finding the right method and implementation that allows this to be done in a reasonable time remains as a potential area of future work.

5.4 Results Tables

The tables in this section contain the results of testing on the CHP extensions described in Chapter 5. The tables come in two forms, mirroring the formats of those seen in Section 4.10.



(a) Tour Length Ratio as Problem Size Increases



(b) Runtime as Problem Size Increases

Figure 5.6: Results of using CHP tour as input to LKH

Type	Partitioning Method	Recursion Level	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
National	Cheapest-15	5000	1.037837	0.00829	1.031333	1.033169	1.035006	1.041089	1.047171
		10000	1.027882	0.012505	1.019253	1.020711	1.02217	1.032196	1.042223
		20000	1.027881	0.012507	1.01925	1.02071	1.02217	1.032197	1.042225
		No recursion	1.027882	0.012505	1.019253	1.020711	1.02217	1.032196	1.042223
	Hull	5000	1.028533	0.004661	1.024381	1.024995	1.026838	1.032493	1.034382
		10000	1.024711	0.007756	1.015622	1.017995	1.025607	1.030093	1.034383
		20000	1.020884	0.00892	1.0156	1.015734	1.015868	1.023526	1.031183
		No recursion	1.020884	0.008921	1.0156	1.015734	1.015868	1.023527	1.031185
	Hybrid-10	5000	1.03595	0.008446	1.029656	1.031151	1.032645	1.039097	1.045548
		10000	1.026111	0.013224	1.017526	1.018497	1.019468	1.030403	1.041339
		20000	1.026077	0.013249	1.017526	1.018446	1.019366	1.030353	1.041339
		No recursion	1.026082	0.013245	1.017526	1.018454	1.019382	1.030361	1.041339
VLSI	Cheapest-15	5000	1.02645	0.004857	1.018879	1.022416	1.02653	1.030665	1.033898
		10000	1.018009	0.006439	1.010873	1.014039	1.015764	1.020098	1.030875
		20000	1.013776	0.00447	1.009798	1.01128	1.011933	1.01372	1.023705
		No recursion	1.013777	0.004469	1.009798	1.01131	1.011933	1.01372	1.023705
	Hybrid-10	5000	1.027937	0.00561	1.017062	1.024743	1.028066	1.030681	1.038
		10000	1.018666	0.005539	1.012422	1.013832	1.017739	1.02203	1.029773
		20000	1.013029	0.002477	1.009312	1.011491	1.012676	1.014935	1.017782
		No recursion	1.013057	0.002452	1.009312	1.011779	1.012676	1.014935	1.017782

Table 5.1: Recursive CHP Tour Length Ratio Summary Statistics

Type	Partitioning Method	Recursion Level	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
National	Cheapest-15	5000	0.789273	0.116352	0.712276	0.722349	0.732422	0.827771	0.923119
		10000	1.018169	0.006728	1.010838	1.015223	1.019608	1.021835	1.024062
		20000	0.984883	0.037098	0.944507	0.968592	0.992677	1.005071	1.017465
	Hull	5000	0.790014	0.106231	0.68165	0.715605	0.754885	0.868855	0.939309
		10000	0.919014	0.129646	0.751486	0.809046	0.959566	1.015093	1.050991
		20000	1.01571	0.021694	0.992438	1.005878	1.019319	1.027346	1.035374
	Hybrid-10	5000	0.754571	0.111569	0.680153	0.69043	0.700707	0.791779	0.882852
		10000	0.981902	0.011856	0.968544	0.977265	0.985986	0.988581	0.991176
		20000	0.993949	0.033811	0.962107	0.976207	0.990307	1.00987	1.029433
VLSI	Cheapest-15	5000	0.516829	0.091201	0.364979	0.476563	0.518201	0.556509	0.724836
		10000	0.742636	0.215706	0.405574	0.576688	0.705847	0.968023	1.027479
		20000	0.999331	0.030514	0.95209	0.985717	0.998534	1.01199	1.077469
	Hybrid-10	5000	0.445583	0.099213	0.276712	0.383597	0.456512	0.521333	0.625012
		10000	0.673866	0.213868	0.33373	0.528328	0.640869	0.871391	1.00813
		20000	0.968508	0.054169	0.896206	0.917791	0.972445	0.998028	1.06825

Table 5.2: Recursive CHP Runtime Summary

Type	Partitioning Method	# of Partitions	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
National	Cheapest-15	20	1.028595	0.009568	1.01925	1.021354	1.024356	1.035812	1.042223
		30	1.028088	0.008208	1.019353	1.022165	1.02523	1.032609	1.042488
		50	1.029342	0.007475	1.021842	1.02419	1.025829	1.033281	1.04278
		Convex hull	1.028393	0.009092	1.019253	1.021653	1.024324	1.034824	1.042223
	Hull	20	1.025202	0.008622	1.015622	1.017408	1.02722	1.030228	1.038302
		30	1.026568	0.008841	1.015582	1.018561	1.029772	1.033719	1.036066
		50	1.030922	0.017917	1.017647	1.019834	1.023226	1.033784	1.068343
		Convex hull	1.024412	0.008434	1.0156	1.017405	1.022456	1.030227	1.037562
	Hybrid-10	20	1.027719	0.010295	1.017505	1.020168	1.024236	1.034728	1.042498
		30	1.030021	0.013919	1.017624	1.020832	1.024313	1.035241	1.056067
		50	1.028384	0.007289	1.020193	1.023476	1.024433	1.034033	1.039048
		Convex hull	1.02883	0.012183	1.017526	1.020392	1.024236	1.034728	1.04981
VLSI	Cheapest-15	20	1.014479	0.003795	1.010685	1.011335	1.013463	1.01678	1.022532
		30	1.014551	0.004136	1.009678	1.011707	1.012937	1.016506	1.022983
		50	1.015415	0.00402	1.010346	1.012447	1.0141	1.017151	1.022653
		Convex hull	1.015064	0.004412	1.009798	1.011436	1.012964	1.018543	1.023705
	Hybrid-10	20	1.013939	0.002882	1.008968	1.012201	1.013192	1.016667	1.019077
		30	1.014181	0.002981	1.009756	1.012252	1.01358	1.016189	1.019849
		50	1.014493	0.002571	1.011048	1.012448	1.013998	1.01679	1.019697
		Convex hull	1.014122	0.002812	1.009312	1.012563	1.014154	1.015837	1.019622

Table 5.3: Augmented CHP Tour Length Ratio Summary

Type	Partitioning Method	# of Partitions	Mean	Std Dev	Minimum	25%	Median	75%	Maximum
National	Cheapest-15	20	1.009456	0.066708	0.951029	0.976167	0.987935	1.010997	1.152903
		30	1.071674	0.184917	0.878001	0.986499	1.049464	1.06955	1.462157
		50	1.071452	0.161424	0.811153	0.981504	1.093423	1.184093	1.264395
	Hull	20	1.014083	0.047745	0.958761	0.974438	1.017489	1.047509	1.078437
		30	1.123717	0.108899	0.978209	1.040922	1.136655	1.204772	1.259768
		50	1.299513	0.190337	1.00972	1.186225	1.292067	1.45381	1.514735
	Hybrid-10	20	0.986998	0.049783	0.905129	0.963773	0.98327	1.022229	1.048583
		30	1.059142	0.178107	0.872193	0.983299	1.015383	1.064299	1.431221
		50	1.055587	0.141432	0.767197	1.033945	1.092574	1.13753	1.186392
VLSI	Cheapest-15	20	1.010186	0.057287	0.881288	0.977469	0.991438	1.054332	1.128474
		30	1.028189	0.110638	0.722431	0.989959	1.016202	1.115002	1.198483
		50	1.063823	0.126532	0.815147	0.971786	1.051984	1.170091	1.292952
	Hybrid-10	20	0.971819	0.060194	0.85374	0.935801	0.978853	1.011143	1.120309
		30	0.974786	0.106192	0.750662	0.903658	0.986833	1.063612	1.172662
		50	0.997058	0.095607	0.843483	0.924819	1.006524	1.05566	1.236394

Table 5.4: Augmented CHP Runtime Summary

Chapter 6

Partitioning for Constrained Routing Problems

Beyond the TSP, applying partitioning to other NP-hard routing problems may require different or more complicated methods. Extending the CHP framework to solve SOPs is fairly straightforward as it requires a single additional step. However, when considering resource-constrained problems the sequential nature of CHP becomes difficult to use. For this reason, the algorithms described for solving these problems uses an entirely different idea of partitioning.

6.1 Sequential Ordering Problems

Applying CHP to the sequential ordering problem (SOP) uses a similar technique to augmented CHP but takes the idea further. Instead of using the convex hull boundary, partitioning the SOP relies on the precedence constraints defining the problem. Consider two subsets of the SOP input points. Precedence points are those used in a precedence relationship. Interior points are any not included in a precedence constraint. To apply the CHP framework to the SOP, precedence points replace hull points as the basis for partitions.

