A PARALLEL IMPLEMENTATION OF A GREEDY TSP ALGORITHM

by

Andres Gonzalez, B.S., M.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2020

Committee Members:

Martin Burtscher, Chair

Wuxu Peng

Kecheng Yang

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

To my beautiful fiancé, as we continue to live the journey that is exploring the world and learning what it means to live life. Thank you for encouraging me to pursue the challenge, to never give up, for the inspiration to continue in the full face of hardship, and for the car rides to campus during the early start of my 8 am classes. I can't wait to live the rest of our lives together! I love you, Allison!

To my best man and forever friend and his wife, Adam and Ashton Shelstead, for the patience and inspiration of what it takes to uphold the meaning of friendship. This pursuit took time that could have otherwise been spent together away. Nevertheless, time and time again, patience, tolerance, and forgiveness prevailed. Love you guys!

ACKNOWLEDGEMENTS

Thank you to my advisor, Dr. Martin Burtscher, for the unbelievable source of inspiration, knowledge, and patience. Thank you for your guidance in this chapter of my journey, as I will long cherish it as the beginning of a new phase of my life.

Thank you to my graduate advisor, Dr. Wuxu Peng, for meeting with me on a random day three years ago when an unknown prospective student makes his way into your office hours to say "I'd like to earn a Master's degree. How can I go about this?" Here we are, three years later, fulfilling what is a small goal in a much grander vision.

Thank you to my committee member, Dr. Kecheng Yang, for the enthusiasm you continue to show beyond your lectures and for being a central part of my endeavors.

Thank you to Mr. Victor Grifols Lucas and his company, GRIFOLS, for both providing me the means and allowing me the time to conquer this enormous accomplishment. A colossal thank you to my manager Bradley Erwin and my supervisors, James Giger, Deanna Leon, and Melanie Lindsey. My extended gratitude also goes to the staff at GRIFOLS' Routine Laboratory, for providing the encouragement to continue and succeed. Without your collective support, this undertaking would absolutely not have been possible!

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
TSP	Traveling Salesman Problem
CONCORDE	Combinatorial Optimization and Networked Combinatorial Optimization Research And Development Environment
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CUDA	Compute Unified Device Architecture
MST	Minimum spanning tree
API	Application Programming Interface
IHC	Iterative Hill Climb
ACO	Ant Colony Optimization
MMAS	Max-Min Ant System

ABSTRACT

The Traveling Salesman Problem has often been used as an exploration ground for building heuristics to calculate the shortest path of a complete graph that traverse every vertex exactly once.

This has a number of important practical applications. Since it is an NP-hard problem, many heuristics have been proposed to obtain near-optimal solutions in polynomial time.

The heuristic explored in this study is based on Kruskal's algorithm, where the edges of a graph are sorted in non-decreasing order. The smallest edges that meet the eligibility criteria are included in the path until a tour that includes all vertices has been constructed. Whether an edge meets the criteria depends on which smaller edges have been inserted. This makes the algorithm difficult to parallelize.

I combined previously published with new parallelization techniques and implemented the algorithm in OpenMP and CUDA. The resulting GPU code is very fast, taking just 0.06 seconds to process a graph with 11,849 vertices and 70,193,476 edges. This is approximately 8 times faster than the serial CPU implementation and has a solution quality that is within 18% of the optimal. Compared to the optimal solver CONCORDE, my code is 1,206,302 times faster.

I. INTRODUCTION

The Traveling Salesman Problem (TSP) is a puzzle-like rendition of a question that simply poses "Which is the shortest route that a salesperson can take to visit all cities on a map exactly once and return to the starting city while keeping the distance of the route at a minimum?". In a complete and undirected graph, the vertices are analogous to the cities, and all the edges available from each vertex to every other vertex is analogous to the paths a salesperson can take. The distances between the vertices correspond to the edge weights between the cities.

One can see that the number of available route options quickly becomes overwhelming as the number of cities grows, where the number of edges available is given by the equation $E = n \cdot (n-1) / 2$, where E is the number of edges in the given graph and n is the number of vertices. As the number of vertices in a graph grow, the number of edges grows quadratically, and the number of paths P exponentially $P = n!$. With the vast number of possibilities available, the kind of problem posed by TSP is known as a Combinatorial Optimization Problem, and its primary goal is to find the Hamiltonian Path—that is, the path that traverses all vertices exactly once, keeping the cost (the weight of the used edges) associated with traversing the path at a minimum before returning to the starting vertex.

The TSP has been used as a topic of study for a wide range of applications in science and engineering such as radiation hybrid mapping, where specific markers are located on a chromosome [2, 3, 22]. It is also used in Very Large-Scale Integration microchip manufacturing [6], X-ray crystallography [8, 17], routing, and production-

scheduling [9]. It has been used to solve the path for which a laser cuts a board for wiring and has applications in network optimizations [5]. It also has applications in computer wiring, wallpaper cutting, and hole punching of metallic sheets in manufacturing plants [9]. Thus, finding time-efficient solutions is of great importance.

Finding the optimal solution to a given TSP is a computationally-expensive task, where the problem is categorized as NP-hard and quickly becomes intractable for large problem sizes [4, 6, 16]. As a workaround to finding the optimal solution, there are a variety of heuristics that can achieve high-quality, *near*-optimal solutions, and each heuristic is associated with its own time-complexity. Some heuristics are faster than others, but they all have polynomial time.

Heuristics vary in their time complexity, ranging from O(n), where the amount of time needed to solve for a particular heuristic is linearly based on the input size n, to $O(n^2)$, where the time needed is correlated with the square of the input size, and the complexities can go well beyond that (i.e. $O(n^3)$, or $O(n^4)$, etc.). Optimal solvers such as the Combinatorial Optimization and Networked Combinatorial Optimization Research and Development Environment (CONCORDE) [20] solve a given TSP graph to the lower bound [10]. Where the TSP has been solved by CONCORDE for optimal solutions of a given set of graphs, finding the optimal solution for practical applications is not always feasible. One could theoretically compute an optimal solution by brute-forcing a given graph and analyzing every possible combination of edges. However, this is unworkable for practical applications involving large input sizes. Therefore, researchers aim to find heuristics that arrive at *near*-optimal paths in workable time. Depending on the chosen

heuristic, there is typically a tradeoff between solution quality—that is, the relative difference in the heuristic's final path length to the optimal—and computation time.

For example, an easily-graspable heuristic can be thought of as "Random". A final path, or *tour*, is generated by selecting random, unvisited vertices from the graph as part of the current vertex's next destination. Associated with a time-complexity of *O(n)*, this heuristic's time to execute is linearly correlated with the input size, and the final solution could provide for a path length that is anywhere between the lower and upper bound of possible path lengths. Another heuristic is the "Nearest-Neighbor", where a vertex's destination is chosen according to its distance from the current vertex. The time-complexity associated with the Nearest-Neighbor heuristic, however, is $O(n^2)$, where every other vertex that has not been visited needs to be considered as a possible next destination. The closest vertex is chosen, and subsequently, every other vertex must again be compared for distance from the chosen vertex. This repeats until the final path is constructed, and the path constructed by the Nearest-Neighbor heuristic is usually within 25% of the lower bound at the trade-off of requiring $O(n^2)$ time.

The Greedy Heuristic that was implemented in this study is based on Kruskal's Algorithm [23, 24, 27], where the tour that is constructed is dependent on the edge weights of the graph. This heuristic takes all the edges in a complete, undirected graph and sorts them in non-decreasing order. The edges are then considered, starting with the smallest, on an individual basis on whether they should be included in the final tour. Whether they are included ultimately depends on two conditions: whether (1) both vertices associated with the edge in question have a degree of less than 2, or (2) if both

vertices of the edge have a degree of 1, they must not be a part of the same set of connected edges. Having a vertex that has a degree higher than 2 would result in an invalid final path. This is due to the fact that a vertex with a degree of 3 or higher ultimately does not constitute a tour—that is, it does not allow for a vertex to be visited *exactly* once in the graph cycle that connects all vertices. If a branching path were to be constructed from a vertex with a degree of 3, the *return* to that vertex (the edge leading to the vertex's second visit) would prevent a valid solution. Likewise, including an edge that would result in a cycle prior to the inclusion of all vertices in the tour would also be invalid because the cycle would not include all vertices.

This heuristic proves difficult to parallelize due to the inherent sequential nature of considering each edge on an individual basis. In other words, the inclusion of an edge cannot be determined based solely on any characteristic or trait associated with that edge—for example, its vertex at one end or the other or the edge's weight—without consideration of all other edges, and its inclusion must be considered in relation to what edges have already been included in the tour. It is only after previous edges have been included or excluded that any edge can be considered for inclusion in the tour. Nonetheless, there remain opportunities within the heuristic for parallelization. For example, this heuristic requires sorting. For implementation on the central processing unit (CPU), the sorting method is a form of recursive quicksort that makes use of the standard library's std::partition and std::find functions, both of which can be called with a parallel execution policy. For the graphics processing unit's (GPU) implementation, functions similar to std::partition can be found in a library known as "CUB" [42].

In this thesis, I serially implemented the presented heuristic. I subsequently implemented the parallel CPU version using a combination of OpenMP and parallel standard library functions, and finally implemented the parallel version of the presented greedy heuristic on the GPU using NVIDIA's Compute Unified Device Architecture (CUDA), an application programming interface (API) developed by NVIDIA for general purpose computing on GPUs.

II. RELATED WORK

CONCORDE, developed by David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook, is an optimal solver that has previously been used to benchmark and publish a set of TSP graphs, the largest of which has had 85,900 cities [5]. CONCORDE uses linear programming and branch-and-bound techniques to solve for the optimal solutions [10, 12], and can only find optimal solutions in sub-exponential time for certain inputs. These benchmarks serve as a good reference point for comparison against heuristics that have been written in an attempt to achieve near-optimal solutions. The following are several examples of heuristics.

Random

This heuristic is relatively easy to grasp considering there is no particular bearing on the order of the visited cities. Essentially, a random city is chosen as the tour is being constructed, and therefore, the final tour length could range anywhere from best- to worst-case—that is, the final path may be anywhere from the lower to the upper bound. It generates a path in $O(n)$, but may produce a solution quality that is anywhere between the lower and upper bound.

Sequenced

Unlike the Random heuristic, the Sequenced heuristic already has its path preconstructed, meaning that every vertex has a predetermined “next vertex” in its path, regardless of the weight of the edge connecting the two vertices. Time complexity

is also *O(n)*, but due to its predetermined order, reveals no new information or potential for exploration.

"Smarter" Heuristics

The heuristics that follow are examples of improved, "smarter" heuristics that can consistently achieve solutions that are closer to the lower-bound. However, their implementations require more time to solve.

Nearest Neighbor

As an introduction to the first heuristic that has a type of algorithmic or rule-based approach, the Nearest Neighbor [9, 19] heuristic provides insight into how applying constraints yields better results. In this heuristic, a starting city is randomly chosen from the given set of vertices. The path's "next vertex" is determined by comparing all edges formed from the current vertex to every other vertex. The shortest edge among them is chosen, and the vertex located on the other end of the edge is added to the tour. The process then repeats for every other vertex, ignoring the vertices that have already been visited along the way. Ultimately, the final path tends to have a result that is within 25% of the lower bound, but the trade-off comes in the heuristics time-complexity $O(n^2)$, where the amount of time that is needed to calculated the final path is dependent on the square of the number of vertices in the graph.

Nearest Insertion

The Nearest Insertion heuristic uses a combination of rules for determining the specific order of which vertex to visit next [11, 19]. Given the current set of vertices that have been included in the tour, find the vertex that is closest to *any* vertex in the tour. To determine where in the tour to include the vertex—that is, to determine which edges should be used to include that vertex in the tour—find an insertion point between two cities where the insertion cost is minimal. In other words, where can the vertex be inserted such that the increase in tour length is minimized when compared to all other possible insertion points. Similar in the time complexity for the Nearest Neighbor heuristic, all possibilities must be considered for every vertex's iteration during the construction of the path.

MST-based

The minimum spanning tree (MST)-based heuristic uses the MST to create a route that attempts to find ways around already-visited nodes in $O(n^2 \cdot \log_2(n))$ time [12]. It begins by first creating an MST, and subsequently, a path is created by traversing all the vertices from that tree. When the tree branches, the terminal node at the end of the branch is connected to the next node n_i that follows when returning from the traversal of that branch. This skips over duplicate vertices and ultimately forms the Hamiltonian Path by removing the edges that connect the origin of the branch to that next node n_i.

Christophides

A multi-step heuristic known as Christophides' heuristic [9, 13] has a unique approach such that even though its time-complexity is $O(n^3)$, the final path is within 10% of the lower bound. Similar to the MST-based heuristic, Christophides' heuristic begins by forming an MST. From the tree, all odd-degree vertices are removed from which a minimum-weight matching graph is made. For the configuration in which the matching of vertices is optimal, they are matched back with the minimum-spanning tree. An Eulerian cycle (a path that visits every *edge* once) is drawn on the resulting combined graph which then results in the Hamiltonian Path.

Nearest Merger

The Nearest Merger heuristic takes an approach that is similar to that of the Nearest Insertion heuristic, however, the Nearest Merger takes into account sets of vertices and vertices that might have already been included in other sets [13]. To elaborate on this, consider that all vertices begin as part of their own sub-tour. In order to consider what merging to do next, the heuristic finds the two points that are nearest each other and combines their sub-tours depending on a small set of rules that minimizes the cost of combination. If either of the vertices are independent (they are still their same, initial-state sub-tour), simply add the independent vertex to the other's sub-tour. If both sub-tours contain more than 1 vertex, find an edge in each sub-tour such that the cost of removal of those edges and addition of two other edges that connect each sub-tour's vertices from the removed edges to each other is minimal. The

heuristic repeats until all sub-tours have been combined and one final tour remains. Again, this is a heuristic in which all combinations of edges and vertices must be considered while the tour is being constructed, giving the heuristic a time complexity of $O(n^2)$.

Clark-Wright

As a more complicated heuristic, the Clark-Wright heuristic [14] also makes use of a randomly chosen vertex as its starting point. The heuristic then creates a path to every other vertex in the graph, meaning that every vertex is visited and the path immediately returns to the starting vertex before visiting another. This produces a starting cost, which is then used to compute a savings cost as the heuristic proceeds. A list of saving-costs is generated by drawing an edge between all other vertices as if their path to the starting vertex was removed (i.e. starting vertex *a* has paths with *b* and *c*; instead of traversing via *a*-*b*-*a*-*c*-*a*, a savings cost is calculated by assessing path *a*-*b*-*c*-*a*). This list of saving-costs is then sorted, the smallest of which is added to the tour. This process is then repeated until a proper, final cycle is formed between all the vertices.

Other Heuristics

The list of heuristics is virtually endless, ranging from simple to rather complicated with each heuristic offering insight and benefits that others might not, each with their own time- and space- complexity. Furthermore, there are a couple of distinct categories that heuristics fall into termed as constructive or improvement [3].

Constructive heuristics aim to build a solution from scratch in that, as soon as a solution is found, the heuristic is complete and terminates, the best of which are capable of achieving between 10-15% of the optimal solution. Improvement heuristics, however, take a premeditated, initial solution and iterate over successive calls that *improve* upon the solution (i.e. exchanging sets of edges that reduce the overall path length). Such heuristics include the K-Opt [18, 19, 20], the Held-Karp heuristic [11], the Lin-Kernighan heuristic [20] (which is an adaptive form of 2-Opt and 3-Opt) and its optimized Lin-Kernighan-Helsgaun (LKH-1 and LKH-2) heuristics [9]. Improvement heuristics can be combined as an additional post-processing-step to construction heuristics such as the Greedy Heuristic presented here, but that is beyond the scope of this thesis.

Parallelization

Various heuristics each offer their own opportunities for parallelization, but oftentimes, direct parallelization of each heuristic may not be feasible. Heuristics at times require a certain degree of dependencies that play a key role in what algorithmically happens next, be it choosing a next vertex, adding or removing an edge, or calculating a cost. Other parts of heuristics are key opportunities for parallel implementation (i.e. sorting), and it is these opportunities that can be exploited to optimize a heuristic. Parallelization by means of the CPU could be as simple as using a predefined, standard library function. APIs such as OpenMP, MPI, and the usage of pthreads each offer their own systematic advantages for parallel implementations.

There have been a variety of multi-threaded approaches that have been implemented as a means of finding the MST [13, 20, 24], and this also includes an algorithm that makes use of "helper threads" as a means of "searching ahead" in the heuristic [25]. In Kruskal's Algorithm, edges are sorted based on their lengths/weights, and edges are considered for inclusion in the final tour based on their cyclic-free connections. The approach presented by Katsigiannis et al. [25] suggests that "helper threads" can scan "ahead of the list" for edges that would form a cycle based on the edges that have already been included *so far.* This means that the edges that meet this condition can be excluded *early*, allowing the algorithm to essentially skip over the edges that have already been discarded and thus saving computation time. The authors showed that this approach allowed for a speed up of 5.5 for 8 helper threads, revealing that this approach exhibits usefulness.

As a further possibility of optimization, parallelization via the use of a GPU offers an advantage over using CPU parallelization. GPUs offer better performance when it comes to the execution of parallel code, especially when the same instruction sets need be applied to large amounts of data. Utilizing this advantage over CPU parallelization could offer faster execution times for particular heuristics.

O'Neil and Burtscher [4] have shown that parallelism on the GPU can benefit in the optimization of the 2-Opt CPU-based heuristic, though at the tradeoff of smaller problem sizes. The 2-Opt heuristic sought to exchange pairs of edges in a graph for edges that allow the tour to remain intact whilst reducing the overall path's length. In an approach termed the Iterative Hill Climb (IHC) [10, 15, 21, 26], a graph can undergo

many 2-Opt exchanges to reach a point where the graph's path length can no further be reduced—that is, it reaches a local minimum. Through many repetitions of the IHC, each beginning with a randomly-constructed tour at the start of the heuristic, many local minimums can be compared and eventually agree on what result is nearest the optimal solution (and therefore achieves a near-optimal solution). With parallelization, many threads can simultaneously run independent climbs on the same graph, and when ported to CUDA, their GPU code reflected that of a 61X speedup over the serial code. To re-emphasize, however, memory constraints on the GPU limited the problem size to 110 cities. O'Neil concludes that the GPU implementation is roughly equivalent to the performance of a pthreads implementation with 32 CPUs and 8 cores per CPU, which clearly shows that the GPU can improve parallel performance over the CPU. Rocki and Suda [8] were able to demonstrate the heuristic with an approach that allowed approximately 6,000 cities on the GPU, taking into account that the distance (where needed) could be recalculated at runtime rather than store the distances in a matrix. Though it slowed the implementation, the storage requirement was reduced from $O(n^2)$ to $O(n)$. They were also able to demonstrate that another GPU implementation based on the 2-opt heuristic achieved a 5 to 45 times faster run time compared to the parallel CPU implementation on 6 cores [15].

Delévacq and others [31] describe an implementation that parallelizes a metaheuristic (a category of heuristics that can be used to generally solve a wider range of optimization problems rather than a more specific problem) known as the Ant Colony Optimization (ACO) on the GPU—specifically, the Max-Min Ant System (MMAS), which is

considered to be one of the most efficient ACO algorithms [30, 32, 41]. It is a constructive heuristic that uses an approach that is inspired by the behavior of ants when building a colony, and parallelization of ACO also comes in two forms: the building of a single ant colony and the building of multiple ant colonies. In the single-colony implementation, individual ants are used as separate computing elements, whereas in the multiple colony implementation, each colony is distributed. This is further enhanced by the incorporation of a cooperation scheme between colonies that aims to reduce computing time and improve solution quality. Delévacq [31] shows that a speedup of nearly 20X can be achieved, albeit different problem sets yielded different variations of speedups. Hongtao and others [30], as well as Skinderowicz [33], were also able to show considerable speedups when using CUDA for the parallelization of the multiple colony ACO. Chen and Chien [34] propose another parallel variation of the ACO heuristic that is based on genetics, but to the best of our knowledge, has not yet been implemented on the GPU. Additionally, others [10] describe heterogenous variations of MMAS in which some component of the metaheuristic is computed on the GPU while other components are computed on the CPU, both of which are executed in parallel. Bai and others [30] describe a multiple-colony, parallel, GPU implementation that also parallelizes individual ants and can achieve a speed up of 32x over the CPU serial implementation. Jiening and others [36] describe an implementation that is written in a combination of C++ and Cg, a variant of NVIDIA's coding language for utilizing their GPUs, that achieves a small speed up of 1.4x, but with a small problem size of 30 cities. Because the ACO metaheuristic involves two stages (tour construction and pheromone

updates), Cecilia and others [37] were able to demonstrate that the tour construction phase could achieve a speed up of 28x and the pheromone update phase could achieve a speedup of 20x when both phases are implemented on the GPU when compared to the serial CPU implementation (the pheromone update phase has typically been implemented on the CPU). Fu and others [38] describe an MMAS implementation using MATLAB and the Jacket toolbox (a tool for writing and executing MATLAB code on the GPU in MATLAB's native M-Language) in which the metaheuristic alternates phases between the CPU and GPU, with a general speedup of over 30x for the GPU+CPU implementation over the CPU implementation. Other various CUDA-implemented ACO TSP solvers exist [16, 39, 40].

In a work that is closely related to the approach of this study, Osipov et al. describe a manner of efficiently building an MST using Kruskal's algorithm [24]. In their approach, they make light of the fact that Kruskal's algorithm is used for building MSTs, but rather than using trees directly, a forest is grown. Separate trees are joined together by edges of increasing weight, but notably, Kruskal's algorithm sorts *all* the edges before iterating through them until $n - 1$ edges are found. The manner in which this inefficiency is addressed is by incorporating a form of "quicksort", where, much to the likes of our approach, quicksorts the "lighter" side of the quicksort, applies the selection algorithm, and if the MST isn't complete, recursively applies to the "heavier" side of the quicksort. This approach is denoted as *qKruskal*. To further improve upon the inefficiency, filtering—that is, the removal of heavier edges from the forest before iterating on them—is applied, which leads to considerable performance improvements.

III. APPROACH

The heuristic taken in this study is based on Kruskal's algorithm—using edges that are ordered by non-decreasing edge weights. TSPLIB [7], a standardized library of two-dimensional, Euclidian-space graphs that has been the basis for the majority of TSP research over the past couple of decades, was used for comparing the performance of the greedy heuristic in this study to others. In this heuristic, a TSPLIB file is read, instantiating all the vertices to be processed, and all possible edges between all vertices are subsequently instantiated. The edges are sorted in non-decreasing order, and the sorted list of edges is traversed, with each eligible edge being added to the final tour.

The edges are said to be sorted in non-decreasing order due to the technical difference in describing the process of sorting. A sorting method that sorts in "ascending" order implies that at every step of the sorted list, the sorting criteria increases at every step. The term "non-decreasing" is used to imply that at every step of the sorted list, the criteria *may* or *may not* increase.

Recall that an edge is considered eligible if either (1) both vertices associated with the edge in question have a degree of less than 2, and (2) if both vertices have a degree of 1, they must not be a part of the same set of connected edges (i.e. they must not form a cycle). During preprocessing, the C++ standard library's std::partition is called to rearrange the currently-processing range of edges in such a way that all edges that meet the criteria for inclusion in the final tour are moved to the front half of the set (see below for the implementation's definition of a "set"). This makes for a more efficient sorting due to fewer edges being recursively rearranged during the sorting step, and as

an additional optimization, the preprocessing step (as is the sorting) can be carried out in parallel (specifically, the calls to *std::partition* and *std::find* as is outlined below). This set is then passed to processing and sorting, where the preprocessed range of edges are sorted. If the range of edges exceeds the current chunk size, the edges that exceed the limit of the chunk are passed in a list to the next main loop's iteration. After the range of edges are processed and sorted, the main loop iterates through the range of edges, adding eligible edges to the final tour and discarding the ineligible ones. If the tour is complete ($n - 1$ edges have been added to the tour), the main loop terminates and the city-order is reconstructed from the array holding the city-connections (the edge array whose vertices appear in sequence). The algorithm terminates, and the final path length is calculated, marking the end of the program. The heuristic is outlined in **Figure 1.**

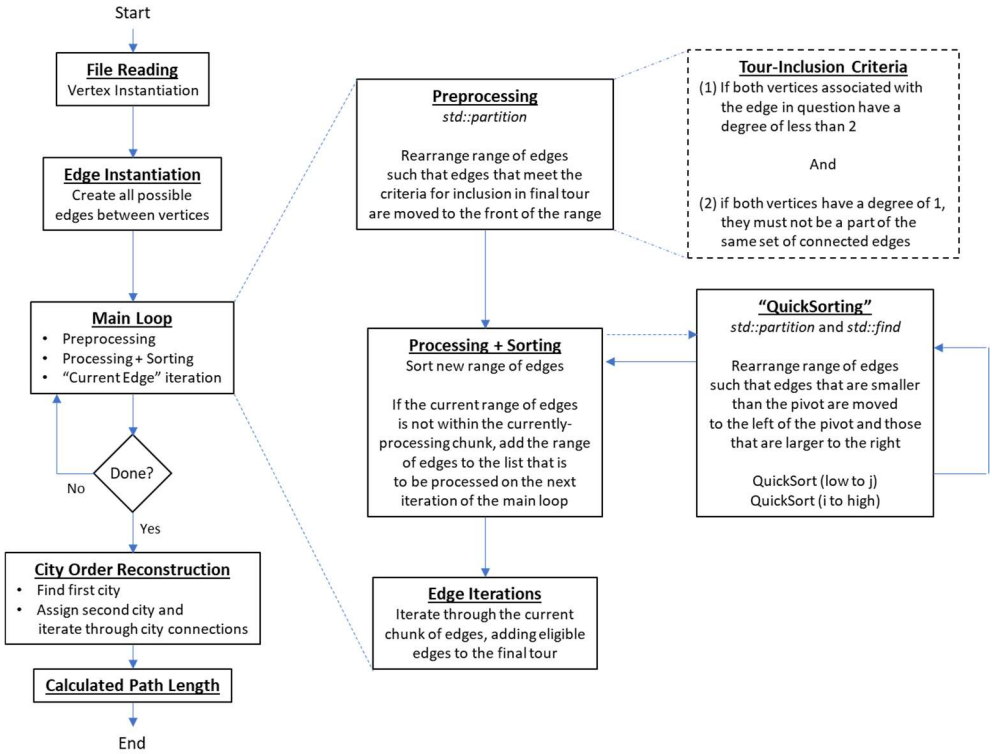

Figure 1 – The Heuristic's Implementation. Shown above is the current implementation's approach. The edges are then processed in the main loop, where a range of edges are preprocessed, processed, and sorted.

Std::Partition

The standard library's std::partition function is one of the two key components to parallelization in the current implementation (the other being std::find; see **Figure 2**). Partition takes as arguments an execution policy, a range, and a function that returns a Boolean that serves as the basis for partitioning the range. Partition takes the elements pointed to by the specified range and segregates them according to the evaluating function—it places the elements whose conditional-check function's return value is true in the front and sends the elements whose return value is false to the back of the range. In the preprocessing step, edges are rearranged according to whether they meet the criteria for inclusion in the final tour (that is, an edge's vertex degrees are checked and

whether the inclusion of the edge will prematurely form a cycle in the currently-constructed tour). All edges that meet the criteria are rearranged such that any edges that meet the criteria precede the edges that do not meet the criteria. This range, consisting solely of tour-eligible edges, is then passed into the processing step for sorting, and in the sorting step, a recursive quicksort in emulated by the same call to partition, albeit using a different condition. The condition differs in that the condition is now checking the length of the edges against a pivot edge. Edges that are smaller than the pivot are rearranged to precede the pivot, and edges that are larger are rearranged to follow the pivot edge. A recursive call to quicksort is then made with new ranges "low to pivot – 1" and "pivot + 1 to high". This recursive call continues until the range of edges (the range defined as the chunk size that is supplied at runtime) is fully sorted, for which the main loop can then iterate through for inclusion in the final tour.

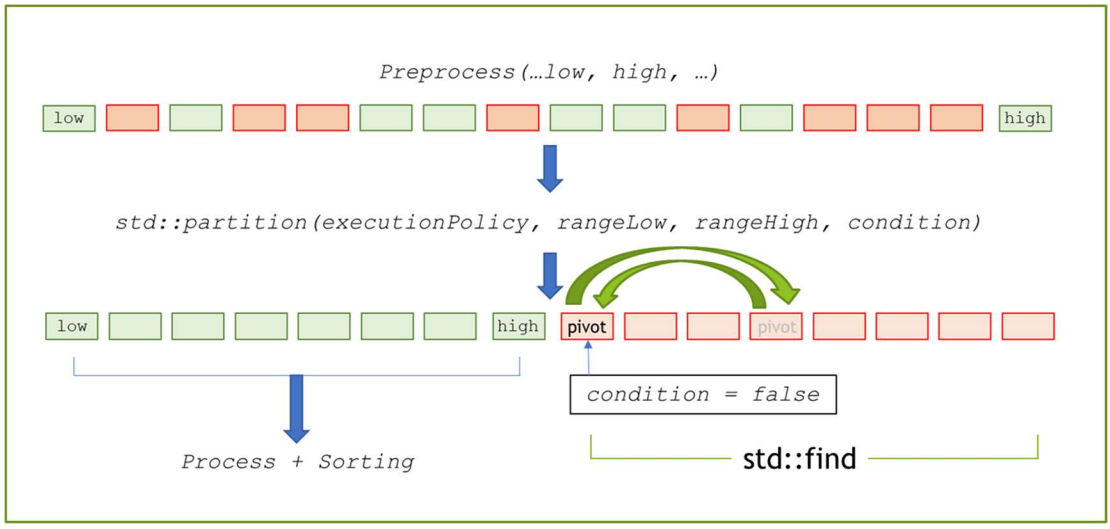

Figure 2 – Std::Partition and Std::Find. Shown above is the result of the call to partition before and after edges are rearranged.

Disjoint-Set Data Structure – "Union Find"

The approach taken to detect cycles in this study (one of two stipulations in the criteria for an edge's inclusion in the final tour) uses a data structure termed Union Find, otherwise also known as a Disjoint-Set Data Structure [24, 25]. Its usefulness comes to light in its implementation, where every element is associated with a set that is identified by the integer at that index. Initially, every element in an integer array is its index (where every vertex is a part of its own set). If the edge under consideration is included in the final Hamiltonian Path, the integer array is updated to reflect an association between the edge's two vertices. The data-structure accomplishes this by changing one of the index's value to reflect the value in the other index. As vertices become connected through the iteration of edges formed by those vertices, the identifying set that a particular element belongs to is updated (see **Figure 3**). In the figure, when considering the edge formed by the use of the vertices indexed at 5 and 8, the disjoint-set data structure updates which set those vertices are a part of in real-time. If the edge is included in the final Hamiltonian Path, the integer array is updated to reflect an association between the two vertices via the edge that forms from their connection. Shown in **Figure 3** is how the integer at array index 8 is updated to *5*, meaning that vertices 5 and 8 are a part of the same set. Likewise, should the next edge to be considered consist of vertices 4 and 5, the inclusion of that edge in the final Hamiltonian Path means that vertex 4 is now a part of the set of associated vertices identified by the value *5*. Over the course of the algorithm's execution, vertex sets are updated in real time, and the benefit of this data structure is that edges whose inclusion

would result in the formation of a premature cycle (because the vertices in that edge are already a part of the same set) are discarded, allowing other potential edges to be included. In this manner, all vertices become associated with a particular set that distinguishes one connected set of edges from another. Furthermore, when two edge "sets" are connected via the inclusion of an edge—that is, the edge in consideration has vertices that are already a part of two differently-identified sets—all vertices' set identifiers (the value in the array at those vertices' indices) are updated to reflect the new set, keeping a consistent record of all edge-segments as they form in the graph. This update is known as path compression, and depending on the application, may or may not be a part of the implementation (it is used in this implementation). This, thusly, prevents the formation of a cycle amongst the vertices, keeping the algorithm intact and valid.