The general procedure is outlined in Algorithm 16. It differs from standard CHP and its other extensions in that the SOP algorithm never uses the convex hull. It most closely resembles augmented CHP because both require solving a smaller version of the original problem prior to forming partitions. In augmented CHP, this is a smaller TSP, and a smaller SOP must be solved in this case. The smaller SOP finds the order through the precedence points without considering any points not specified in a precedence relationship. In SOPs, the order of the convex hull points is not guaranteed to be the same in an optimal tour. For this reason, another subset of known points with known order is necessary to serve as the basis of partitions. The given precedence information helps form this ordered subset.

Because of the initial, smaller SOP, the effectiveness of this method depends on the number of precedence points. In the case that every point is included in a precedence constraint, then this method will have no effect on the output because the initial SOP is the same as the original problem. At the other extreme, if no points are included in precedence relationship, this method should not be used and instead the problem should be solved as a TSP. More generally, a happy medium likely exists in the number of precedence points for this method to be effective. If too many or too few precedence points are given in the problem, then runtime increases. If the number of precedence points is large relative to the total number of points, then the initial SOP takes more time, but the partitions, and thus subproblems, will be smaller. If the problem includes relatively few precedence points, then the initial SOP

can be run quickly. The downside is large subproblems taking longer to solve.

Input: N: set of points, PC: set of precedence relationships
Output: T: heuristic tour
PP = points used in precedence relationship;
T_{PP} = SOP through precedence points;
Initialize partitions by forming one for each pair of consecutive points in T_{PP};
Assign points to partitions;
Solve Hamiltonian path through each partition;
T = join partition paths using points in T_{PP} ; Return T;

Algorithm 16: SOP Partitioning Framework

It is possible to include the convex hull points when forming partitions. However, any hull points not included in precedence relationships would be additional points included when solving the initial SOP. This may cause a longer runtime for the initial SOP but also leads to fewer, and likely smaller, subproblems.

The current version of the SOP partitioning framework builds on the CHP implementation. For that reason, it currently solves only Euclidean SOP instances. A Euclidean SOP uses edge symmetric distances that satisfy the triangle inequality. Preliminary test results seem promising but making any major performance claims requires more thorough testing. Additionally, because it shares much of the CHP implementation, the current SOP framework requires both point coordinates and precedence relationships to solve the problem. SOP test instances consist of only distance matrices and typically have asymmetric edge costs. For that reason, the current SOP partitioning framework cannot be tested on these instances. New SOP instances, created to test

the framework, combine point coordinate data from symmetric TSPs with the precedence constraint data from existing SOP test problems. Unfortunately, this means optimal solutions to the test problems do not already exist. In summary, the two main areas of future work for the SOP partitioning framework include further testing of new SOP instances and extending the framework to allow for use of the existing test problems.

6.2 Resource-Constrained Problems

When this work began, the idea was to use a similar, geometry-based partitioning method to solve resource-constrained problems, including the RCSPP and RCTSP. However, difficulties in this approach became apparent. For the remainder of this section, the discussion focuses on the RCSPP. However, the same issues, and more, arise for the RCTSP.

First, the RCSPP lacks maybe the most important characteristic allowing for CHP to be effective in solving TSPs. The relationship between a point set's convex hull and the optimal Hamiltonian tour through those points makes the problem of combining subproblem solutions trivial in CHP. Any partitioning method must specify ways of dividing the original problem into subproblems, solving subproblems, and then combining the subproblem solutions to form a solution to the original problem. The known order of the convex hull points immediately handles the issue of combining subproblem solutions. Additionally, it serves as the basis for how to form subproblems, at least in CHP. Unfortunately, there is no immediate analog for resource-constrained

problems. One idea to address this issue was to find bottleneck nodes within the network. The premise was to find subnetworks with limited incoming and outgoing edges and form subproblems based on those. Unfortunately, every subnetwork could only have a single incoming edge and a single outgoing edge for this idea to use the methodologies developed for CHP. In early testing, subnetworks with only a one entry and one exit did not occur with any regularity or on the scale required within a given network. Instead, subnetworks had many entry or exit nodes, which led to the second major roadblock in applying a CHP-like method. That is, any node with an incoming edge can be the origin of the subproblem. Similarly, any node with an outgoing edge can be the destination.

Hamiltonian path subproblems in CHP had an obvious origin and destination, again thanks to the relationship between the convex hull and optimal tour. When subproblems have several potential origins and destinations, how should they be chosen? Again, any method should follow the general procedure of forming subproblems, solving subproblems, and then combining their solutions. To accommodate the case of subproblems with multiple origins and destinations, subproblem solutions must consist of the path between each combination of origin and destination. When solving a subproblem, it is not known which incoming and outgoing edges from a subnetwork a final solution includes. For the RCSPP, the outgoing edges are less of an issue, as good all-destinations methods exist. On the other hand, an all-destinations method needs to be run for each possible origin, making it a time intensive approach.

Methods using a false origin were developed in an attempt to speed up solution time. Unfortunately, these methods faced issues because paths preceding each incoming edge can have different distances and resource costs. These preceding values must be somehow incorporated. Otherwise, domination criteria eliminate subpaths that are non-dominated when considering a full path through the network. In the end, any attempted technique addressing this issue led to the same runtime issues.

Finally, using a geometric partitioning method similar to CHP requires dividing the global resource constraint across the points subsets. In theory, the subpath through any subnetwork could consume any amount of the constrained resource between 0 and β and still be included in a final solution. In practice, preprocessing can determine the maximum resource cost value relevant to any subproblem, limiting the work required for each. However, a larger issue exists when it comes to combining subproblem solutions and forming a final solution. To form a correct final solution, non-dominated paths from every origin to every destination for every subproblem must be known. Subproblem solutions do not inherently take into account the distances and resource costs of paths leading into them. For example, the shortest path through a subnetwork may lead to infeasibility when used as part of a path through the entire network. Therefore, any combination of non-dominated paths from subnetworks must be checked when forming a solution for the original problem.

A somewhat obvious method solves subproblems as a form of preprocessing. Then a secondary algorithm solves the original problem on the pre-

processed network. This idea led to two methods being developed. First, a network reduction procedure formed a reduced network composed of only nodes and edges used in non-dominated subpaths. An RCSPP was then solved on this reduced network. The second approach attempted to reduce the size of the problem by forming a secondary network consisting of only subproblem origins and destinations. Then, non-dominated paths from each subproblem were added as edges between their origin and destination nodes. In the end, this led to fewer nodes but a larger number of edges in the secondary network. More work could probably be done to reduce the number of edges, specifically when it comes to redundancy. Given the time required to solve subproblems, make a new network, and solve the reduced problem, neither method showed significant runtime improvement.

In the end, the issues discussed above led to a different approach to partitioning being applied to the RCSPP.

6.2.1 Resource Partitioned Dynamic Programming

In lieu of forming groups based on the inherent geometry of the input network, an alternative means of partitioning was used for the RCSPP. Like CHP, the partitioning method described in this section was developed for use in solving a specific problem. In this case, that problem is the ADRCSPP. It can therefore be immediately applied to the single-destination RCSPP. Additionally, a slight modification allows it to be used to find all non-dominated paths to all destinations.

As the name implies, resource partitioned DAD (RPDAD) is built around the DAD dynamic programming algorithm presented in Section 2.2.2. An outline of RPDAD is shown in Algorithm 17. The algorithm first forms sets of nodes based on resource cost partitioning, a procedure which is described in the next section. Then, RPDAD follows the same process as standard DAD nearly exactly. The only difference is the set of nodes iterated through at each cost value. Instead of going through every node for every cost, only nodes that could have a non-dominated path with the current resource cost value are searched. This is a seemingly small difference. However, limiting the search space at each iteration can greatly improve the total runtime.

Input: N: set of nodes, E: set of edges, β: upper limit on resource consumption
Output: L: map of non-dominated path lengths,P : map previous nodes
C,RP = overall cost range and partitions from resource cost partitioning $L(s,0) = 0, P(s,0) = -1$;
$L(v,0) = \infty, P(n,0) = -1, \forall n \in N, n \neq s$;
$t = 0$;
for $t \in C$ **do**
  for $n \in RP[t]$ **do**
    $L(n,t) =$
    $\min\{L(n,t-1), \min_{u:(u,n)\in E \text{ and } c_{un}<=t}\{L(u,t-c_{un})+d_{un}\}\}$;
    Track $P(n,t)$;
  end
end

Algorithm 17: Resource Partitioned DAD

Before diving into more specifics, there are some key differences with previous methods presented in this dissertation that should be discussed. First

and foremost, the way the input points are partitioned is entirely different. In CHP, points are grouped based on their proximity to other points. Therefore, partitions are likely to be continuous regions of points. On the other hand, RPDAD does not use proximity at all when grouping points. For this reason, partitions in RPDAD may contain nodes spread throughout the network. Another major difference is that partitions in CHP are largely disjoint. Hull points are contained in exactly two groups and interior points are in exactly one. Typically, nodes will have a range of non-dominated resource costs, meaning nodes are contained in many partitions. Maybe the biggest difference is the structure of the algorithms themselves and how their subproblems may be solved in parallel. While it has not been implemented, the Hamiltonian path subproblems in CHP could be solved in parallel. On the other hand, the iterations of RPDAD must be run sequentially, as the paths found in one iteration depend on those from the previous iteration. In theory, the search through nodes at a given resource cost could be done in parallel because their results do not depend on one another. Parallelization is not the focus of this dissertation and will be left for future work.