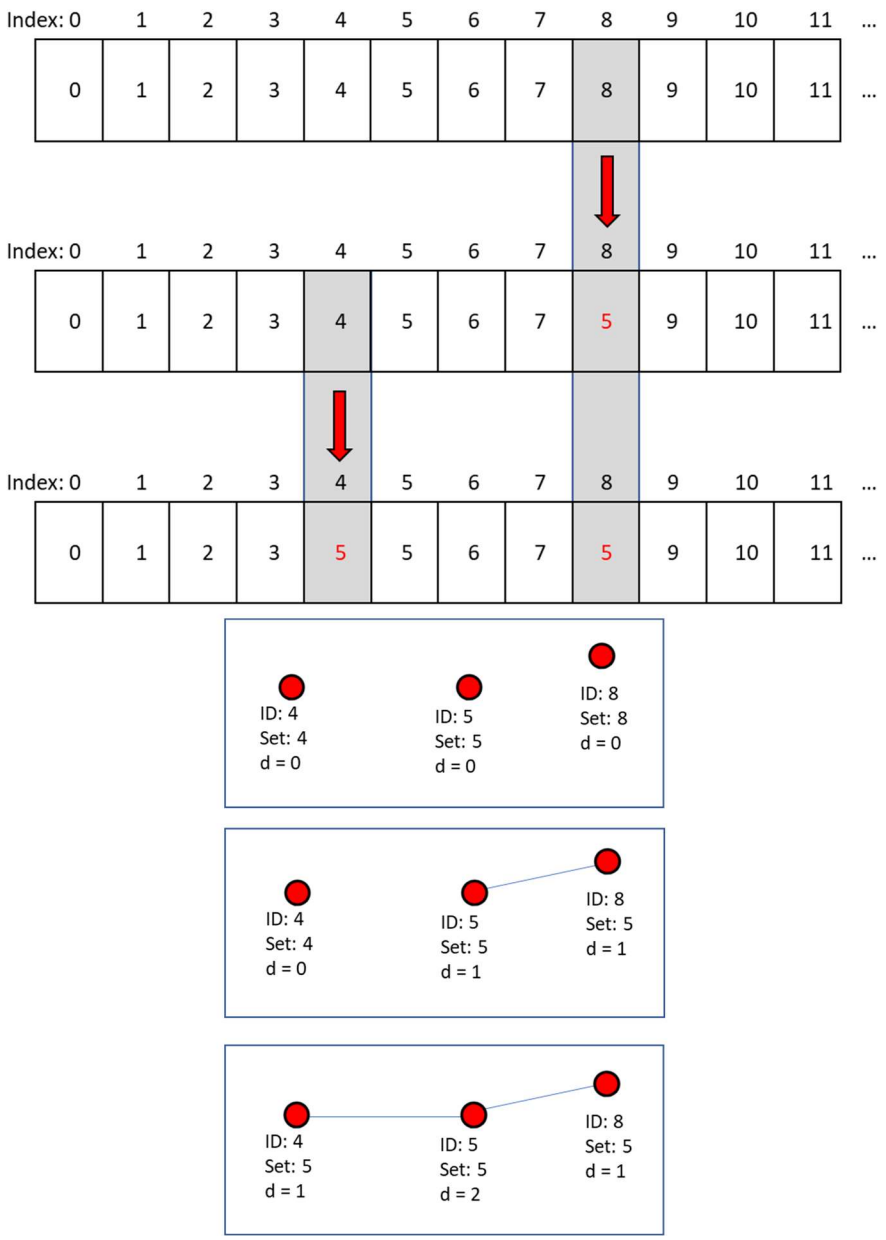

Figure 3 – Disjoint-Set Data Structure. How the elements in a Union-Find are changed to reflect a vertex's association to other vertices. Shown in the top half is the manner in which the data structure is represented in the array. The bottom half visualizes what is happening in the graph as edges are joined.

The following paragraphs discuss each of the major steps in more detail.

File Reading and Vertex Instantiation

The implementation begins by reading a TSP file as input. TSP graphs are taken from TSPLIB [7], which provides pre-calculated optimal solutions for comparison. Standard with every TSP file format is a handful of introductory lines detailing the list of vertices and the kind of graph associated with the file. Vertex identification numbers, and their associated coordinates, are listed in sequential order following the details of the TSP graph. The coordinates are stored as floats in a struct containing two values, one for each coordinate *x* and *y*. Vertices are identified in the code according to their array index.

Edge Instantiation

The stored vertices are then used to instantiate all edges in the implementation. This was initially accomplished serially via the use of two for-loops that traversed all other vertices for a given vertex. Due to loop-carried dependency, this form of instantiation was not directly parallelizable. The parallel implementation, however, makes use of a closed-form single for-loop that assigns Edge vertices based on constants. This closed-form implementation was parallelized via OpenMP, utilizing its specific parallel-for preprocessing directive. The length of each edge was not necessary to store, as the length is recalculated when needed (i.e. during sorting or tour length calculation). For the GPU implementation, a kernel that launched one thread per edge

was used to simultaneously instantiate all edges in a graph, assigning each edge instance with both of its designated vertices.

Implementation of the Main Loop

The serial and parallel CPU implementation's main loop of the chosen heuristic consists of four segments—preprocessing, processing, sorting, and inclusion—each of which are described below in detail.

Preprocessing. The standard library's std::partition is called to rearrange the currently-processing set of edges in such a way that all edges that meet the criteria for inclusion in the final tour are moved to the front half of the set. Recall that the criteria for inclusion in the final tour is dependent on two stipulations: (1) if both vertices associated with the edge in question have a degree of less than 2, and (2) if both vertices have a degree of 1, they must not be a part of the same set of connected edges (otherwise, the inclusion of the edge in question would result in a cycle, meaning the final path would be invalid). This preprocessing step aids the processing segment in that only edges that are deemed eligible to be included in the final tour are sorted, eliminating the need to sort edges that will ultimately be excluded.

Processing and Sorting. As part of the implementation's sorting, std::partition is used to implement an emulated recursive quicksort, and during the processing portion of the main loop, the currently-processing set of edges is sorted in a manner that is

consistent with the way a standard quicksort is implemented. A pivot edge is chosen, and based on its weight (in addition to its vertex IDs), all other edges are either placed preceding or following that particular edge via that call to std::partition. Additional calls to quicksort are then recursively stacked, calling std::partition on the edges that were placed prior-to and after the previous pivot edge. In other words, a recursive call to quicksort is made with the subranges specified as "low" to "pivotEdge – 1" and "pivotEdge + 1" to "high". To find the edge index at which the call to std::partition was completed (the pivot point in separating the edges), the standard library's std::find is called to search in the range of currently-processing list of edges. This index is returned and used in subsequent recursive calls to quicksort (and therefore std::partition).

"Chunked" Sorting. As a means of optimization for the traversal of the list of edges that need to be sorted, a "chunk size" was introduced to the algorithm. This means that the algorithm can dynamically account for how many edges to process at a time, potentially reducing the program's runtime. If a value of 1024 is supplied at runtime, 1024 edges will be iterated through before moving onto the next 1024 edges. The use of this "chunk size" introduces the opportunity to save time from the algorithm's total time to execute. This is especially true in the first iteration of edge traversal, where the entire list of edges would need to be sorted. In using this "chunk size" parameter, only a certain number of edges need be rearranged at a time. Total runtime does vary based on what chunk size is chosen at runtime, and correlations between chunk size and final runtime have yet to be analyzed, but a chunk size of 1024

has been used as a consistent runtime comparison between iterations of the algorithm implementation. Edges that are excluded from processing due to the chunk size are included in a list that is then passed to the main loop's next iteration.

Main Loop Edge Iteration. The final portion of the main loop consists of iterating over the edges that have been preprocessed, processed, and sorted. This means that as edges are added to the final tour, their respective vertices are updated in the Union Find data structure, their respective vertex degrees are updated (which will be used to determine an edge's inclusion in the preprocessing step), and the counter for included edges is incremented. Additionally, an array that keeps track of what cities are connected via the included edges is updated—this array is recalled during the city-order reconstruction step after the main loop. The edge iteration loop terminates either after the current chunk size is reached or after the last edge is included in the tour (this is known via a counter that keeps track of how many edges have been included in the final tour; the tour is complete once $n - 1$ edges have been included in the final tour, where n is the number of vertices in the graph).

GPU Main Loop Implementation

The GPU implementation is similar to the CPU's parallel implementation with the exception of the Main Loop's Edge Iteration; std::partition has been replaced with cub::DevicePartition and std::find has been replaced with a CUDA kernel that finds the pivot edge and swaps it for the first edge that returns false for the call to partition.

What may not be immediately clear is that the range of partitioned edges that reside above the pivot edge (that is, the partition's condition that returns false for edges that are larger than the pivot) does not necessarily contain the pivot edge as the first "false" element. Therefore, the pivot edge must be found in the range of false-partitioned edges and swapped for the first edge for the quicksort implementation to be carried out correctly.

The GPU's implementation of the Main Loop's Edge Iteration segment, though similar in approach, can be observed to have characteristics that would inherently occur faster than the serial iteration of every edge in the chunk (see **Figure 4**). When the GPU is evaluating a chunk of edges (e.g. edges #1024 - #2048 that would comprise the second chunk when a chunk size of 1024 is chosen), a keen observation can be made about the vertices of each of those edges: there is a distinction between edges according to their vertices, and edges that contain unique vertices in the chunk have no bearing on other edges with different vertices. In other words, when considering to include an edge in the final tour or not, an edge with vertex IDs of 1 and 2 do not exert an influence on whether the edge with vertices 3 and 4 is included or excluded. Furthermore, it can be determined whether an edge will be included or excluded based purely on the vertices' degrees. For example, if no edge containing vertices 5 and 6 has been included in the tour, and the current edge under consideration for inclusion in the tour is the edge with vertices 5 and 6, it can be safely concluded that this edge can be included in the tour because the list of edges is sorted and, therefore, is the shortest edge containing those two vertices. Including the edge with vertices 5 and 6 will

consequently increase their degree by 1, meaning that the next time an edge contains either of those vertices is considered alongside whichever vertex is located at the other end of each of those edges.

Bearing the aforementioned in mind, the example in **Figure 4** can now be traced. During the GPU's main loop initialization, every edge is given a designation as "Ready" or "Done", and this designation depends on the same criteria mentioned before: (1) if either of the edge's vertices have a degree of greater than 1, the edge is designated as "Done", and likewise (2) if both of the edge's vertices are already a part of the same set (from the Union Find data structure), the edge is also designated as "Done". In the figure, the edges marked as "Done" are colored red, meaning they have been excluded from the tour based on the fact that those vertices have already been used, and this leaves a certain set of edges to be considered by the next step. As mentioned previously, edges that contain unique vertices in the chunk have no influence from other edges that also utilize those same vertices. It can be seen that vertex 0 appears in edge 3 and 5. Similarly, vertex 2 makes an appearance in edges 3 and 7, vertex 4 first makes an appearance in edge 4, and vertex 5 first makes an appearance on edges 4 and 5. These facts mean that at this point in the algorithm, vertices 0, 2, 4, and 5, are eligible for inclusion in the tour, and therefore, their respective edges can be included in the tour. Edges 5, 7, and others that contain vertices that have already made appearance in previous, eligible edges must wait an iteration in order to correctly determine whether or not they are eligible for inclusion (and this is the main dependency of the heuristic that is taken advantage of in relation to the serial

implementation). After those edges have been included in the tour, their state is designated at "Done", and are excluded from further iterations of the chunk. The subsequent iterations of the chunk take into account previously included edges, and are instead considered for inclusion in the tour based on the new set of circumstances on a vertex's degree and set. This iteration of the main loop repeats until all edges have been designated as "Done" (colored in black in the figure), and the implementation can then proceed to analyze the next chunk of edges (after they are preprocessed and sorted as denoted in the heuristic's outline).

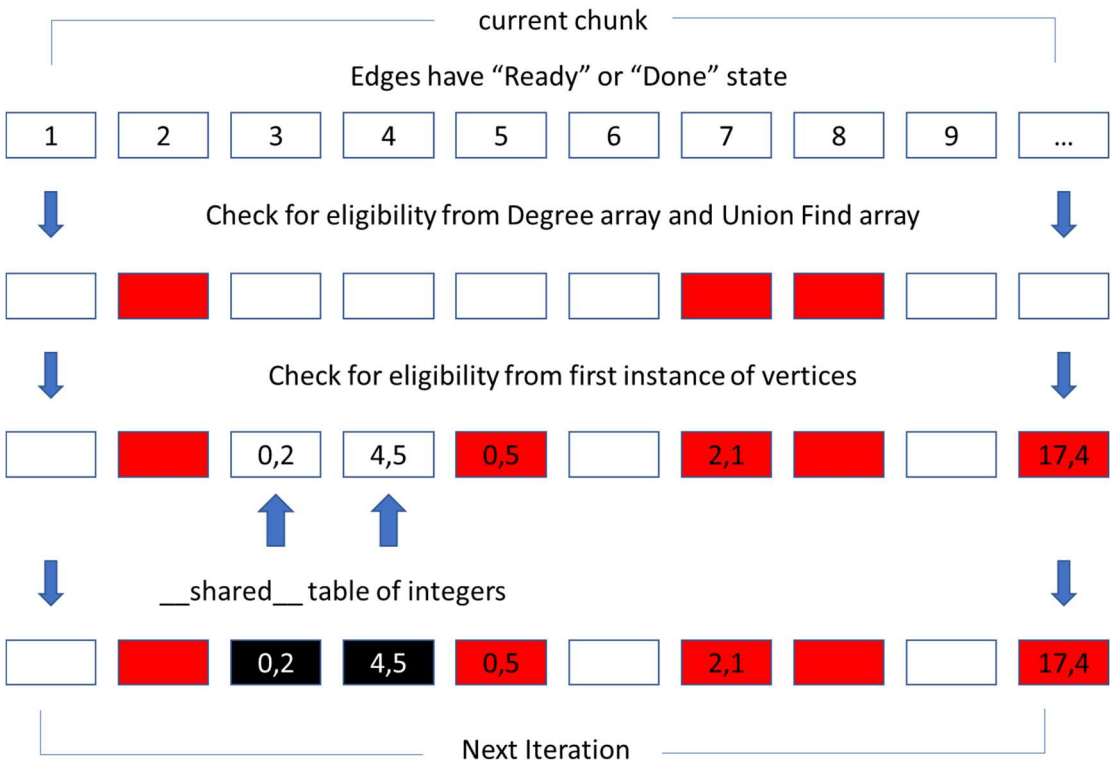

Figure 4 – GPU Main Loop Implementation. Shown above is an example of how the GPU's main loop kernel carries out the edge inclusion/exclusion process of the heuristic.

City-Order Reconstruction

Once the final tour is complete, the implementation reconstructs the city-visit order. This is accomplished by first finding one of the two vertices that have a degree of 1. This means that that particular vertex is the terminal-vertex at one end of the connected-set of edges that is the tour. From there, the array used to keep track of what cities are connected to each vertex is used to construct an array of vertices that is the visit-order. This is accomplished by the same means on the GPU for the parallel implementation due to the inherent nature of the data structure. Although a truly parallel approach could be derived from the portion of the heuristic that reconstructs the city-visit-order, the amount of time the heuristic spends reconstructing the order is negligible.

Calculating the Tour's Length

After the heuristic completes, the tour's length is found by re-calculating and totaling the distances between the vertices provided by the city-visit order array. This is the value that is ultimately compared to the data provided by TSPLIB in order to analyze solution quality. This step is also carried out identically for both the CPU and GPU versions of the heuristic.

IV. METHODOLOGY

Parallelism in the CPU implementation has been introduced by calls to std::partition and std::find as well as the use of OpenMP for the edge instantiation portion. The C++ standard library provides for an overloaded function call to both std::partition and std::find, where an execution policy can be defined for those functions. The execution policy supplied to the function can dictate whether the calls to partition and find will be executed serially or in parallel, and to decide whether a call should be made serially or in parallel, a threshold is used. If the number of edges to preprocess or sort exceeds this threshold, the call is carried out in parallel with the standard library's std::execution::par_unseq execution policy. Alternatively, if the number of edges to preprocess or sort falls below this threshold, the call to std::partition is carried out serially. The threshold was defined as 512 * 512, where various execution times were measured for optimization (data not shown).

Results for this algorithm are interpreted through two primary points: (1) the final path length calculated and (2) the time taken for the program to run. The time taken for the program to run is measured as the best run time achieved, and multiple runs are necessary to account for other possible processing tasks in the background, eliminating the potential for inconsistent times. Performance of the implemented heuristic is measured based on both the amount of time taken to complete as well as the heuristic's final path length relative to the optimal. Measuring these two factors will allow for an effectiveness comparison between the TSPLIB optimal solutions as well as other, related heuristics' implementations. The greedy heuristic's implementation

makes use of timestamps for measuring how long a particular section in the heuristic is taking during the program's execution. All execution times are measured in seconds rounded to the fourth decimal place, with execution times comprised of the Setup time, Main Loop Time (which consists of the List Processing, Preprocessing, and Sorting sections), City Order Build Time, and Deletion time. Although the heuristic's individual segments were timed (data not shown), the metric primarily used for comparison was the overall runtime, as other work can only be compared according to the total time it takes to execute (rather than individual segments that are not directly comparable).

The current system "Ithaca" runs Fedora 30 (Server Edition), kernel version Linux 5.6.13-100.fc30.x86_64 with a 64-bit architecture. The additional specifications are as follows:

- CPU: AMD Ryzen Threadripper 2950X 16-Core Processor with 2 threads per core (32 total CPU logical cores) 2149.175 MHz processor with boost speed of up to 3500 MHz.
- L1d cache: 32K
- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 8192K
- 48 GB of memory with 2 non-uniform memory architecture nodes (logical cores 0 – 15 on node 0 and logical cores 16-31 on node 1)

All GPU comparisons in this study were performed on an NVIDIA GV100 TITAN V. The TITAN V specifications are as follows:

- 12 GB of global memory
- 96 KB per SM L1 cache
- 4.5 MB L2 cache
- “Volta” architecture
- Base clock speed: 1.2 GHz with boosting up to 1.455 GHz
- 5,120 CUDA cores

The CPU implementations were compiled using the GNU C++ compiler “g++” version 9.3.1, along with optimization option level 3 “-O3”. Flags “-march=native”, “-std=c++17”, “-fopenmp”, and “-ltbb” are also used in compilation to support instruction optimization flags that are specific Ithaca’s hardware, use the 2017 C++ standard, enable the usage of openMP, and enable the Threading Building Blocks C++ template library, respectively. The CUDA code was compiled on Ithaca using the CUDA Toolkit v11.0.167 with the NVCC compiler [28] build version cuda_11.0 with the flags “-std=c++17”, “-arch=sm_35”, “-O3”, and “-ltbb”.

Inputs analyzed were taken from TSPLIB’s collection of TSP graphs [7, 29], and final outputs of the computed path lengths and calculation times were compared to TSPLIB’s optimal solutions. The TSPLIB graphs serve as a basis for reference comparison, as available optimal solutions have been proven mathematically.

The program’s times of execution were measured via timestamps for each section. However, only total execution time is comparable to other heuristics. Also, the

final path length calculated from the heuristic was compared to that of TSPLIB's optimal solutions. Final path length is calculated after the algorithm's termination and is recalculated as part of the returned city-visit order array during a loop that references every vertex's coordinates from each edge.

Metrics

Two metrics were used for the comparison of the heuristic's performance to other heuristics: solution quality and execution time. Solution quality is defined as the final path length calculated for the constructed tour. This final path length is then compared to the optimal length as designated by TSPLIB, and a measurement of percent difference is taken into account. In a similar fashion, execution times are measured in terms of seconds. How long the algorithm took was compared to how other implementations fared in their execution times.

V. RESULTS

CPU Serial and CPU Parallel Comparison

Direct comparison between the serial CPU and parallel CPU implementations revealed that the parallelization had an overall positive effect on the program's runtimes. The GPU implementation also exhibited similar improvements over the parallel CPU implementation. See **Table 1** (faster runtimes have been bolded) for a comparison of the serial CPU, parallel CPU, and GPU implementations and **Figure 5** for a comparison of the smallest and largest eight TSPLIB graphs.

The GPU implementation was able to achieve a best speedup of 8 over the serial CPU implementation (see graph rl11849.tsp). However, all other graph speedups remained below a speedup of 4. Surprisingly, a few graphs, though faster on the GPU over the serial implementation, had their runtimes execute the fastest on the parallel CPU implementation (see graphs pcb3038.tsp, fl3795.tsp, and rl5915.tsp).

Solution quality, measured as a percent difference from TSPLIB's reported optimal path length, remained below 18%, with the best solution quality presenting for graph ts225.tsp at 2.54%. There is no clear correlation between the number of nodes a graph has and the solution quality achieved by the heuristic. Similarly, there does not appear to be a correlation between the number of nodes in a graph and the execution time, though generally speaking, runtime does tend to increase as the number of nodes increases.

The heuristic was able to achieve a throughput of 668,000 nodes per second (see graph pr1002.tsp)

Table 1 – **CPU and GPU Total Runtime Comparison.** Shown below are each graph's results for each run of the serial CPU, parallel CPU, and GPU implementations. Percent difference is a measure of how far the solution quality is from the optimal path. Runtimes were measured in seconds (best of 10 runs), the fastest of which has been bolded. Due to hardware constraints, the largest of the TSPLIB graphs could not be run on the GPU. Additionally, the speedup gained by GPU execution over the serial CPU implementation is shown.

Filename	Nodes	% Difference from Optimal	CPU Runtime (Serial)	CPU Runtime (Parallel)	GPU Runtime	GPU Speedup over CPU Serial
kroE100.tsp	100	5.80%	0.0013	0.0008	**0.0004**	3.25
ts225.tsp	225	2.54%	0.0016	0.0016	**0.0005**	3.20
rat575.tsp	575	10.31%	0.0023	0.0023	**0.0014**	1.64
pr1002.tsp	1002	12.90%	0.0050	0.0068	**0.0015**	3.33
vm1084.tsp	1084	11.31%	0.0060	0.0068	**0.0030**	2.00
pcb1173.tsp	1173	15.54%	0.0049	0.0058	**0.0028**	1.75
d1291.tsp	1291	8.75%	0.0061	0.0060	**0.0029**	2.10
rl1304.tsp	1304	10.02%	0.0059	0.0066	**0.0028**	2.11
rl1323.tsp	1323	9.06%	0.0072	0.0088	**0.0043**	1.67
fl1400.tsp	1400	9.20%	0.0070	0.0108	**0.0034**	2.06
u1432.tsp	1432	15.41%	0.0085	0.0082	**0.0051**	1.67
fl1577.tsp	1577	5.60%	0.0075	0.0071	**0.0037**	2.03
vm1748.tsp	1748	13.54%	0.0204	0.0111	**0.0092**	2.22
u1817.tsp	1817	10.71%	0.0091	0.0079	**0.0051**	1.78
rl1889.tsp	1889	14.45%	0.0110	0.0082	**0.0042**	2.62
d2103.tsp	2103	9.14%	0.0130	0.0112	**0.0079**	1.65
u2152.tsp	2152	16.39%	0.0119	0.0094	**0.0063**	1.89
u2319.tsp	2319	9.59%	0.0139	0.0116	**0.0076**	1.83
pr2392.tsp	2392	16.14%	0.0175	0.0168	**0.0061**	2.87
pcb3038.tsp	3038	17.37%	0.0275	**0.0216**	0.0249	1.10
fl3795.tsp	3795	6.90%	0.0369	**0.0235**	0.0362	1.02
fnl4461.tsp	4461	14.51%	0.0380	0.0183	**0.0105**	3.62
rl5915.tsp	5915	9.91%	0.0769	**0.0364**	0.0399	1.93
rl5934.tsp	5934	12.23%	0.0693	0.0284	**0.0204**	3.40
pla7397.tsp	7397	12.84%	0.0909	0.0402	**0.0294**	3.09
rl11849.tsp	11849	11.82%	0.4804	0.1301	**0.0600**	8.01
usa13509.tsp	13509	14.82%	0.4254	0.1420		
brd14051.tsp	14051	13.90%	0.7722	0.2409		
d15112.tsp	15112	14.55%	0.9673	0.2444		
d18512.tsp	18512	14.05%	1.5961	0.4240		
pla33810.tsp	33810	12.41%	2.4651	1.3114		

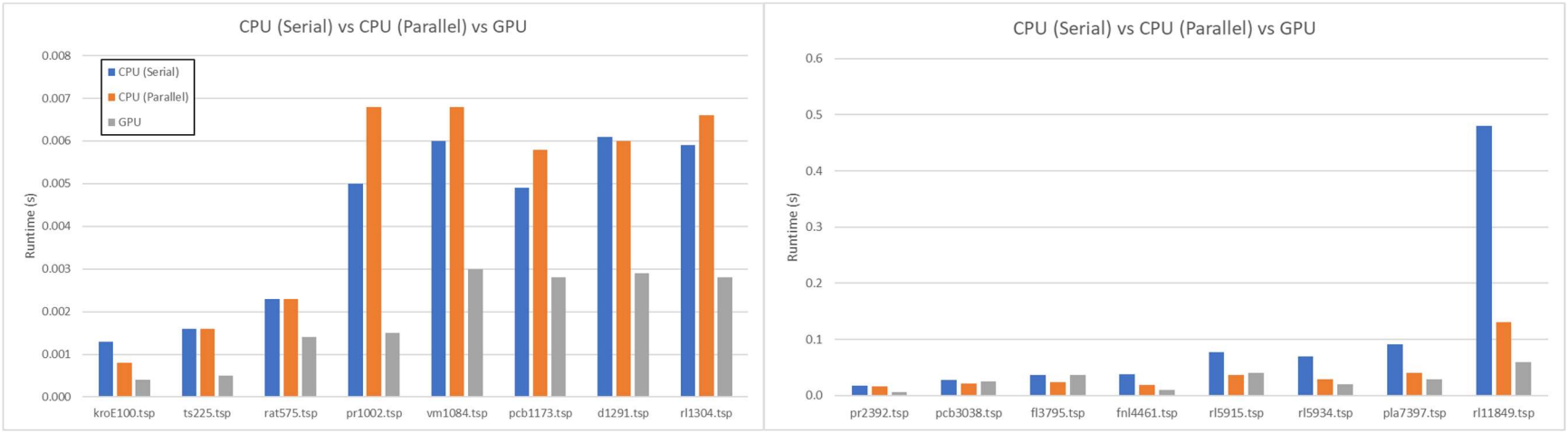

Figure 5 – CPU and GPU Runtime Comparison of the Smallest and Largest Graphs. Shown on the left is the runtime comparison of the 8 smallest TSPLIB graphs. The parallel CPU implementation can be seen to not consistently improve up on the runtime when compared to the serial implementation. The GPU implementation, however, consistently outperformed the serial and parallel CPU implementations. Shown on the right is the runtime comparison of the largest 8 graphs. It can be seen that the GPU implementation did not always achieve the best performance. The largest graph, however, did directly benefit from CPU and GPU parallelization.

IHC 2-Opt Comparison

The following data is the greedy heuristic's comparison to Burtscher and O'Neil's [4] previous implementation of the IHC 2-Opt TSP heuristic. Recall that the IHC 2-Opt is a combination of constructive (IHC) and improvement (2-Opt) heuristics. The path is initially constructed using the random heuristic and is improved upon by the 2-Opt heuristic until it finds a local minimum. This minimum is then compared with however many other IHC runs were initiated during the program run. **Table 2** and **Figure 6** outline the data obtained. For **Table 2**, the optimal path listed by TSPLIB is notated alongside the graph name, with the greedy heuristic's runtime and calculated final path length noted directly below it (shown as row "v9.8"). The lower columns denote the runtime and path length achieved by Burtscher and O'Neil's IHC 2-Opt. The column on the left denotes how many restarts were used and, therefore, the runtime and final length acquired for that run. Blacked out cells indicate that the run could not be executed due to hardware constraints. **Figure 6** visualizes the tabulated data. Graphs kroE100.tsp and ts225.tsp were able to achieve the optimal result for restarts of 65,536 and upwards and 262,144, respectively. No other instance of IHC 2-Opt was able to achieve optimal result. Runtime for the greedy heuristic was far superior to the IHC 2-Opt, running even the largest graph of 11,849 nodes in under approximately 0.06 seconds. Recall that this performance may come at a tradeoff of solution quality. **Figure 6** aids in visualizing the observation that the greedy heuristic outperforms the IHC 2-Opt in both solution quality and runtime for graphs d2102, rl5915, and rl11849. Depending on the number of restarts used, the rest of the graphs varied in solution quality for IHC 2-Opt, not

achieving better solution quality for a lower number of restarts, but surpassing the greedy heuristic for larger numbers of restarts.