6.2.1.1 Resource Cost Partitioning

Resource cost partitioning (RCP) forms a group of nodes for each resource cost value relevant to the problem. The general steps in RCP consist of finding the relevant cost ranges for each node, finding the cost range for the problem overall, and then forming groups of points for each cost level. A more

detailed outline of the steps of RCP is shown in Algorithm 18.

Input: N: set of nodes, E: set of edges, s: origin of paths, β: resource consumption upper bound
Output: C: set of relevant costs, Q: set of partitions
$MinCosts = Dijkstra(s, cost)$;
$MinDistPaths = DijkstraPath(s, dist)$;
for $i \in N$ **do**
  $MaxCosts[i] = cost(MinDistPaths[i])$;
end
$LB = min\{MinCosts\}$;
$UB = max\{MaxCosts\}$;
$C =$ range from LB to UB;
for $c \in C$ **do**
  Make empty partition $q_c \in Q$;
  for $i \in N$ **do**
    if $MinCosts[i] \leq C \leq MaxCosts[i]$ **then**
      Add i to q_c;
    end
  end
end
Return C,Q;

Algorithm 18: Resource cost partitioning

Finding the relevant costs for each node first requires a discussion of what relevant means in this context. Consider resource cost c and node i. In this dissertation, c is relevant for i if it is possible that a non-dominated path ending in i has cost c. In other words, c falls between the minimum feasible cost to i and the cost of the minimum distance path to i. The minimum feasible cost is the lower bound on the range of relevant costs for i. The resource cost of the minimum distance path is the upper bound because no path with higher cost can be non-dominated. To be non-dominated, a higher

cost path would need to have a distance less than the minimum distance path, which is impossible. Fortunately, finding the relevant cost range is simple for the RCSPP and ADRCSPP. Two runs of Dijkstra's algorithm can find the range for every point. One run uses distance as the edge weight being minimized and the other uses resource cost. This finds the minimum distance and minimum cost paths for each point. The lower bounds are given directly from the minimum cost paths. The upper bounds require one additional step. To find the maximum relevant costs for a node, use the total sum of resource costs of edges in its minimum distance path.

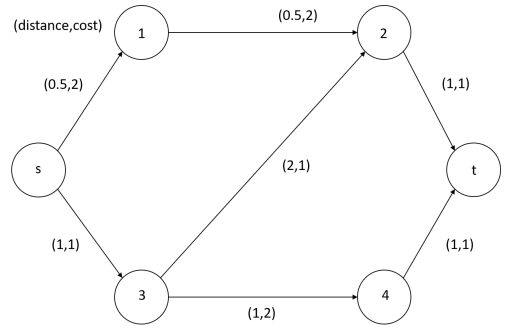
Next, the relevant resource consumption range for the entire problem must be found. The lower bound on this range can be trivially set to 0. For a slightly smaller range, set the global minimum of each node's minimum feasible cost as the lower bound. Depending on the problem being solved, the upper bound on this range of costs differs. Single-destination RCSPPs are not explicitly interested in all non-dominated paths. Therefore, the upper bound is the global resource constraint, β. If the cost of the minimum distance path is less than β, then that path is also resource-feasible, and no more work need be done. To find all non-dominated paths less than β, the upper bound is the minimum of the resource cost of the minimum distance path and β. To find all non-dominated paths regardless of β, then the upper bound is the cost of the minimum distance path. To find all non-dominated paths to more than one destination, the upper bound is the maximum of the costs of the minimum distance path to any desired destination. This can be expanded to

the all-destinations case as well.

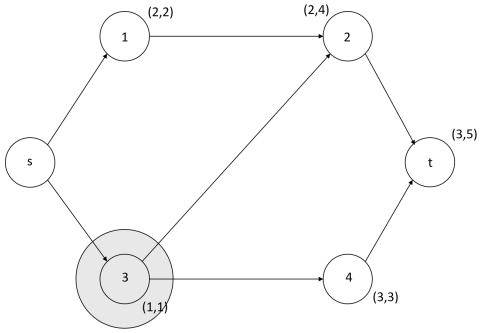
Finally, forming partitions requires matching the overall cost range with the nodes' relevant cost ranges. Consider a relevant resource consumption value c and node i. Let c_i^+ and c_i^- be the maximum and minimum relevant costs i. Node i belongs in cost partition q_c if $c_i^- \leq c \leq c_i^+$, where q_c is the group of nodes to be searched for cost c.

6.2.1.2 Illustrative Example

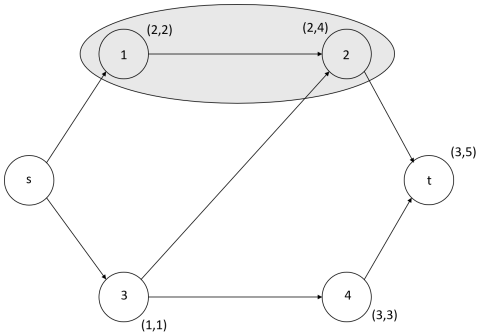
An example of RCP is shown in Figure 6.1. The example network is the same as previously seen in Figure 2.3. The first step of RCP is finding the minimum and maximum relevant costs for each node in the network. These resource cost ranges are shown in parentheses by each node starting in Figure 6.1b. The subsequent figures show the group of points needing to be searched at each incremental resource cost. Nodes are only included in groups if the current resource cost falls within their relevant range. To see this, consider node 2. Its relevant cost range is $(2, 4)$ and thus it is only included the searches for paths with resource cost 2, 3, or 4. As a reminder, these groups are meant to limit the number of nodes searched in an iteration of a dynamic programming algorithm. Clearly, iterating through only the corresponding group of points for each cost level requires less work than searching each node at each resource cost value.



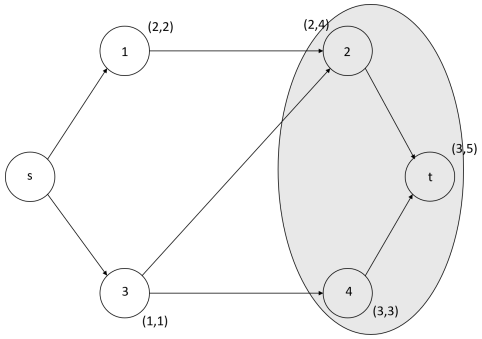
(a) Network



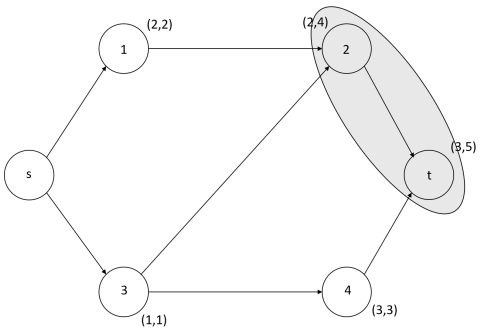
(b) Cost 1 Partition



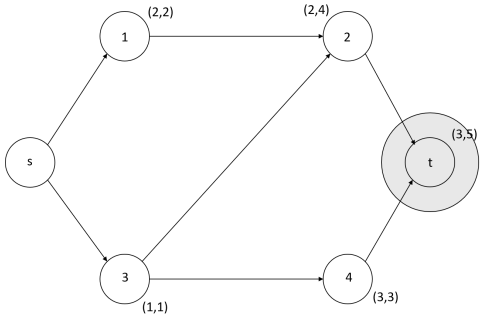
(c) Cost 2 Partition



(d) Cost 3 Partition



(e) Cost 4 Partition



(f) Cost 5 Partition

Figure 6.1: Illustrative Example of CHP

6.2.1.3 Resource Partitioning Experiments

To test the effectiveness of RPDAD, experiments compare it to two other dynamic programming algorithms, DSA and LSA. Test problems consist of transportation network test problems (TNTP) based on city street grids [Stabler,]. The tested networks ranged in size from 24 to 13741 nodes. These networks are typically used to test transportation system problems, such as the traffic assignment problem (TAP). Edges in these networks have associated lengths, which are used to compute the distance of paths through the network. To use these networks to test RPDAD and other RCSPP methods, resource costs must be associated with each edge. Some networks contain edges with toll values, which could be used as a resource cost. However, these are not consistently used across the networks; even when networks contain toll values, they a typically used on only a few edges. For these reasons, random edge costs were added to edges. For these tests, edge resource costs were drawn from a random uniform distribution between input lower and upper bounds. Method comparisons were made for several cost ranges. Additionally, edge costs are integer in these tests. Similar to DAD, RPDAD finds optimal solutions with integer edge costs. The effects of non-integer edge costs and rounding is also an area of interest but left for future work at this points.