Table 2 – IHC 2-Opt comparison to the greedy heuristic. The optimal path listed by TSPLIB is notated alongside the graph name, with the greedy heuristic's runtime and calculated final path length noted directly below it (shown as row "v9.8"). The lower columns denote the runtime and path length achieved by Burtscher and O'Neil's IHC 2-Opt. The column on the left denotes how many restarts were used and, therefore, the runtime and final length acquired for that run. Blacked out cells indicate that the run could not be executed due to hardware constraints.

	kroE100	22068	**ts225**	126643	**rat575**	6773	**pr1002**	259045	**vm1084**	239297	**pcb1173**	56892	**d1291**	50801
v9.8	0.0004	23349	0.0005	129854	0.0014	7471	0.0015	292449	0.003	266350	0.0028	65735	0.0029	55247
Restarts	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**
2	0.0012	23354	0.0077	131745	0.0918	7555	0.4426	283470	0.5742	262952	0.6817	64334	0.9387	56969
4	0.0012	22791	0.0077	131745	0.0937	7412	0.4413	282663	0.5825	259129	0.6887	62250	0.9389	56969
8	0.0012	22791	0.0077	128353	0.0937	7412	0.4502	282286	0.5895	258470	0.6894	62250	0.9411	56781
16	0.0012	22791	0.0078	128353	0.0937	7412	0.4507	282227	0.5894	258470	0.6925	62250	0.9533	55793
32	0.0012	22791	0.0078	128353	0.0937	7370	0.4508	281379	0.5885	258470	0.6967	62250	0.9539	55793
64	0.0012	22521	0.0080	128353	0.0953	7370	0.4499	278597	0.5870	255196	0.6965	62250	0.9604	55793
128	0.0013	22470	0.0087	127893	0.1033	7338	0.4992	279147	0.6548	255196	0.7759	62170	1.0700	55273
256	0.0014	22470	0.0102	127463	0.1296	7235	0.6544	276953	0.8539	253727	1.0174	62484	1.4218	54914
512	0.0018	22359	0.0143	127536	0.1904	7335	1.0167	275999	1.3374	253727	1.5850	62407	2.2062	55204
1024	0.0026	22359	0.0243	127296	0.3342	7231	1.8288	275999	2.4095	253022	2.8751	62472	4.0181	54511
2048	0.0054	22307	0.0538	127296	0.7402	7294	4.1276	275999	5.2987	254290	6.3140	62047	9.1440	54702
4096	0.0100	22117	0.1003	127396	1.3390	7289	7.6762	275999	10.0108	251262	11.7744	62146	16.7200	54411
8192	0.0189	22117	0.1902	126880	2.6501	7277	14.6537	275281	19.5846	253262	23.3522	61721	32.4649	54150
16384	0.0370	22117	0.3530	126809	5.0193	7273	28.4877	274936	37.7007	251190	45.0404	61915	66.1463	54386
32768	0.0736	22100	0.6953	126873	10.0116	7250	56.7630	274778	78.2803	251190	94.4421	61739	135.8891	54122
65536	0.1449	22068	1.3568	126783	19.8261	7235	118.7882	274931	158.3493	252024	188.9999	61751	270.2249	54107
131072	0.2752	22068	2.6960	126783	39.5461	7234	240.2684	272296	317.6040	251485	378.8874	61712	540.3628	53922
262144	0.5353	22068	5.3683	126643	79.0569	7247								
524288	1.0599	22068	10.7160	126726										
1048576	2.1070	22068												

Table 2 (cont'd) – IHC 2-Opt comparison to the greedy heuristic. The optimal path listed by TSPLIB is notated alongside the graph name, with the greedy heuristic's runtime and calculated final path length noted directly below it (shown as row "v9.8"). The lower columns denote the runtime and path length achieved by Burtscher and O'Neil's IHC 2-Opt. The column on the left denotes how many restarts were used and, therefore, the runtime and final length acquired for that run. Blacked out cells indicate that the run could not be executed due to hardware constraints.

	rl1304	252948	**rl1323**	270199	**fl1400**	20127	**u1432**	152970	**fl1577**	22249	**vm1748**	382112
v9.8	0.0028	278302	0.0043	294680	0.0034	21978	0.0051	176549	0.0037	23494	0.0092	336556
Restarts	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**
2	0.9967	283163	1.0381	297170	1.0911	21244	1.1182	171713	1.6402	25020	2.3402	366627
4	1.0165	282040	1.0438	293227	1.0915	21244	1.1186	171236	1.6720	23851	2.3417	365116
8	1.0203	279097	1.0568	293227	1.0921	20998	1.1323	171236	1.6769	23851	2.3375	365116
16	1.0208	275899	1.0560	292838	1.0943	20871	1.1268	169959	1.6967	23851	2.3388	365116
32	1.0206	273993	1.0571	291953	1.0983	20863	1.1456	167253	1.6876	23851	2.3507	364664
64	1.0245	269484	1.0568	291953	1.1046	20820	1.1458	167253	1.6974	23797	2.3594	363404
128	1.1452	269484	1.1840	289689	1.2322	20692	1.2865	168991	1.8936	23643	2.6294	363404
256	1.5253	269484	1.5597	289689	1.6315	20786	1.7110	168139	2.5308	23555	3.5449	362379
512	2.3687	269450	2.4928	289647	2.5640	20664	2.6546	168161	3.9868	23531	5.6056	361968
1024	4.2852	269450	4.4485	289359	4.6764	20634	4.8375	168264	7.2330	23381	10.1940	360975
2048	10.0497	268573	10.1326	288779	10.2943	20657	10.6175	167103	16.5294	23360	23.2903	360222
4096	18.1324	268988	18.5393	286290	19.1878	20627	20.4914	166710	30.2735	23389	42.8607	360043
8192	35.2735	267095	36.6249	286290	37.8876	20559	38.8256	166625	61.5420	23263	87.6391	360957
16384	68.5698	266978	70.8741	286732	77.1384	20491						
32768	141.8876	266787										
65536	283.8886	266551										
131072	566.1644	265854										

Table 2 (cont'd) – IHC 2-Opt comparison to the greedy heuristic. The optimal path listed by TSPLIB is notated alongside the graph name, with the greedy heuristic's runtime and calculated final path length noted directly below it (shown as row "v9.8"). The lower columns denote the runtime and path length achieved by Burtscher and O'Neil's IHC 2-Opt. The column on the left denotes how many restarts were used and, therefore, the runtime and final length acquired for that run. Blacked out cells indicate that the run could not be executed due to hardware constraints.

	u1817	57201	**rl1889**	316536	**d2103**	80450	**u2152**	64253	**u2319**	234256	**pr2392**	378032	**pcb3038**	137694
v9.8	0.0051	63329	0.0042	362289	0.0079	87800	0.0063	74784	0.0076	256713	0.0061	439056	0.0249	161615
Restarts	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**
2	2.4678	65443	2.9539	348149	3.8057	93151	4.0304	72831	4.4010	252451	5.6702	422800	11.2115	154678
4	2.4678	65443	2.9638	344845	3.8031	93151	4.0301	72831	4.3980	251720	5.6895	421718	11.2959	153703
8	2.4750	64932	2.9997	344845	3.8105	92276	4.0277	72831	4.3949	251681	5.6927	420107	11.2960	153703
16	2.5199	64643	2.9876	344845	3.8757	92276	4.0296	72831	4.4161	250304	5.7332	420107	11.2982	153703
32	2.5209	64557	3.0039	344845	3.8615	92038	4.0503	72649	4.4316	250304	5.7353	419362	11.3590	152730
64	2.5071	64204	3.0009	344845	3.8599	90785	4.0663	72595	4.4324	250304	5.7593	418056	11.3586	152730
128	2.7860	64000	3.3702	344350	4.3116	91030	4.5608	72547	5.0287	250062	6.4628	418453	12.8792	153326
256	3.7395	64117	4.5499	342594	5.8415	91181	6.1716	72216	6.7983	249403	8.7932	416769	17.4464	153091
512	5.9269	63979	7.1670	340932	9.2372	90068	9.7807	72114	10.7267	249358	13.8490	414940	27.6220	152674
1024	10.7718	63665	13.0898	341252	16.8716	90829	17.7932	72032	19.5798	249415	25.3184	414937	50.9162	152561
2048	23.7896	63554	30.9299	340315	37.3678	89988	40.6762	72069	44.8716	249630	58.9094	415839	121.7172	152781
4096	45.9845	63403	55.5633	337373	75.8613	89747	77.9760	71851	85.0558	248770			231.5837	151806
8192	93.3413	63407			148.2330	89999								
16384	184.7253	63470			289.4093	89919								
32768	369.8695	63031			578.3940	89552								
65536					1144.0993	89378								

Table 2 (cont'd) – IHC 2-Opt comparison to the greedy heuristic. The optimal path listed by TSPLIB is notated alongside the graph name, with the greedy heuristic's runtime and calculated final path length noted directly below it (shown as row "v9.8"). The lower columns denote the runtime and path length achieved by Burtscher and O'Neil's IHC 2-Opt. The column on the left denotes how many restarts were used and, therefore, the runtime and final length acquired for that run. Blacked out cells indicate that the run could not be executed due to hardware constraints.

	fl3795	28772	**fnl4461**	182566	**rl5915**	565530	**rl5934**	556045	**pla7397**	23260728	**rl11849**	923288
v9.8	0.0362	30757	0.0105	209061	0.0399	621577	0.0204	624046	0.0294	26248277	0.06	1032389
Restarts	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**	**Runtime**	**Length**
2	22.4951	32085	35.1204	202505	92.4223	639616	91.9893	628726	165.1220	25827828	741.6745	1051691
4	22.8023	31204	35.1175	202505	92.8680	638001	92.5841	625091	166.4640	25698456	742.4362	1051691
8	22.7973	31204	35.1697	202505	92.8765	636543	92.5926	623799	167.4434	25641951	744.2086	1042867
16	22.8029	30548	35.6342	202505	92.8655	632066	93.3550	622591	167.5170	25544020	747.9204	1042867
32	22.8648	30548	35.6362	202171	92.8397	632066	93.5277	622591	168.3295	25534893	747.8780	1042867
64	22.8910	30548	35.6346	202171	93.0942	632066	93.5107	622591	168.2361	25534893	763.7597	1042867
128	26.0922	30497	40.8956	202127	107.0453	630126	107.6986	624241	194.1727	25560558	857.8493	1042447
256	35.3507	30612	55.0980	202176	143.6880	631865	144.2281	620152	257.6325	25484836	1140.3858	1038796
512	56.0552	30424	87.5686	201143	232.3392	628919	234.3094	620448	432.0093	25463609	1889.1486	1039033
1024	108.6311	30127	168.4478	201392	439.6970	629912	443.0789	621075			3544.7993	1039334
2048			387.2900	201617	1036.2319	627612						
4096			733.3511	201654	1847.6337	626669						
8192					3640.1165	627325						
16384					7056.9592	625011						