One focus of testing is how the runtime of different algorithms scaled as networks increased in size and edge costs became larger. To that end, tests were run using edge cost ranges of $(0, 10,\ (0, 100),\ (0, 1000)$, and $(0, 10000)$. In these tests, an edge cost range of $(0, 10)$ means that each edge has an

integer resource cost between 0 and 10, inclusive. Because there are not that many TNTP networks and costs are random, several tests were run for each network, each with new edge costs. In each test, all non-dominated paths from an origin to all other nodes were found. The results of these tests are shown in Figures 6.2 and 6.3. In these charts, different colors represent different solution methods and markers differentiate the cost range.

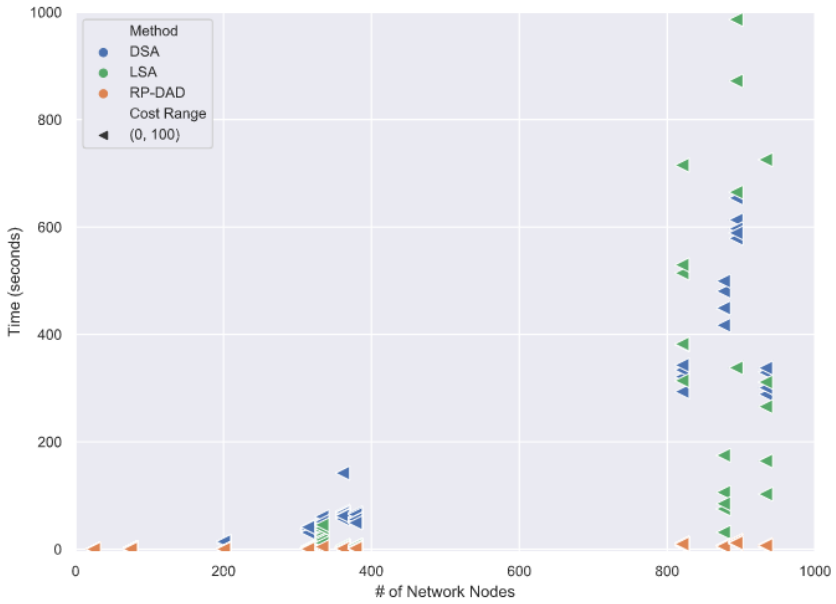
Figure 6.2 shows runtimes for networks with edge costs between 0 and 100. Figures 6.2a and 6.2b show a comparison of RPDAD, LSA, and DSA runtimes when solving networks up to 1000, and 400 nodes, respectively. Clearly, RPDAD solves these networks much faster than the other two methods. Comparing the two charts allows examination of how the algorithms' runtimes scale as problems grow in size. RPDAD's runtime does not grow nearly as quickly as other methods. Additionally, the runtimes experienced by multiple tests of the same network are much more consistent for RPDAD.

Figure 6.3 shows results for a larger cost range, with edge costs ranging from 0 and 1000. Given its performance at the lower cost range, similarity to RPDAD, and expected runtime scaling, DSA was not included in these tests. Because both are incremental cost algorithms, anything increasing the runtime of RPDAD would lead to an even larger increase in DSA. Similar to the top two charts, Figures 6.3a and 6.3b show results for networks with less than 1000 and 400 nodes, respectively. In this case, RPDAD does not perform as well for smaller networks when compared to LSA. However, when looking at larger networks, RPDAD's runtime is again more consistent and grows more slowly

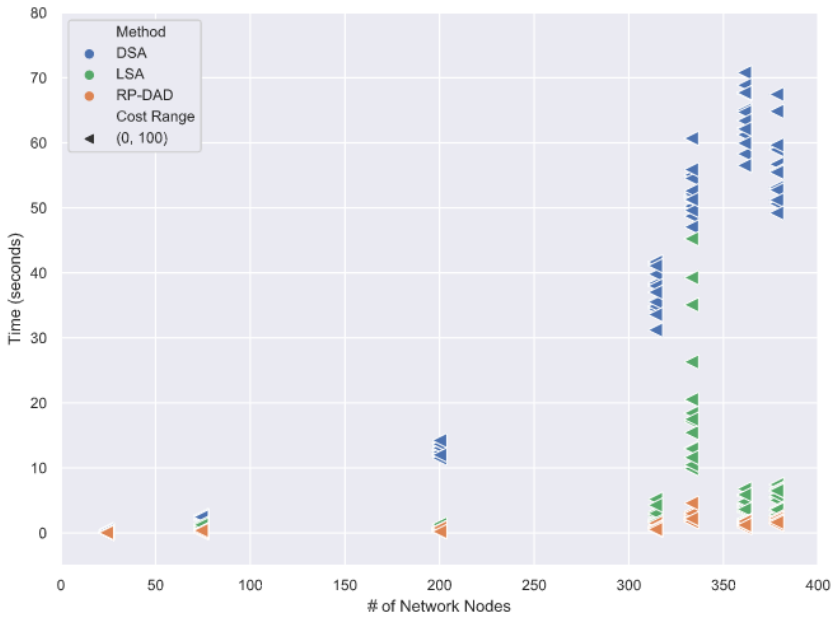
with network size.

Another interesting comparison is seen by looking at Figures 6.2a and 6.3a. Looking at the RPDAD runtimes for the higher cost range in Figure 6.3a, they are on par with the fastest LSA runtimes seen in Figure 6.2a. On the other hand, the LSA runtimes increase much more between the two cases. This again emphasizes how well RPDAD scales to larger networks, especially relative to LSA.

These trends make sense given how each algorithm works. Both RPDAD and DSA are incremental cost algorithms. That is, they use solutions at incremental cost values to build solutions. LSA, on the other hand, extends paths by checking each successor node. The expected runtimes of these two methodologies are impacted by different network characteristics. RPDAD and DSA depend much more on the overall maximum cost range of the network. This range is the difference between the minimum cost of the minimum cost paths to any node and the maximum cost of the minimum distance paths to any node. This overall range determines how many resource cost values DSA or RPDAD must search in order to find all solutions. Increasing the range of values each edge can take increases the size of the overall range, leading to the longer runtimes. LSA's runtime is much more determined by the number of a network's nodes and edges. This is because LSA does not search incremental cost values but by extending labels one node at a time. As the number of nodes increases, the number of labels created and checked for domination grows very quickly. This is what leads to the steep increase in runtime as networks grow