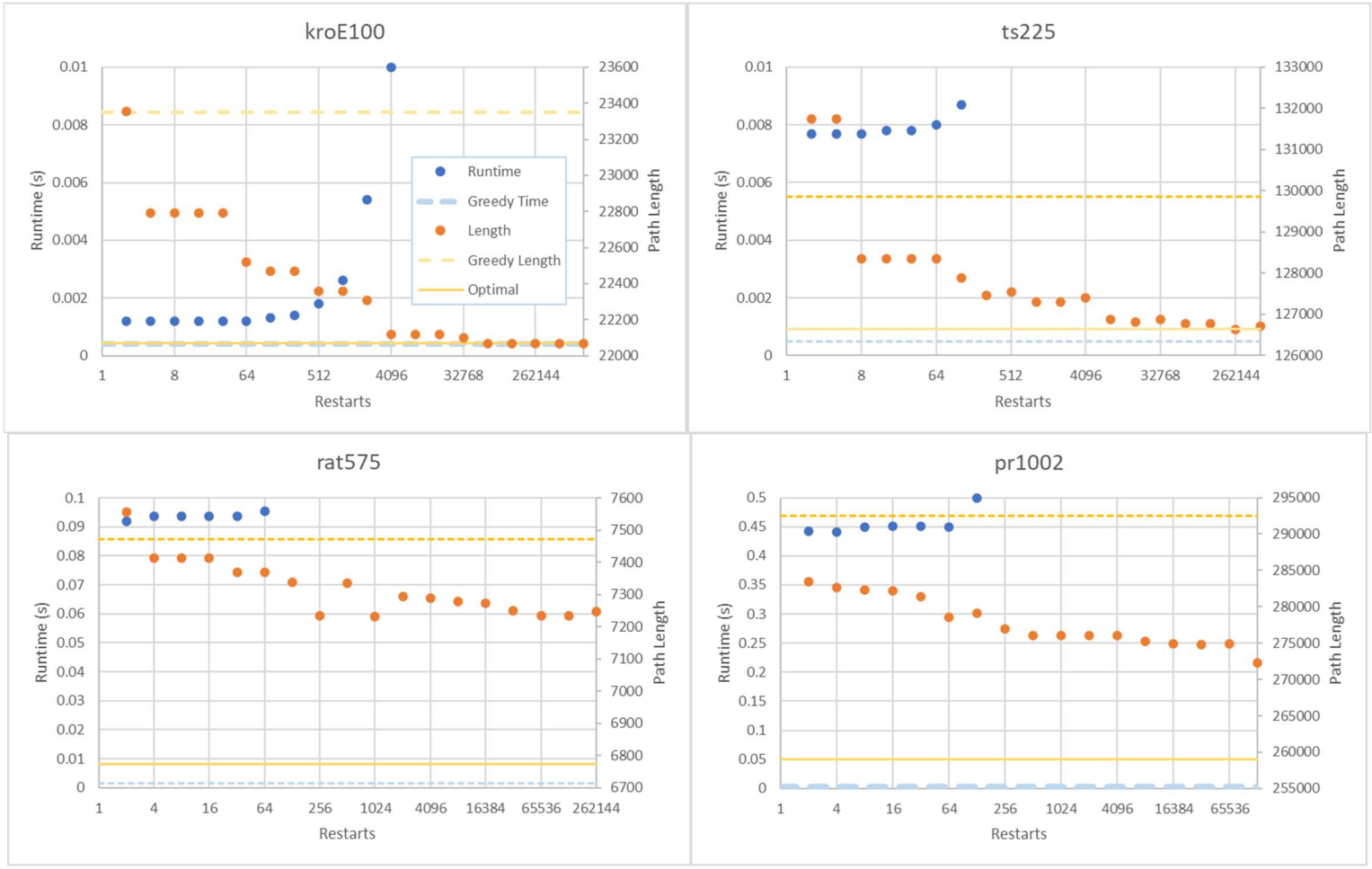

Figure 6 – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

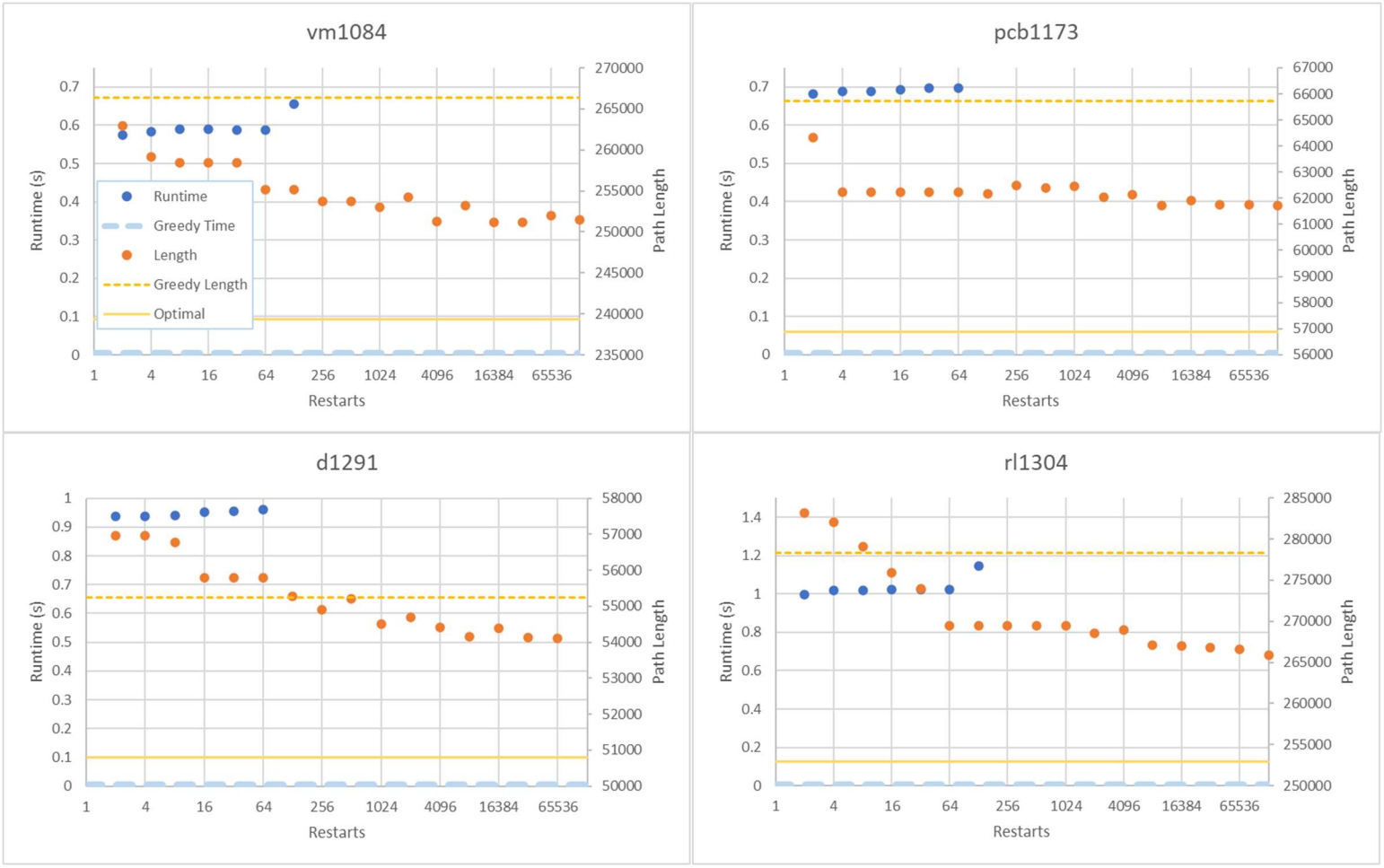

Figure 6 (cont'd) – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

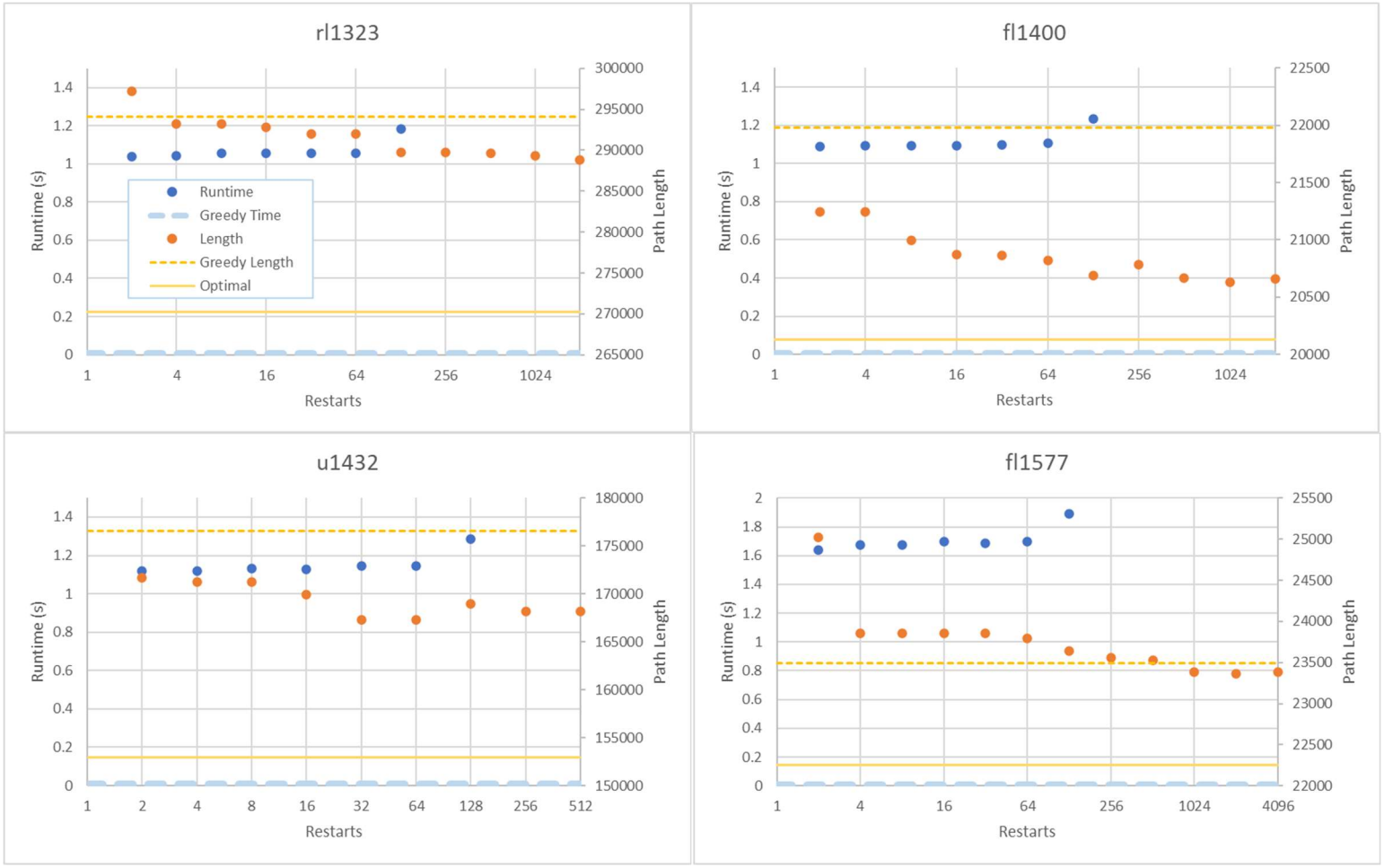

Figure 6 (cont'd) – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

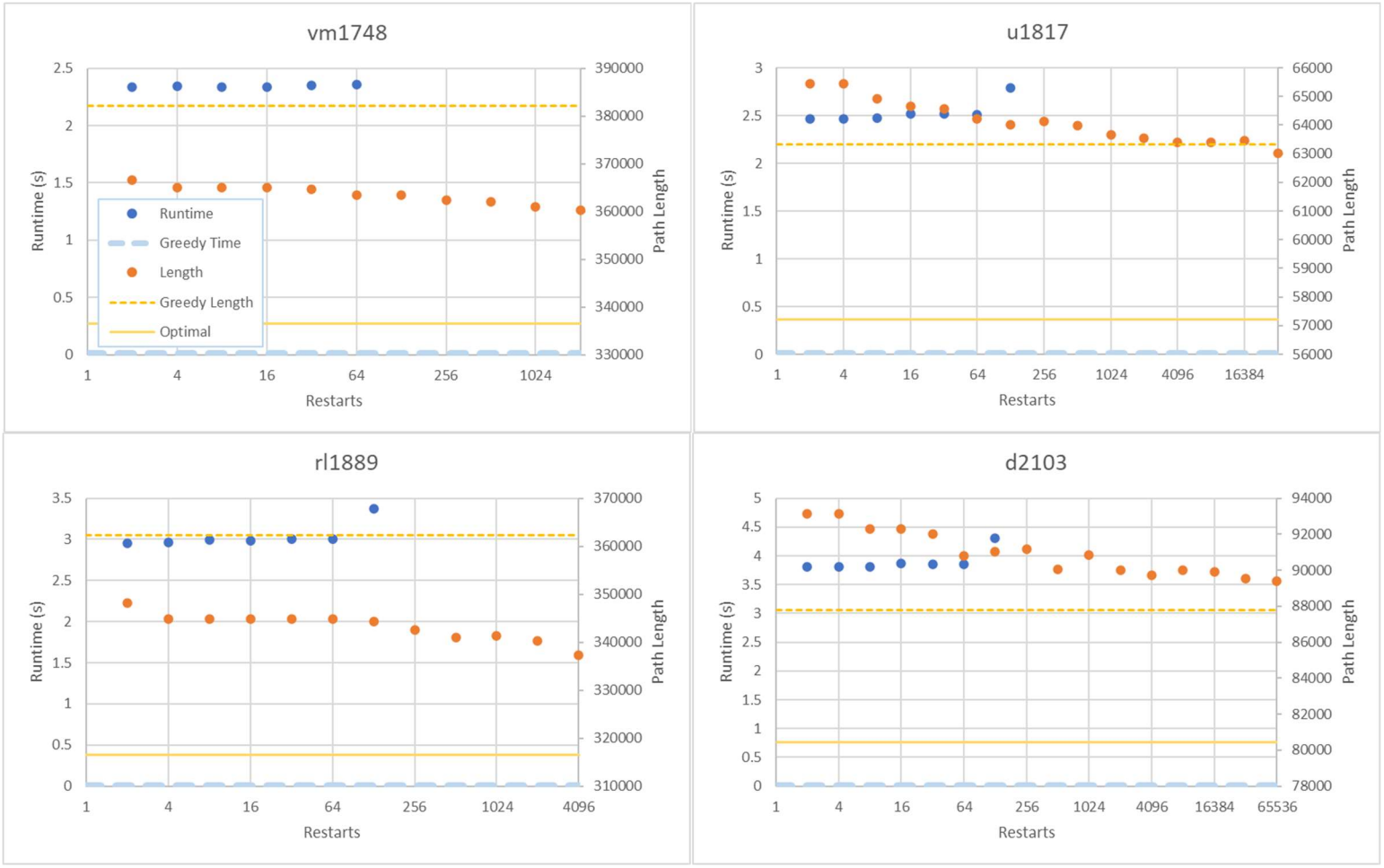

Figure 6 (cont'd) – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

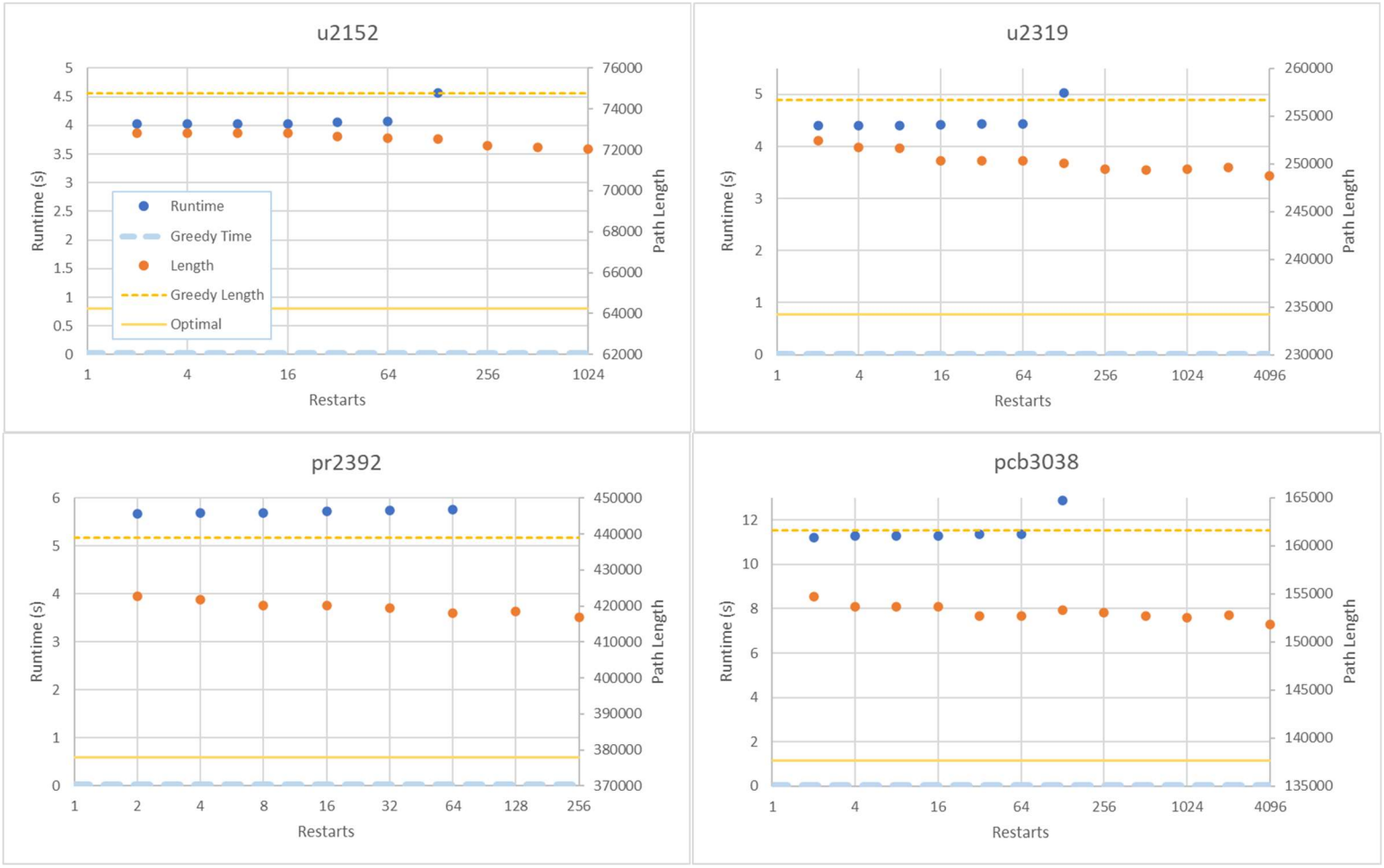

Figure 6 (cont'd) – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

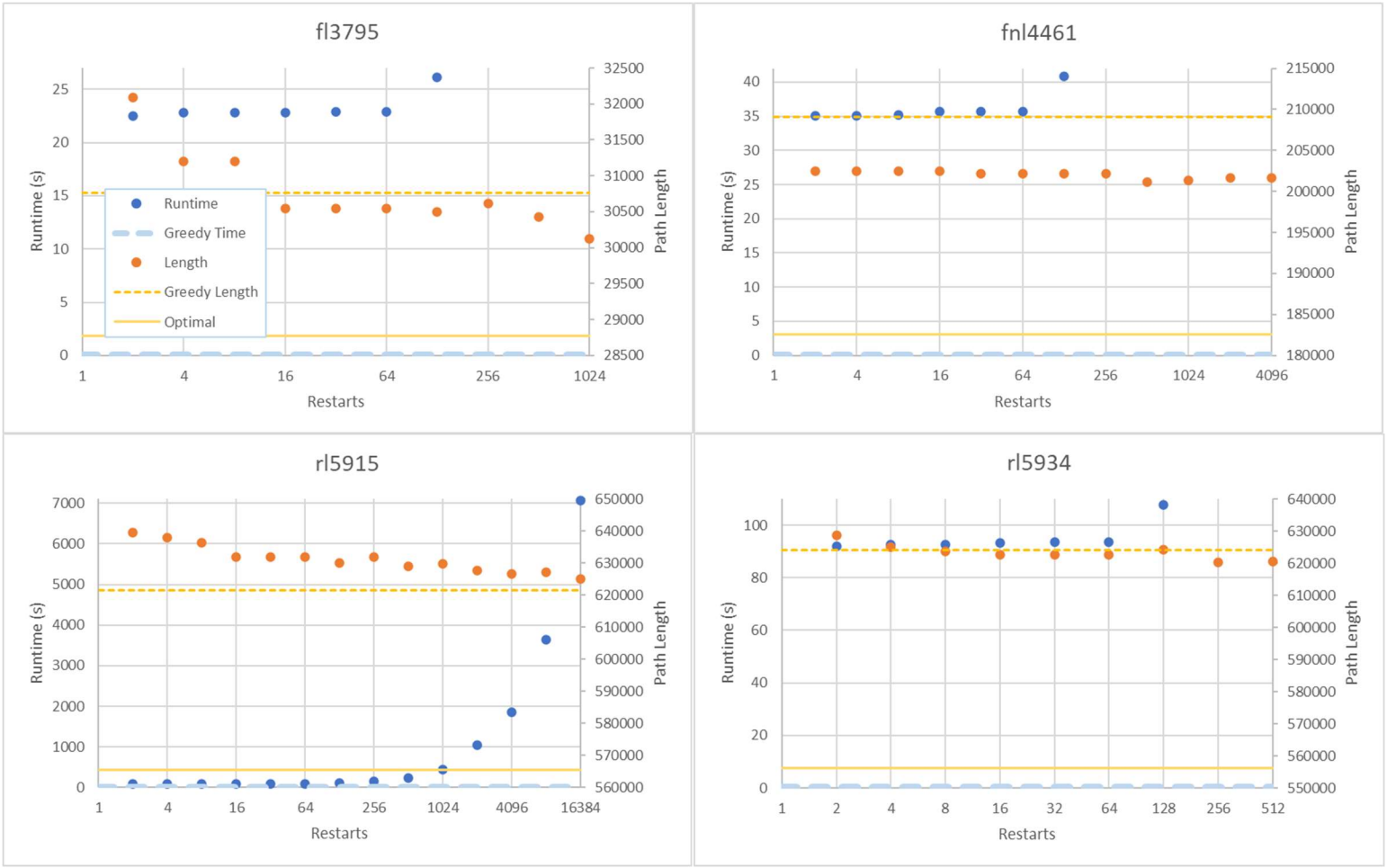

Figure 6 (cont'd) – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

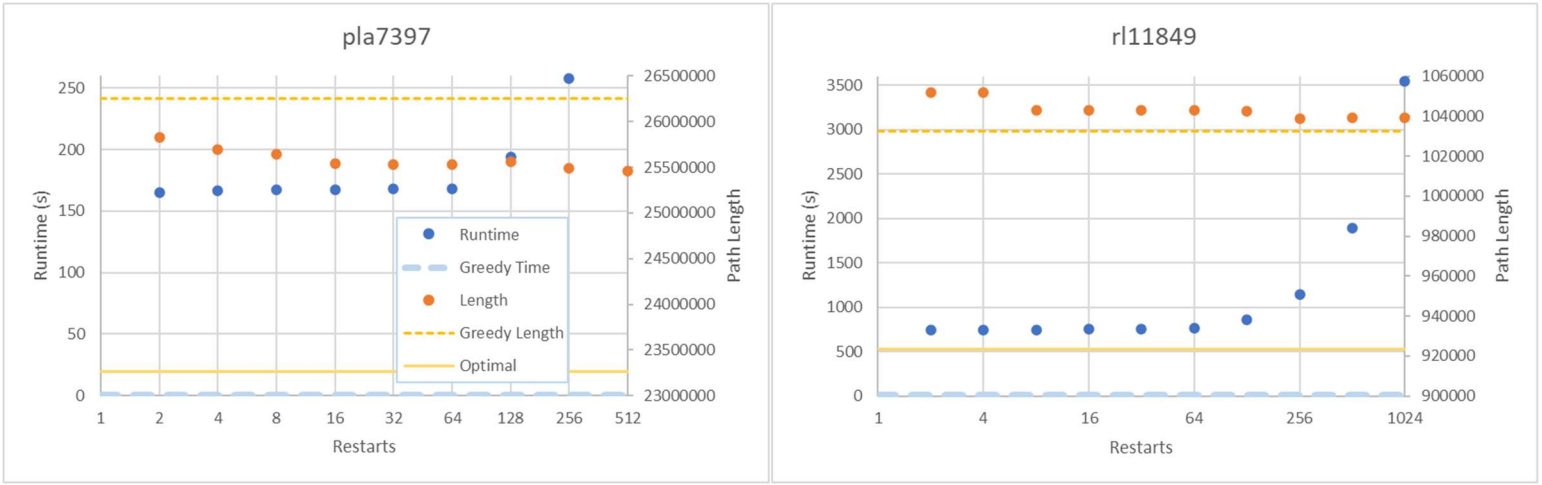

Figure 6 (cont'd) – Visualization of IHC 2-Opt comparison to the greedy heuristic. Shown here is the visualization of the tabulated data from **Table 2**. The IHC 2-Opt runtime is shown on the left axis as the blue dots, whereas the greedy heuristic's runtime is denoted as a blue hashed line. The right axis shows the final path lengths calculated by IHC 2-Opt as the red dots, whereas the greedy heuristic's final path length is shown as the red hashed line (recall that the greedy heuristic is deterministic and always achieves the same path length). The optimal path length is shown as a solid red line.

CONCORDE Comparison

The following data show the comparison between the greedy heuristic and the TSP solver, CONCORDE, for graphs that were runnable on the system configuration used for this study (see **Table 3** and **Table** 4; some graph runs were unfeasible due to the extended times benchmarked by TSPLIB; i.e. the benchmark time for graphs rl11849 and usa13509 is shown as approximately 155 days and 4 years, respectively). Again, the run time performance of the greedy heuristic exhibited far superior run times in comparison to the CONCORDE solver. According to CONCORDE's documentation [5], benchmarks were performed on version 99.12.15 with a flag of "-s 99" to define the random seed on a 500 MHz Compaq XP1000 workstation. Likewise, the comparison runs on Ithaca used the same flag for obtaining the CONCORDE runtime. The greedy heuristic was able to achieve a solution quality that was at most 16% from the optimal (see graphs pcb1173 and pr2392), and a highest speedup of 1,206,302 over CONCORDE (see graph u2319).

Table 3 – CONCORDE runs compared to the greedy heuristic. Shown here are the run comparisons between the system configuration used for this study and the greedy heuristic's performance, including the percent difference over the optimal length, the run time, and speedup over CONCORDE performance.

Nodes	File	Concorde Runtime (s)	GPU Length % Difference over Optimal	GPU Runtime	GPU Speedup over Concorde
100	kroE100.tsp	0.2	6%	0.0004	500
225	ts225.tsp	2.25	3%	0.0005	4,500
575	rat575.tsp	23.35	10%	0.0014	16,679
1002	pr1002.tsp	3.65	13%	0.0015	2,433
1084	vm1084.tsp	65.23	11%	0.0030	21,743
1173	pcb1173.tsp	39.52	16%	0.0028	14,114
1304	rl1304.tsp	27.07	10%	0.0029	9,334
1323	rl1323.tsp	626.79	9%	0.0028	223,854
1400	fl1400.tsp	3027.75	9%	0.0043	704,128
1432	u1432.tsp	73.95	15%	0.0034	21,750
1577	fl1577.tsp	1765.57	6%	0.0051	346,190
1748	vm1748.tsp	145.68	14%	0.0037	39,373
2319	u2319.tsp	11097.98	10%	0.0092	1,206,302
2392	pr2392.tsp	12.35	16%	0.0051	2,422

Table 4 denotes the benchmark times achieved by the developers of CONCORDE for the listed graphs. Due to the extended run times, the graphs were not tested on CONCORDE with this study's system. Still, comparing CONCORDE's benchmark times to the greedy heuristic shows that the greedy heuristic exhibits superior run time performance relative to the benchmark. For example, the TSP solver takes approximately 155 days to compute the optimal path for graph rl11849, but the greedy heuristic takes 0.06 seconds! As shown before however, this run time superiority comes at tradeoff of a solution quality that is just under 12% from the optimal solution (see **Table 1**'s data for rl11849).

Table 4 – The greedy heuristic's runtime in comparison to CONCORDE's benchmark times. Shown here are the greedy heuristic's runtimes achieved for the listed graphs. In this study's system configuration, however, CONCORDE runs of the listed graphs were unfeasible due to the extended run times listed by CONCORDE's benchmarks.

FILENAME	Benchmark Time (s)	GPU (Runtime)
d1291.tsp	27393.72	0.0029
u1817.tsp	449230.55	0.0051
rl1889.tsp	10023.02	0.0042
d2103.tsp	11179253.91	0.0079
u2152.tsp	45204.53	0.0063
pcb3038.tsp	80828.87	0.0249
fl3795.tsp	69886.48	0.0362
fnl4461.tsp	53420.13	0.0105
rl5915.tsp	2319671.71	0.0399
rl5934.tsp	588936.85	0.0204
pla7397.tsp	428996.2	0.0294
rl11849.tsp	~155 Days	0.0600
usa13509.tsp	~4 years	
brd14051.tsp	OPEN	
d15112.tsp	~22.6 years	
d18512.tsp	OPEN	
pla33810.tsp	OPEN	

Ant Colony Systems Comparison

The following data show the greedy heuristic's performance to that of Skinderowicz's Ant Colony System [32] (see **Table 5**). All runs were performed with the flag "--alg=mmas_wrs_bt" denoting the node selection scheme for the run. It is apparent that this particular heuristic performs well in terms of solution quality, generally keeping the final solution length within 1% from the optimal with a few exceptions whose solution quality was within 4% (see graphs pcb3038, fnl4461, pla7397, and rl11849; see **Figure 7**). Similar to Burtscher and O'Neil's implementation, the heuristic was able to achieve the optimal result for graphs kroE100 and ts225. Still, run times exceeded that of the greedy heuristic for even the smallest of graphs, and solution quality did remain consistent relative to that of the greedy heuristic.

Table 5 – Ant Colony System performance. Shown here is the performance of the Max-Min Ant System on the TSPLIB graphs. The table denotes the graph's optimal length as reported by TSPLIB, the solution length as computed by the Ant System, followed by the percent difference the length is from the optimal and run time.

FILENAME	Optimal Length	Final Solution Path Length	% Diff	Time (s)
kroE100.tsp	22068	22068	0.00%	0.3414
ts225.tsp	126643	126643	0.00%	0.8384
rat575.tsp	6773	6797	0.35%	3.6877
pr1002.tsp	259045	259537	0.19%	8.3668
vm1084.tsp	239297	239988	0.29%	11.3875
pcb1173.tsp	56892	57245	0.62%	12.3012
d1291.tsp	50801	50885	0.17%	16.2311
rl1304.tsp	252948	253558	0.24%	19.1571
rl1323.tsp	270199	271206	0.37%	20.3273
fl1400.tsp	20127	20167	0.20%	21.0434
u1432.tsp	152970	153753	0.51%	20.1726
fl1577.tsp	22249	22269	0.09%	23.7505
vm1748.tsp	336556	338601	0.61%	38.2227
u1817.tsp	57201	57517	0.55%	32.2369
rl1889.tsp	316536	316968	0.14%	44.9916
d2103.tsp	80450	80479	0.04%	40.9954
u2152.tsp	64253	64793	0.84%	46.1069
u2319.tsp	234256	236405	0.92%	61.4841
pr2392.tsp	378032	379912	0.50%	55.8826
pcb3038.tsp	137694	139490	1.30%	101.9360
fl3795.tsp	28772	29010	0.83%	159.9340
fnl4461.tsp	182566	188081	3.02%	262.2600
rl5915.tsp	565530	569111	0.63%	452.2410
rl5934.tsp	556045	560071	0.72%	452.0720
pla7397.tsp	23260728	23960336	3.01%	645.5420
rl11849.tsp	923288	956706	3.62%	1955.2600

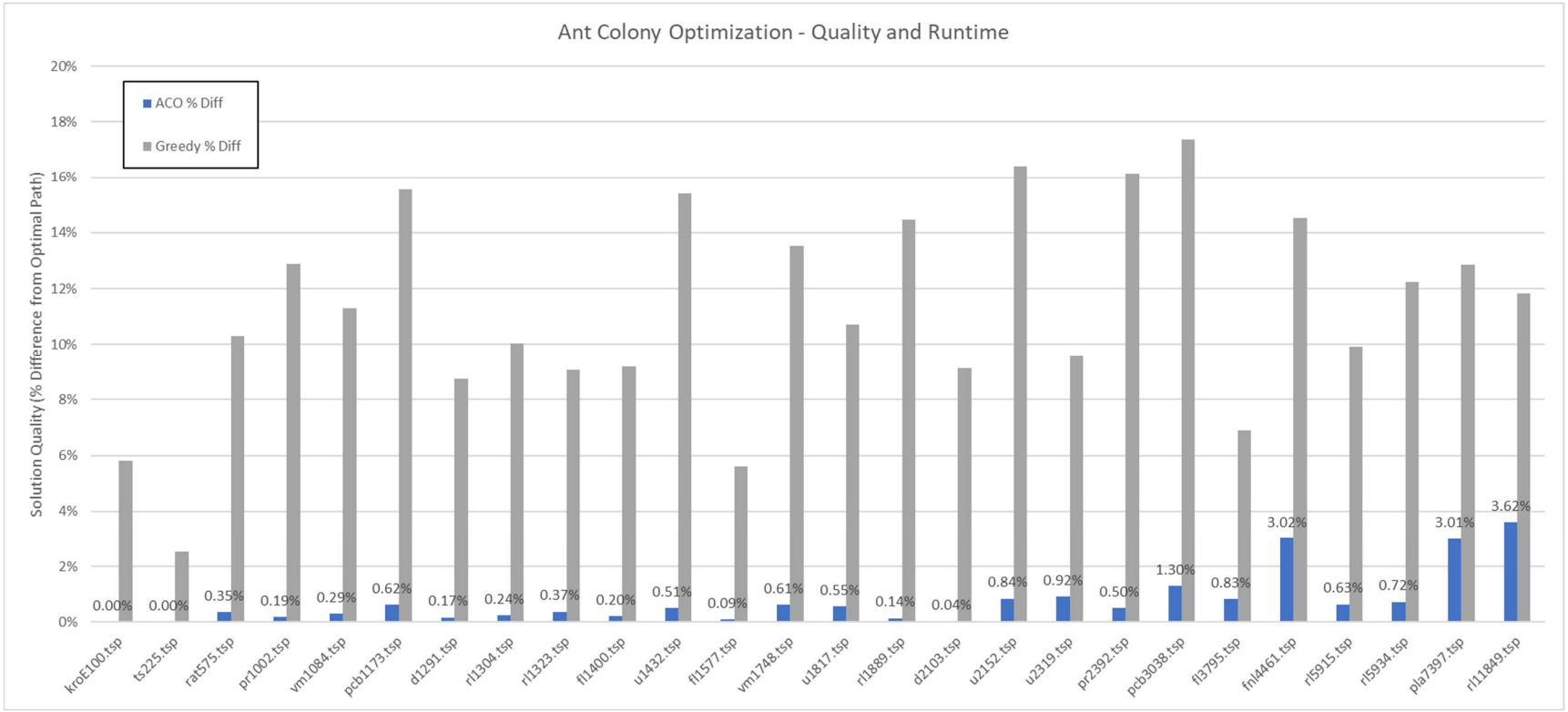

Figure 7 – Comparison of the solution quality of the Ant Colony System to the greedy heuristic. Shown here is the solution quality comparison for each of the TSPLIB graphs included in this study. The greedy heuristic is listed as "v9.8" and the Ant System is denoted as "ACO

VI. SUMMARY

The Traveling Salesman Problem continues to be a topic of study for many researchers, as it has many applications in a diverse set of disciplines. Having many possibilities for any one particular graph, this combinatorial optimization problem seeks a heuristic that reaches *near*-optimal results, for finding the optimal solution is both computationally expensive and intractable for large problem sizes. In this study, a heuristic based on Kruskal's algorithm was implemented—a complete, undirected graph's edges are sorted in non-decreasing order and iterated through for inclusion in the final tour, keeping the tour's overall path length at a minimum. As edges are traversed, edges are included in the final tour based on the following criteria: (1) if both vertices associated with the edge in question have a degree of less than 2, and (2) if both vertices have a degree of 1, they must not be a part of the same set of connected edges. The second criteria are determined by the use of a data structure known as a Union Find, a data structure that is efficiently able to keep track of what vertices are associated with what other vertices to make a set, thusly being able to prevent the early formation of cycles. Additionally, the Greedy Heuristic implementation used here makes use of a chunk-size that dictates how many edges to process before checking for completion of the tour. If the tour is not complete, the heuristic continues with the next chunk of edges, and the process repeats. This implementation also makes use of the C++ standard library's std::partition and std::find to emulate a form of quick-sort, where std::partition rearranges a range of elements based on a condition—in this case, whether an edge is shorter than a pivot edge—as well as OpenMP for the parallel

instantiation of edges. Serial implementations show that the most time-consuming portions of the heuristic involve sorting, and to further improve the sorting's execution time, a range of edges are separated according to whether they are eligible to be included in the final tour before the range is passed for sorting. In this way, edges that are not eligible for inclusion are automatically excluded from sorting, allowing for a quicker execution time. Moreover, the calls to std::partition and std::find can be called to execute with a parallel execution policy. Once the tour is complete, the city-order is reconstructed and the Hamiltonian Path's length is calculated. Parallelism in the heuristic's implementation has shown that execution times can be reduced, and this parallelism was exploited in CUDA, where the GPU was utilized for its better efficiency in handling parallel tasks over the CPU. This included a novel method of iterating over edges in the heuristic's main loop. Edges were primarily considered for tour-inclusion on the basis of their degree and their association with other vertices (namely, which set they were a part of).

Serial to Parallel Implementation

Transitioning from the serial implementation to the parallel CPU implementation did not show changes in performance for graphs that were below 1,748 nodes, but this is merely a side effect of the heuristic's implementation. Since the majority of the heuristic's execution time was spent in the sorting phase (data not shown), the parallelism defined by the implementation was not engaged. That is, the parallelism

was only engaged for graphs that exceed a particular threshold as defined by the implementation.