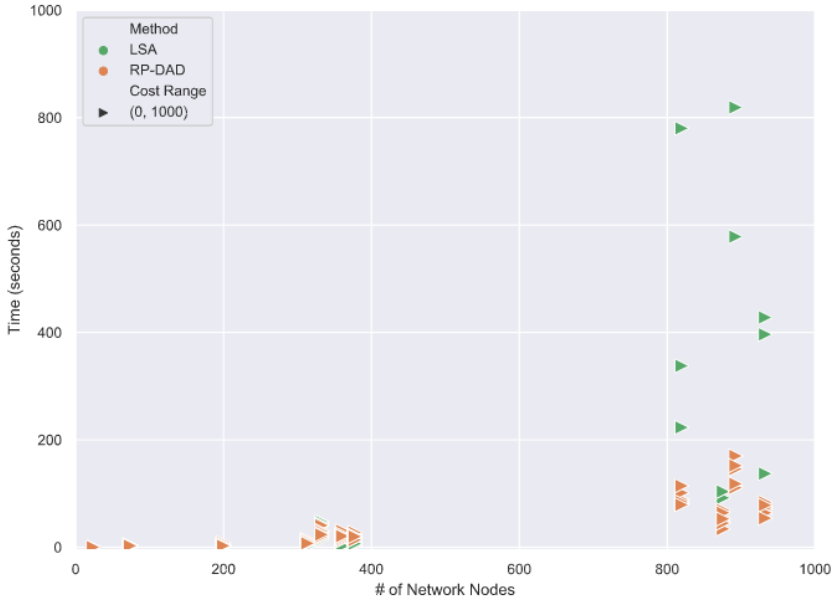


(a) Networks up to 1000 Nodes

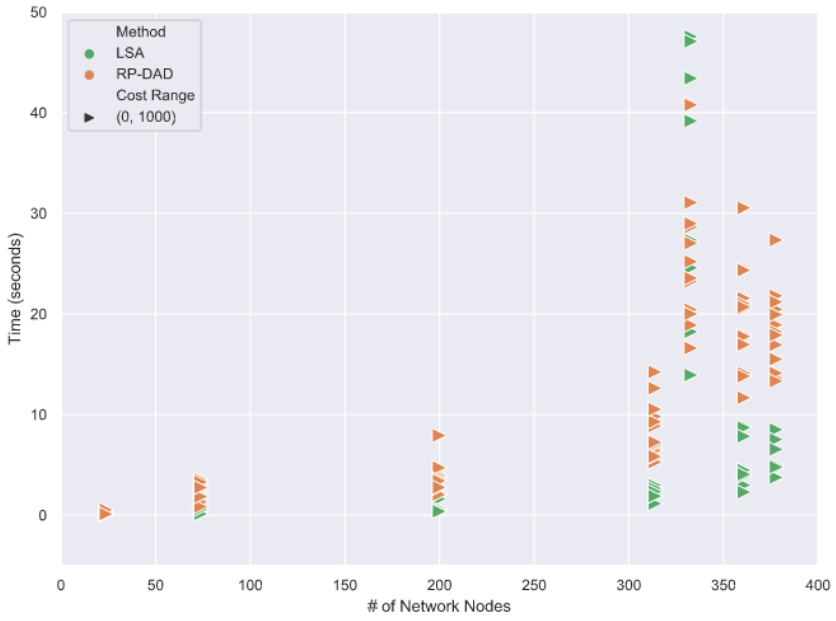


(b) Networks up to 400 Nodes

Figure 6.2: Runtime Comparison using Edge Cost Range $(0, 100)$



(a) Networks up to 1000 nodes



(b) Networks up to 400 nodes

Figure 6.3: Runtime Comparison using Edge Cost Range $(0, 1000)$

in size. Again looking at Figures 6.3a and 6.3b, the tradeoff between LSA and RPDAD can be seen. Clearly, with the higher cost range, LSA typically runs faster for smaller networks and RPDAD remains better for larger networks.

To see how RPDAD's runtime scales, more edge cost ranges were tested. The runtime of RPDAD tests using edge cost ranges of $(0, 10)$, $(0, 100)$, $(0, 1000)$, and $(0, 10000)$ are shown in Figure 6.4. As expected, a larger range of edge costs leads to a longer runtime. Given the time taken to solve each instance, cost ranges of $(0, 1000)$ and $(0, 10000)$ were not tested on instances larger than 4000 nodes. In Figure 6.4 edge resource cost ranges are differentiated by both color and marker shape, as these results only compare methods from RPDAD.

Based on testing, RPDAD provides a good alternative to LSA and improves on other incremental cost algorithms, such as DSA. Specifically, for large networks, RPDAD seems to find non-dominated paths to all destinations in a network more quickly than LSA or DSA. When networks are smaller and the overall cost range is larger, LSA may run more quickly. RPDAD improves on the runtime of DSA in either case, probably because both are incremental cost algorithms. Therefore, any factor leading to an increased runtime for RPDAD would also lead to an increase in that of DSA.

6.2.2 Limitations and Future Work

The biggest limitation to RPDAD carries over from the DAD algorithm. Given the structure of both algorithms, they only work with integer valued edge resource costs and global resource constraint. Any non-integer values

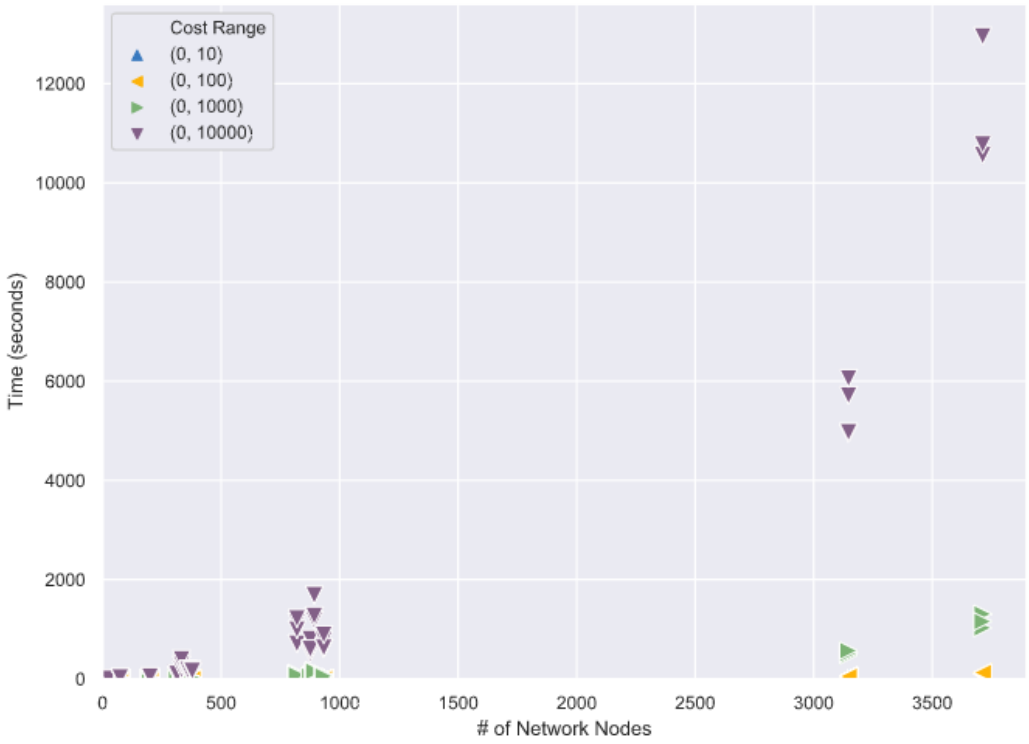


Figure 6.4: RPDAD runtime as edge cost range increases

must be somehow made integer before solving the problem. As discussed in Section 2.2.2, rounding is one way to address this issue. However, rounded edge costs may not lead to an optimal solution. Integrating these rounding methods into RPDAD and testing the effects remain as areas of future work. Another option for handling non-integer edge costs lies in scaling all values to become integer. To better understand this idea, consider a toll road that costs \$1.80. To make this integer, it must be multiplied by 100, giving the toll road a cost of 180. Therefore, to use a network where some edges have costs in dollars and cents, all edge costs and the global constraint must be multiplied by 100. Because tolls on a single edge very rarely exceed \$100, scaling in this way puts edge costs in the range [0,1000]. Testing shows RPDAD performs competitively for this cost range. Some combination of scaling and rounding could also lead to an effective version of RPDAD. For example, rounding any non-integer value to a single decimal point and then scaling by a factor of 10 leads gives an integer value. While this may not lead to optimal solutions, a combination of the two methods may balance runtime and solution accuracy.

In its current form, RPDAD finds non-dominated paths from a single origin to all other nodes in a network. Clearly, this covers the single-destination case as well. However, because it solves for every paths to every other node, RPDAD likely runs more slowly than many dedicated single-destination methods, including bidirectional LSA. Applying preprocessing methods to limit network size or overall cost range could increase the speed of RPDAD. A preprocessing method currently being developed reduces an input network to in-

clude only edges in non-dominated paths between an origin and a destination. Running RPDAD on the reduced network then finds the exact non-dominated paths. An additional benefit of combining preprocessing with RPDAD is the ability to find non-dominated paths to some, but not all, other nodes. Running the previously mentioned preprocessing method for a subset of nodes leads to a reduced network containing edges in a non-dominated path between the origin and any of the destinations. In theory, methods similar to bidirectional methods could be expanded to this case as well, but it may not be as straightforward.

Another major area of potential future work lies in applying resource cost partitioning to the RCTSP. The Hamiltonian requirement stands as the major roadblock to this application. Because every node must be visited in a Hamiltonian tour, the same domination criteria do not apply. An additional limitation lies in finding the minimum and maximum resource cost for every point. Given the structure of dynamic programming algorithms for the TSP, the minimum and maximum resource cost of paths ending in every point through every subset of other points would need to be identified to apply resource cost partitioning. Clearly, this is intractable. The minimum and maximum resource cost tours can be found by running two instances of the unconstrained TSP. This mirrors the two runs of Dijkstra's algorithm in RPDAD. However, because solving the TSP often requires a significant amount of time, even finding minimum and maximum resource cost tours may be unrealistic when solving the RCTSP. Given these limitations, applying partitioning to the

RCTSP requires much more work. If a partitioning method similar to RPDAD can be developed for the RCTSP, it would naturally extend to the orienteering problem. In theory, the output would consist of all non-dominated Hamiltonian paths of any length to each point. This also includes all non-dominated Hamiltonian tours of any length. This then solves the orienteering problem by giving the tour visiting the most points within the distance constraint.

Chapter 7

Conclusion

Partitioning frameworks allow NP-hard routing problems to be solved more quickly. These methods work by dividing the original problem into groups, solving an easier subproblem within each group, and forming an overall solution by combining the subproblem solutions. Partitioning can be especially effective at reducing runtime for routing problems because the computational effort grows exponentially as problem size increases. Therefore, solving several smaller problems may require less time than solving a single large instance. For large problems, the time savings offset the additional time spent forming subproblems and combining their solutions. This idea led to the partitioning methods for the TSP, SOP, and RCSPP described in this dissertation. Developing the convex hull partitioning framework for the TSP is the biggest contribution of this dissertation. Most of the additional work attempts to improve the framework or extend the idea to other problems. The most promising of these extensions include a recursive CHP implementation and a modification for use in solving SOPs. Some characteristics of resource-constrained problems make applying a CHP-like framework difficult. For that reason, an algorithm for the RCSPP uses an entirely different partitioning paradigm to form groups of nodes based on resource cost. Computational experiments demonstrate the

effectiveness of CHP, its extensions, and resource cost partitioning. These results support the claim that partitioning frameworks can be effective at solving NP-hard routing problems.

Convex hull partitioning uses the relationship between the convex hull boundary of a TSP and its optimal tour. Hull points occur in the same order in both the convex hull boundary and the optimal tour. This order provides two benefits. First, it allows subproblems to be defined with a known order. Second, it makes the problem of combining subproblem solutions trivially easy. Each pair of consecutive hull points anchors a partition. Partitioning methods assign each interior point to a single group. Subproblems consist of Hamiltonian path problems through each partition. The hull points serve as the origins and destinations for these paths. Joining partition Hamiltonian paths at their shared hull points produces a solution to the original TSP. In summary, CHP finds portions of the final tour between each pair of consecutive hull points. The work included the development of several new partitioning methods. Some of these extend ideas from existing TSP heuristics while other are entirely new.

Because combining subproblems solutions is made easy, CHP consists of two main problems, dividing the input points and solving subproblems. Given the variety of partitioning methods and subproblem solvers, many variations of CHP exist. Computational experiments attempted to demonstrate the effectiveness of CHP overall and identify variants that perform well across many instances. Experiments consisted of several types of testing and focused

mostly on the partitioning phase. One of the predominant TSP heuristics, LKH, solves subproblems and serves as a benchmark method. Experiments use the optimal solutions of the TSP test instances as another benchmark. The results show the capabilities of CHP and demonstrate the framework's ability to solve TSP effectively. CHP's performance relative to the two benchmarks improves as problems grow in size. On larger instances, CHP runs significantly faster than LKH without a large decrease in tour quality. Different partitioning methods work better on instances from the two different test sets. Hull partitioning appears to work best on TSPs with an irregular outline and variable point density, such as problems from the national problem set. For national TSPs with more than 10000 points, CHP with hull partitioning found tours within 3% of optimal for most instances and required less than 30% of the time used by LKH. In general, cheapest-m and hybrid-m seem to work better on VLSI TSPs, which have a rectangular outline and consistent point density throughout. Specifically, CHP with cheapest-15, hybrid-10, and hybrid-15 all solve VLSI instances with more than 20000 points in around half the time of LKH. Additionally, each CHP variant solves most large VLSI instances well within 1.5% of optimal. These results show a few trends. CHP seems to be more effective on TSPs like those from the VLSI test set. As problems grow in size CHP's performance relative to LKH improves. The framework solves large TSPs much more quickly than LKH and still finds tours with near optimal length. The biggest opportunity for further improvement lies in a faster implementation or one more integrated with LKH.

Some of the extensions of CHP include a recursive version, a method to increase the number of partitions, and an application to solve SOPs. Recursive CHP sets an upper limit on the size of a subproblem solved by LKH. Based on some tests, recursive CHP runs more quickly than CHP and therefore exhibits potential for an even bigger runtime improvement over LKH. The decrease in solution time does lead to slightly longer tours. In general, smaller recursion limits lead to faster solutions and longer tours. Again, recursive CHP offers the biggest benefit when solving large instances and seems to perform better on VLSI instances. For example, 5000 points was the smallest recursion limit tested and typically led to tours with length within 3% of optimal for large VLSI instances. However, it required only around half the time spent by CHP for most instances. This corresponds to around a quarter of LKH time. Another extension, augmented CHP, attempts to increase the number of partitions by using additional points as partition anchors. Augmented CHP led to better tours or faster runtime for some instances, but there were no conclusive benefits across all instances. However, the idea behind augmented CHP led to the application of the CHP framework for solving the SOP. Instead of using the convex hull boundary, points used in precedence constraints serve as the partition bases. This requires the additional step of solving a smaller SOP consisting of only precedence points. While major testing this method is left as future work, some preliminary experiments show promise.

Finally, applying partitioning methods to resource-constrained problems led to resource cost partitioning which groups nodes based on the resource

costs required to reach of their incoming paths. When used in conjunction with existing dynamic programming algorithms, RCP leads to a significant decrease in the search at each iteration. The resulting algorithm RPDAD finds all non-dominated paths from a single origin to all other nodes. RPDAD finds optimal solutions with integer edge resource costs. Experiments compare the runtime of RPDAD to two other dynamic programming algorithms solving for the same information. In general, RPDAD runs more quickly than DSA because the same factors lead to runtime increases in both algorithms. For small networks with a large range of edge resource costs, LSA may run more quickly than RPDAD. However, as networks grow larger, RPDAD outperforms LSA. Therefore, RPDAD provides a good alternative to LSA, and the choice of method depends on the specific instance being solved. Additional testing could examine the case of non-integer resource costs, which would require rounding to be solved with RPDAD. Some other areas of potential future work include combining the benefits of resource cost partitioning and LSA. Finally, the biggest potential for the method lies in extending it to other resource-constrained problems such as the resource-constrained TSP or orienteering problem.

The partitioning frameworks described in this work offer a strong alternative to existing methods. They exhibit the overall potential of partitioning-based methods to effectively solve NP-hard routing problems. Given the increasing dependence on fast and efficient movement in our everyday lives, routing problems are likely to become even more relevant. As they continue

to grow in size and complexity, partitioning frameworks provide the ability to solve these problems quickly while maintaining a high degree of solution accuracy.

Appendices

Appendix A

Additional Background

A.1 TSP

A.1.1 Construction Heuristic Performance

Method	Worst-Case $\frac{\lvert T\rvert}{\lvert T^*\rvert}$	Time Complexity
Greedy (Nearest neighbor)	$\frac{1}{2}\lceil\log(n)\rceil + \frac{1}{2}$	$\Theta(n^2)$
Nearest Insertion	2	$\mathcal{O}(n^2)$
Cheapest Insertion	2	$\mathcal{O}(n^2)$
Arbitrary Insertion	$2ln(n) + 0.16$	$\mathcal{O}(n^2)$
Farthest Insertion	$2ln(n) + 0.16$	$\mathcal{O}(n^2)$
Christofides Algorithm	$\frac{3}{2}$	$\mathcal{O}(n^3)$

Table A.1: Performance and Time Complexity of Construction Heuristics [Golden et al., 1980]

A.1.2 Utilizing Existing Heuristic Bounds

The Held-Karp (HK) lower bound is the solution to the LP relaxation of the symmetric TSP [Held and Karp, 1970],[Held and Karp, 1971]. Let $\overline{T_{HK}}$ and $\overline{T^*}$ be the HK bound value and optimal tour length, respectively. Then, the bound is proven to be no less than $\frac{2}{3}$ of the optimal integer solution, $\frac{\overline{T_{HK}}}{\overline{T^*}} \geq \frac{2}{3}$ [Wolsey, 1980]. Additionally, a widely-accepted conjecture states $\frac{\overline{T_{HK}}}{\overline{T^*}} \geq \frac{3}{4}$ [Carr and Vempala, 2004]. In practice, the HK bound often lies within 1% of the exact cost [Valenzuela and Jones, 1997]. Because it is deterministic

and has a known worst-case, comparing to the HK bound gives an idea of the relative performance of different heuristic methods [Johnson et al., 1996]. The HK bound is especially useful when solving the problem to optimality is impractical. An iterative method using minimum spanning trees solves for the bound in $\mathcal{O}(n\ \log(n))$, allowing for its use without a significant computational cost [Johnson et al., 1996], [Valenzuela and Jones, 1997].

The Christofides algorithm also provides a useful and efficient bound for heuristic comparison. Unlike the HK bound, a solution from the Christofides algorithm gives a well-defined upper bound on the optimal value [Christofides, 1976]. Let $\overline{T_c}$ be the solution value of the Christofides algorithm. Then in the worst case, $\frac{\overline{T_c}}{\overline{T^*}} \leq \frac{3}{2}$.

Consider Lemma A.1.1, which uses the Christofides and HK bounds to place bounds on the ratio of any heuristic tour length to the optimal tour length.

Lemma A.1.1. *Let $\overline{T_{HK}}, \overline{T_c}, z,$ and $\overline{T^*}$ be the HK bound, Christofides bound, heuristic value, and optimal value of a symmetric Euclidean TSP instance. Then, $\frac{2}{3}\frac{z}{\overline{T_{HK}}} \leq \frac{z}{\overline{T^*}} \leq \frac{3}{2}\frac{z}{\overline{T_c}}$.*

Proof. The bounds found through the Held-Karp and Christofides algorithms give an upper and lower bound on the optimal solution, $\frac{3}{2}\overline{T_{HK}} \geq \overline{T^*} \geq \frac{2}{3}\overline{T_c}$. Then, inverting these ratios and multiplying by the heuristic tour length gives bounds on the ratio of heuristic length to the optimal length, $\frac{2}{3}\frac{z}{\overline{T_{HK}}} \leq \frac{z}{\overline{T^*}} \leq \frac{3}{2}\frac{z}{\overline{T_c}}$. □

Lemma A.1.1 combines the bounds on $\overline{T^*}$ from the HK and Christofides algorithms to provide a performance range on the ratio of a heuristic solution to the optimal solution. The effectiveness of this range depends on the heuristic used to produce z and its goal. For example, if z is a feasible solution, the trivial lower bound on $\frac{z}{\overline{T^*}}$ is 1. Thus, the lower bound in Lemma A.1.1 is only relevant if z is at least $\frac{3}{2}\overline{T_{HK}}$. On the other hand, if z is infeasible and a lower bound on $\overline{T^*}$, then Lemma A.1.1 provides a range on how close to $\overline{T^*}$ the lower bound can be. While the performance range may be useful in some applications, the main purpose of the lemma here is to demonstrate how to construct such a bound structure using known relationships between different heuristic solutions, related structures, and the optimal solution.

A.2 Constrained Shortest Path Problem

A.2.1 Example DAD Length and Predecessor Tables

After Iteration 0

L

Node\β	0	1	2	3	4	5
s	0	0	0	0	0	0
1	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞
t	∞	∞	∞	∞	∞	∞

P

Node\β	0	1	2	3	4	5
s	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1
t	-1	-1	-1	-1	-1	-1

After Iteration 1

L

Node\β	0	1	2	3	4	5
s	0	0	0	0	0	0
1	∞	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
3	∞	1	∞	∞	∞	∞
4	∞	∞	∞	∞	∞	∞
t	∞	∞	∞	∞	∞	∞

P

Node\β	0	1	2	3	4	5
s	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1
3	-1	s	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1
t	-1	-1	-1	-1	-1	-1

After Iteration 2

L

Node\β	0	1	2	3	4	5
s	0	0	0	0	0	0
1	∞	∞	0.5	∞	∞	∞
2	∞	∞	3	∞	∞	∞
3	∞	1	1	∞	∞	∞
4	∞	∞	∞	∞	∞	∞
t	∞	∞	∞	∞	∞	∞

P

Node\β	0	1	2	3	4	5
s	-1	-1	-1	-1	-1	-1
1	-1	-1	s	-1	-1	-1
2	-1	-1	3	-1	-1	-1
3	-1	s	s	-1	-1	-1
4	-1	-1	-1	-1	-1	-1
t	-1	-1	-1	-1	-1	-1

After Iteration 5

L

Node\β	0	1	2	3	4	5
s	0	0	0	0	0	0
1	∞	∞	0.5	0.5	0.5	0.5
2	∞	∞	3	3	1	1
3	∞	1	1	1	1	1
4	∞	∞	∞	2	2	2
t	∞	∞	∞	4	3	2

P

Node\β	0	1	2	3	4	5
s	-1	-1	-1	-1	-1	-1
1	-1	-1	s	s	s	s
2	-1	-1	3	3	1	1
3	-1	s	s	s	s	s
4	-1	-1	-1	3	3	3
t	-1	-1	-1	2	4	2

Figure A.1: DAD L and P Tables for Figure 2.3

A.2.2 Lagrangian Relaxation for Resource Constrained Problems

The Lagrangian relaxation moves the knapsack constraint (2.2.4) in the IP formulation into the objective function and leaves only flow balance and

binary domain constraints to define the feasibility region [Xiao et al., 2005]. This relaxation is very effective because the resulting problem is simply a shortest path problem with the objective coefficients as a weighted sum of distance and cost. Thus, the relaxed problem can be further relaxed to a linear program and will still produce an integer solution. The key is to find a Lagrangian multiplier, λ, that minimizes the Lagrangian relaxation objective (2.2.6).

$$\text{Minimize} \sum_{(i,j)\in E} ((d_{ij} + \lambda c_{ij}) * x_{ij}) - \lambda\beta \tag{A.2.1}$$

subject to:

$$\sum_{j\in N} x_{ij} - \sum_{j\in N} x_{ji} = \begin{cases} 1 & \text{if } i == s \\ -1 & \text{if } i == t \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in N \tag{A.2.2}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i,j) \in E \tag{A.2.3}$$

Figure A.2: RCSPP Lagrangian Relaxation IP Formulation

A.2.2.1 LARAC-BIN

LARAC-BIN seeks to find the best Lagrangian multiplier by performing a binary search over the values [Xiao et al., 2005]. First, an upper and lower bound on the optimal Lagrangian multiplier are set and bisected to choose a multiplier. This multiplier is then used to augment the edge distances. Then Dijkstra's algorithm is used to find the shortest path with the cost-augmented edge distances. Depending on the result of Dijkstra's, the upper and lower

bounds on the Lagrangian multiplier is altered. The process of bisection, Dijkstra's, and multiplier bound updating continues until the termination criteria has been met. This criteria can be set such that the difference between the heuristic solution error has a known bound, or an actual optimality condition can be enforced. The details of LARAC-BIN can be seen in Algorithm 19. In the algorithm, the third parameter of $Dijkstra(s, t, \cdot)$ indicates which edge

attributes to use when determining the paths.

Input: N, E, s,t,β, τ
Output: Shortest distance path with cost $\leq \beta + \tau$
$p_{dist} = Dijkstra(s, t, distance)$
if $c(p_{dist}) \leq \beta$ **then**
  return p_{dist}
end
$p_{cost} = Dijkstra(s, t, cost)$
if $c(p_{cost}) > \beta$ **then**
  return *Infeasible*
end
if $c(p_{cost}) == \beta$ *or* $d(p_{cost}) == d(p_{cost})$ **then**
  return p_{cost}
end
$\lambda_0 = 0$
$\lambda_1 = \frac{d(p_{cost}) - d(p_{dist})}{\beta - c(p_{cost})}$
while $\frac{\lambda_1 - \lambda_0}{\beta - c(p_{cost})} > \tau$ **do**
  $\lambda = \frac{\lambda_1 + \lambda_0}{2}$
  $c_\lambda = \{d(i,j) + \lambda * c(i,j), \forall (i,j) \in E\}$
  $p = Dijkstra(s, t, c_\lambda)$
  if $c(p) == \beta$ **then**
    return p
  else if $c(p) < \beta$ **then**
    $\lambda_1 = \lambda$
  else
    $\lambda_0 = \lambda$
  end
end
return $p = Dijkstra(s, t, c_{\lambda_1})$

Algorithm 19: LARAC-BIN

Aside from the termination criteria, the initial Lagrangian multiplier bounds serve as the most important definitions in the algorithm. Xiao et al. provide theorems outlining their choice. These bounds need to enclose the optimal multiplier, otherwise the binary search cannot identify it. Obviously,

the lower bound begins at 0. The choice in upper bound is not as obvious, however. The upper bound is initialized to $\frac{d(p_{cost})-d(p_{dist})}{\beta-c(p_{cost})}$, where p_{cost} and p_{dist} are the lowest cost and lowest distance paths, respectively. It is clear $d(p_{cost}) \geq d(p_{dist})$ and $\beta \geq c(p_{cost})$. Additionally, p_{dist} must be infeasible given the knapsack constraint, otherwise it would have been returned in an earlier step. Let p_{λ_1} be the optimal path found using c_{λ_1}. Lemma A.2.1 shows p_{λ_1} is feasible and because it is feasible, $d(p_{\lambda_1})$ is at least $d(p_{dist})$. Additionally, any λ greater than the initial λ_1 leads to a longer path than p_{λ_1} Therefore, the initial λ_1 is an upper bound on the λ value leading to the best solution. The proof for Lemma A.2.1 is analogous to the two-line proof from Xiao et al. but shows additional steps that are not obvious upon first inspection [Xiao et al., 2005].

Lemma A.2.1. *If* $\lambda = \frac{d(p_{cost})-d(p_{dist})}{\beta-c(p_{cost})}$, $c(p_{cost}) < \beta$, *and* $d(p_{cost}) > d(p_dist)$, *then* p_λ *is feasible.*

Proof. See [Xiao et al., 2005]. □

Bibliography

[Aneja et al., 1983] Aneja, Y. P., Aggarwal, V., and Nair, K. P. K. (1983). Shortest chain subject to side constraints. *Networks*, 13(2):295–302.

[Applegate et al., 2007] Applegate, D. L., Bixby, R. E., Cook, W. J., Applegate, D. L. L., Bixby, R. E. E., Chvátal, V., Cook, W. J. J., Cook, W. J. J., and Chvtal, V. (2007). *The Traveling Salesman Problem : A Computational Study*. Princeton University Press, Princeton, UNITED STATES.

[Bellman, 1962] Bellman, R. (1962). Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63.

[Bentley, 1992] Bentley, J. J. (1992). Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4(4):387–411.

[Bianco et al., 1994] Bianco, L., Mingozzi, A., Ricciardelli, S., and Spadoni, M. (1994). Exact and heuristic procedures for the traveling salesman problem with precedence constraints, based on dynamic programming. *INFOR: Information Systems and Operational Research*, 32(1):19–32.

[Boland et al., 2006] Boland, N., Dethridge, J., and Dumitrescu, I. (2006). Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34:58–68.

[Campbell, 2006] Campbell, A. M. (2006). Aggregation for the probabilistic traveling salesman problem. *Computers and Operations Research*, 33(9):2703 – 2724. Part Special Issue: Anniversary Focused Issue of Computers and Operations Research on Tabu Search.

[Carr and Vempala, 2004] Carr, R. and Vempala, S. (2004). On the Held-Karp relaxation for the asymmetric and symmetric traveling salesman problems. *Mathematical Programming*, 100(3):569–587.

[Chan, 1996] Chan, T. M. (1996). Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16(4):361–368.

[Chen et al., 2008] Chen, S., Song, M., and Sahni, S. (2008). Two techniques for fast computation of constrained shortest paths. *IEEE/ACM Transactions on Networking*, 16(1):105–115.

[Christofides, 1976] Christofides, N. (1976). Worst-case analysis of a new heuristic for the traveling salesman problem. *Carnegie Mellon University*, page 10.

[Cook, a] Cook, W. J. Concorde TSP solver. `http://www.math.uwaterloo.ca/tsp/concorde/index.html`.

[Cook, b] Cook, W. J. National traveling salesman problems. `http://www.math.uwaterloo.ca/tsp/world/countries.html`.

[Cook, c] Cook, W. J. Status of national TSP instances. `http://www.math.uwaterloo.ca/tsp/world/summary.html`.

[Cook, d] Cook, W. J. VLSI data sets. `http://www.math.uwaterloo.ca/tsp/vlsi/index.html`.

[Cook, 2011] Cook, W. J. (2011). *In Pursuit of the Traveling Salesman : Mathematics at the Limit of Computation*. Princeton University Press, Princeton, UNITED STATES.

[Cutler, 1980] Cutler, M. (1980). Efficient special case algorithms for the n-line planar traveling salesman problem. *Networks*, 10(3):183–195.

[de Berg et al., 2008] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer.

[Deineko et al., 1994] Deineko, V. G., van Dal, R., and Rote, G. (1994). The convex-hull-and-line traveling salesman problem: a solvable case. *Information Processing Letters*, 51(3):141 – 148.

[Desrochers and Soumis, 1988] Desrochers, M. and Soumis, F. (1988). A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR: Information Systems and Operational Research*, 26(3):191–212.

[Eilon et al., 1971] Eilon, S., Watson-Gandy, C. D. T., and Christofides, N. (1971). *Distribution management: Mathematical modelling and practical analysis*. London : Griffin.

[Feillet et al., 2004] Feillet, D., Dejax, P., Gendreau, M., and Gueguen, C. (2004). An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks*, 44(3):216–229.

[García and Tejel, 1997] García, A. and Tejel, F. J. (1997). The two-convex-polygons TSP: A solvable case. *Top*, 5(1):105–126.

[Goel et al., 2001] Goel, A., Ramakrishnan, K. G., Kataria, D., and Logothetis, D. (2001). Efficient computation of delay-sensitive routes from one source to all destinations. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 854–858. IEEE.

[Golden et al., 1980] Golden, B., Bodin, L., Doyle, T., and Stewart, W. (1980). Approximate traveling salesman algorithms. *Operations Research*, 28(3):694–711.

[Graham, 1972] Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133.

[Gutin et al., 2002] Gutin, G., Punnen, A. P., and Gutin, G. (2002). *The Traveling Salesman Problem and Its Variations*. Springer, New York, NY, UNITED STATES.

[Held and Karp, 1962] Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210.

[Held and Karp, 1970] Held, M. and Karp, R. M. (1970). The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162.

[Held and Karp, 1971] Held, M. and Karp, R. M. (1971). The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25.

[Helsgaun, a] Helsgaun, K. LKH. `http://akira.ruc.dk/~keld/research/LKH/`.

[Helsgaun, b] Helsgaun, K. LKH 2.0 user guide.

[Helsgaun, 2000] Helsgaun, K. (2000). An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130.

[Jarvis, 1973] Jarvis, R. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18 – 21.

[Johnson et al., 1996] Johnson, D. S., Mcgeoch, J., and Rothberg, E. E. (1996). Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 341–350.

[Karp, 1977] Karp, R. M. (1977). Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research*, 2(3):209–224.

[Kubo and Kasugai, 1991] Kubo, M. and Kasugai, H. (1991). The precedence constrained traveling salesman problem. *Journal of the Operations Research Society of Japan*, 34(2):152–172.

[Liew, 2012] Liew, S. (2012). Introducing convex layers to the traveling salesman problem. *arXiv preprint arXiv:1204.2348*.

[Lin, 1965] Lin, S. (1965). Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269.

[Lin and Kernighan, 1973] Lin, S. and Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516.

[Mulder and Wunsch, 2003] Mulder, S. A. and Wunsch, D. C. (2003). Million city traveling salesman problem solution by divide and conquer clustering with adaptive resonance neural networks. *Neural Networks*, 16(5):827 – 832. Advances in Neural Networks Research: IJCNN '03.

[Papadimitriou, 1977] Papadimitriou, C. H. (1977). The Euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237 – 244.

[Platzman and Bartholdi, 1989] Platzman, L. K. and Bartholdi, III, J. J. (1989). Spacefilling curves and the planar travelling salesman problem. *Journal of the ACM*, 36(4):719–737.

[Pugliese and Guerriero, 2013] Pugliese, L. D. P. and Guerriero, F. (2013). A survey of resource constrained shortest path problems: Exact solution approaches. *Networks*, 62(3):183–200.

[Reinelt, 1995] Reinelt, G. (1995). The TSPLIB symmetric traveling salesman problem instances. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf`.

[Righini and Salani, 2006] Righini, G. and Salani, M. (2006). Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization*, 3(3):255 – 273. Graphs and Combinatorial Optimization.

[Rosenkrantz et al., 1974] Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. (1974). Approximate algorithms for the traveling salesperson problem. In *Switching and Automata Theory, 1974., IEEE Conference Record of 15th Annual Symposium on*, pages 33–42. IEEE.

[Rote, 1992] Rote, G. (1992). The n-line traveling salesman problem. *Networks*, 22:91 – 108.

[Stabler,] Stabler, B. Tansportation networks. `https://github.com/bstabler/TransportationNetworks`.

[Tan et al., 2019] Tan, P. N., Steinbach, M., Karpatne, A., and Kumar, V. (2019). *Introduction to Data Mining*. Pearson Education, Inc.

[Tilk et al., 2017] Tilk, C., Rothenbächer, A.-K., Gschwind, T., and Irnich, S. (2017). Asymmetry matters: Dynamic half-way points in bidirectional labeling for solving shortest path problems with resource constraints faster. *European Journal of Operational Research*, 261(2):530 – 539.

[Valenzuela and Jones, 1997] Valenzuela, C. L. and Jones, A. J. (1997). Estimating the Held-Karp lower bound for the geometric TSP. *European Journal of Operational Research*, 102(1):157 – 175.

[Wolsey, 1980] Wolsey, L. A. (1980). *Heuristic analysis, linear programming and branch and bound*, pages 121–134. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Xiang et al., 2015] Xiang, Z., Chen, Z., Gao, X., Wang, X., Di, F., Li, L., Liu, G., and Zhang, Y. (2015). Solving large-scale TSP using a fast wedging insertion partitioning approach. *Mathematical Problems in Engineering*, 2015.

[Xiao et al., 2005] Xiao, Y., Thulasiraman, K., Xue, G., and Jüttner, A. (2005). The constrained shortest path problem: Algorithmic approaches and an algebraic study with generalization. *AKCE International Journal of Graphs and Combinatorics*, 2(2):63–86.

[Zheng Wang and Crowcroft, 1996] Zheng Wang and Crowcroft, J. (1996). Quality-of-service routing for supporting multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1228–1234.

Vita

Patrick Alexander Mannon was born in Urbana, Illinois. He received the degree of Bachelor of Science from Washington University in St. Louis in May 2015. In August 2015, he entered the Graduate School at the University of Texas at Austin.

Permanent address: patrick.mannon@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.