In this study, the threshold chosen for engaging parallelism in the preprocessing and sorting phase was 262,144 (512 * 512). When the number of edges that needed to be preprocessed or sorted exceeded that of the threshold, the parallel execution policies were engaged. Otherwise, the calls to std::partition and std::find were executed serially. This threshold was chosen by data collected on the run times achieved by selected different thresholds (data not shown), and a selection of 512 * 512 performed well. This value was then used for collecting data on the rest of the graphs.

In regards to the larger graphs, transitioning to the parallel implementation did consistently show better run time performance, at times exceeding the performance of that of the GPU implementation (see graphs pcb3038, fl3795, and rl5915). This could be due to a number of different possibilities that were not investigated in this study. For example, there was no correlation established between graph topologies and run times. In other words, the manner in which the graph's vertices are dispersed could have an effect on the efficiency of the heuristic, such as the potential for handling the magnitudes of the vertex coordinates themselves. Some graphs contain coordinates that are relatively close (i.e. x and y coordinates that remain under a value of 10,000) whereas other graphs may have vertices that are farther apart (i.e. x and y coordinates that exceed the hundreds of thousands). The heuristic may also have a disadvantage for the relative difference between all edges in the graph. For example, if the graph contained a large number of edges that were the same magnitude, the algorithm itself

may be encumbered by these large collections of same-weight edge instances. These aspects, however, were not investigated as part of this study.

One could notice that the runtime of the heuristic is not necessarily correlated with that of the number of nodes it contains. For example, graph d2103 has a CONCORDE benchmark time of 11,179,253.91 seconds (approximately 130 days), but the benchmark time for pr2392 (a graph with slightly more vertices) is just over 12 seconds. Likewise, the benchmark time for u1432 (a graph with fewer vertices) has a benchmark time of just under 74 seconds. For the greedy heuristic, rl5915 had a runtime of 0.0399 seconds, but graphs fnl4461 and rl5934 had runtimes of 0.0105 seconds and 0.0204 seconds, respectively. This discrepancy was not investigated as part of this study.

The same thinking could be applied to the differences in the heuristic's percent difference from the optimal length of the graph. Some graphs perform better than others in terms of the heuristic's implementation, but the specific nature of the reasoning behind this performance discrepancy was not investigated. There was, however, consistency between the final path lengths of the serial and parallel implementations of this study, which provides a solid point of reference for each implementation's runtime comparison.

When transitioning to the parallel GPU implementation, the effect on runtime can be easily noticed. With a few exceptions, the GPU outperformed that of the serial and parallel CPU implementations in terms of runtime. Generally, speedups achieved varied between 1 and 4, but the largest speedup between the serial CPU

implementation and the parallel GPU implementation was observed for rl11849 at a speedup of 8. There were a few instances that performed better on the parallel CPU implementation than that of the GPU (see graphs pcb3038, fl3795, and rl5915). This could be due to a number of possibilities as mentioned before. Perhaps the cumulative time needed to transfer data between the CPU and GPU exceeded that of the time needed to execute the heuristic on the CPU, but again, these specifics were not investigated as part of this study. Generally, speaking however, the heuristic did benefit from the GPU's implementation. Additionally, a maximum throughput of 668,000 nodes per second was achieved for graph pr1002.

It should also be noted that, due to the deterministic nature of the greedy heuristic's implementation—that is, the fact that the heuristic performs the same calculations in the same order for every run—the average runtime's standard deviation for the GPU implementation was 1.46×10^{-4}, providing a consistent basis for measuring the runtime metric (data not shown).

Parallelization of the Main Loop

The centerpiece of this study comes from the parallelization of the main loop's responsibilities: iterating through sorted edges, considering each one on an individual basis on whether its inclusion in the final tour is valid. Due to an edge's dependencies on whether previous edges have been included, parallelization of the main loop is difficult. Nonetheless, there is a key property of the main loop that can be exploited to introduce parallelization—the fact that the inclusion of an edge is solely dependent on

the status of its vertices, namely the degrees, in relation to previously-included edges that share those vertices. This implies that a range of edge may be considered simultaneously so long as the edges don't contain the same vertices. If difference edges happen to include the same vertices, the heuristic must still rely on an iteration of the sorted first-edge before considering the second (and potentially third, fourth, etc.). Furthermore, inclusion of the longer edge depends on whether the smaller edge was included. Due to the fact that a graph has a large number of edges, the main loop need only iterate fewer times than considering a single edge per iteration. This immensely reduces the time needed to compute the path, as (potentially) over a thousand edges can be considered at once.

IHC 2-Opt Comparison

In comparing the results achieved by the greedy heuristic to that of the IHC 2-Opt implementation, it can be observed that performance can vary based on the parameters used for a heuristic's execution. For example, in considering graph rl1304, solution quality was weaker than that of the greedy heuristic for restarts of 8 and fewer, but for restarts of 16 or greater, solution quality was better. Some graphs exhibited consistently poorer quality even when the number of restarts increased. For example, graph d2103 could not achieve solution qualities that were better than the greedy heuristic even when the number of restarts was 65,536. This leads to the conclusion that different heuristics offer different performance benefits in terms of runtime and solution quality that depends on the nature of the problem. Some graphs perform

better with one heuristic that other graphs would perform better with the same heuristic. Furthermore, performance benefits vary based on the parameters chosen for the given heuristic. In this case, the number of restarts affected the final solution quality at the expense of runtime. Generally speaking, the higher the number of restarts used for the IHC 2-Opt heuristic, the better the solution quality and higher the runtime. This was not fully consistent however. For example, when running the IHC 2-Opt heuristic on graph ts225 with 256 restarts, a final length of 127,463 was achieved, but when running the same graph with 512 restarts, the final path length achieved was 127,536. The same situation occurred with graph pr1002, where the final path length was longer when using 128 restarts compared to that of the final path length when using 64. Again, this could be due to the nature of the heuristic. Part of the IHC 2-Opt heuristic involves a degree of randomness when instantiating the graphs before they undergo the 2-Opt improvement heuristic. Naturally, some executions would fare better than others. Nonetheless, the IHC 2-Opt heuristic was able to achieve the optimal result in two graphs (see kroE100 and ts225), whereas the greedy heuristic came as close as 2.54% to the optimal (see graph ts225). In terms of the run time, however, the greedy heuristic consistently outperformed the IHC 2-Opt heuristic, achieving an average speed up of 1384.994. The maximum speedup achieved was 12,361. Ultimately, the chosen heuristic for a particular problem depends on whether the user is looking for better solution quality or faster run times.

CONCORDE Comparison

When comparing the greedy heuristic's runtimes to that of CONCORDE, it is immediately apparent that the greedy heuristic vastly outperforms CONCORDE in terms of runtime. The greatest speedup obtained by the GPU was 1,206,302 for graph u2319, but if the GPU runtime was compared to the CONCORDE benchmark time of d2103, the estimated speedup becomes 1,415,095,432!

In terms of comparing the greedy heuristic to that of a solver, the considerations one must bring forth now reside purely on how much of the solution quality would be sacrificed for run time benefits. In other words, the solver can achieve the optimal result, but at the expense of consuming a much larger amount of time to arrive at the solution. This is most apparent in the largest graph compared, rl11849, where the solver was benchmarked at 155 days. If the solution from the greedy heuristic was used, the solution, given in 0.06 seconds, comes at a sacrifice of nearly 12% of the optimal in terms of final path length. Additionally, graphs such as d15112 (not tested in this study) have a CONCORDE runtime of approximately 22.6 years, and some graphs, such as brd14051 and d18512, have yet to achieve a CONCORDE benchmark. Of course, the downside of this particular heuristic is that, considering the hardware constraints given in this study, such large graphs are intractable. Further study is open for opportunity in this regard.

CONCORDE also supports the notion that a graph's run time is not necessarily correlated with the number of nodes in the graph. For example, the runtime for graph u2319 was 11097.98 seconds, but the run times for graphs vm1748 and pr2392 were

145.68 seconds and 12.35 seconds, respectively. Graph topology, amongst other factors such as mode of edge weight (that is, the potential for a large number of edges to possess the same length), could play a role in the performance of the heuristic. The amount that each of these other factors would also vary on the effect's magnitude towards any particular heuristic and ultimately depends on the nature of the heuristic itself.

Ant Colony Optimization Comparison

The Ant Colony System shows its strength in its ability to get high quality results in relatively shorter run times. With a mere handful of exceptions, the ACO was able to obtain solution qualities below 1% in under a minute. The largest of the percent differences presented itself in graphs pcb3038, fnl4461, pla7397, and rl11849, and these graphs, as one would come to expect, have the longer of the runtimes. Still, the greedy heuristic was able to outperform the ACO in terms of runtime. The longest runtime for the greedy heuristic was 0.06 seconds, but the fastest ACO was 0.3414 seconds. This reiterates the notion that different heuristics offer different benefits. The ACO heuristic benefits in that solution quality is high with final path lengths that are very-near optimal. Run time is still longer, but certainly not as long as when compared to CONCORDE. The run times for ACO are still considered feasible in the sense that they do not take many years to compute. The greedy heuristic in this study managed to achieve a maximum speedup of 32,588 on graph rl11849 over ACO, but generally, the larger the number of nodes in a graph, the greater the speedup benefit.

Skinderowicz [32] continues to explain in his study of ACO that other so-called metaheuristics are also capable of achieving optimal results and at times, with faster run time. He compares his heuristic's implementation to that of the "Artificial Bee Colony with a Modified Choice Function" heuristic, an implementation that was able to achieve optimal results in a large majority of TSP graphs over ACO. Quite often though, these run times were only slightly slower than ACO. This is also enlightening in the sense that different approaches and heuristics, however complex they may be, offer different benefits when it comes to solution quality and runtime. They may also reveal additional details in terms of sub-algorithms in their heuristics. For example, the greedy heuristic revealed that edge iteration can be much faster by taking an alternate approach to analyzing the properties of vertices. Perhaps some aspect of these other heuristics is applicable to the greedy heuristic in order to improve upon solution quality.

VII. FUTURE WORK

There yet remains further possibility for study in this heuristic's implementation. For example, heuristics, however useful they may be for some graphs, may exhibit weaker or stronger performance on other graphs. Establishing a correlation between the heuristic and the graph topology would be useful for alternative implementations that can potentially further reduce the heuristic's runtime.

Other heuristics also combine constructive and improvement heuristics for better solution quality and the expense of longer runtimes. The heuristic implemented here exhibited extremely fast performance in finding valid tours, and the performance benefit is extended in that the heuristic is deterministic. That is, the heuristic will always perform the same steps in the same order for every execution. Other heuristics' approach, at times, makes use of a random number generator for their chosen mode of attack for a particular graph. Such heuristics are termed metaheuristics, and they show a distinct disadvantage in that some execution times are arbitrarily faster or slower than others. This heuristic being deterministic provides ample opportunity for combination with improvement heuristics such as the K-Opt or Lin-Kernighan heuristics for both fast execution times and high-quality solutions.

Though negligible in the time it takes to execute, the heuristic's final city-visit-order could use some reevaluation for a truly parallel implementation. Currently, the heuristic's implementation merely executes sequential steps on parallel hardware, which makes the implementation no different that the CPU serial implementation. Further opportunity for parallelization, therefore, can be explored here.

Another opportunity for study is the investigation into changing the fundamental data types that the GPU implementation uses for its execution with the aim of supporting larger inputs.

Specific to Texas State University's Efficient Computing Laboratory's Ithaca Linux environment, further exploration for larger graph CONCORDE benchmarks remains open. For example, TSPLIB's CONCORDE benchmarks place graph rl11849.tsp's compute time at approximately 155 days, but this benchmark is from a series of tests that took place more than a decade ago. Computing power has inevitably become more developed and accessible, and establishing a benchmark for the larger graphs could prove insightful.

Reimplementation of the heuristic in a manner that makes efficient use of a GPU's warp utilization or global memory coalescing, as well as the reduction of thread divergence, may prove beneficial to the heuristic's overall runtime. The implementation in this study, as remarkable as the execution times have been shown to achieve, did not take an approach that took such aspects into consideration during its development.

Further exploration into alternative, more powerful GPUs is also a considerable option for gathering data on larger TSP graphs. Current memory constraints prohibit the program's GPU execution of graphs that were larger than 11,849 nodes. Perhaps even an alternative implementation that utilizes multi-GPU processing may prove beneficial towards characterizing the limits of this heuristic.

BIBLIOGRAPHY

[1] Applegate, David, et al. "Implementing The Dantzig-Fulkerson-Johnson Algorithm For Large Traveling Salesman Problems." *Mathematical Programming* (2003): 91-153.

[2] Ivansson, Lars; Lagergren, Jens."Algorithms For Rh Mapping: New Ideas And Improved Analysis." *SIAM Journal on Computing.* (2004), Vol. 33 Issue 6, 89-108.

[3] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *"The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)."* Princeton, NJ, USA: Princeton University Press, 2007

[4] O'Neil, M. A., and Burtscher, M. "Rethinking the Parallelization of Random-Restart Hill Climbing - A Case Study in Optimizing a 2-Opt TSP Solver for GPU Execution". *GPGPU-8* (2015): 99-108.

[5] *Concorde TSP Solver*. (2020, August 13). Retrieved from The Traveling Salesman Problem: http://www.math.uwaterloo.ca/tsp/concorde/index.html

[6] C. N. Ravikumar, et al. "An Efficient Technique To Solve TSP And Its Extension To Spatially Spread Vertices" *Current Science*. 75(10):1046-1052.

[7] *TSPLIB*. (2020, August 8). Retrieved from Ruprecht-Karl University of Heidelberg: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[8] Rocki, K. and Suda, R. 2012. "An Efficient GPU Implementation of a Multi-Start TSP Solver for Large Problem Instances". *Proceedings of the Fourteenth Annual Conference Companion on Genetic and Evolutionary Computation* (Jul. 2012), 1441-1442.

[9] Laporte, Gilbert. "The Traveling Salesman Problem: An overview of exact and approximate algorithms" *European Journal of Operational Research.* (1992), 59(2), 231-247.

[10] Yelmewad, P.; Talawar, B.. "Parallel Iterative Hill Climbing Algorithm To Solve TSP On GPU" *Concurrency Computation*, (2019), 31(7)

[11] Xiang, Zuoyong, et al. "Solving Large-Scale TSP Using a Fast Wedging Insertion Partitioning Approach" *Mathematical Problems in Engineering*, (2015)

[12] Michael Held; Richard M. Karp. "The Traveling-Salesman Problem and Minimum Spanning Trees" *Operations Research*. 18(6):1138-1162

[13] Kindervater, Lenstra, et al. “The Parallel Complexity Of TSP Heuristics.” *Journal of Algorithms*. (1989) 10(2):249-270

[14] Golden, B. L. “A Statistical Approach to the TSP” *Networks*. (1977) (no.~3):209-225

[15] Rocki, K. and Suda, R. “High Performance GPU Accelerated Local Optimization in TSP” *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum Parallel and Distributed Processing Symposium Workshops & PhD Forum* (*IPDPSW*). (2013) 1788-1796

[16] Burkard , Rainer E.; et al. “Well-Solvable Special Cases of the Traveling Salesman Problem - A Survey” *SIAM Review*. (1998) 40(3):496-546

[17] Bland, Rg; Shallcross, Df. “Large Traveling Salesman Problems Arising From Experiments In X-Ray Crystallography A Preliminary Report On Computation” *Operations Research Letters*; (1989); 8; 3; 125-p128

[18] Johnson, David S. and McGeoch, Lyle A. “The Traveling Salesman Problem A Case Study in Local Optimization” *Local Search in Combinatorial Optimization, E. H. L. Aarts and J. K. Lenstra (eds.),* John Wiley and Sons, London (1997), 215-310

[19] Rosenkrantz , Daniel J., et al. “An Analysis of Several Heuristics for the Traveling Salesman Problem” *SIAM Journal of Computing* (1977) 6(3)

[20] Rego, César, et al. “Traveling Salesman Problem Heuristics: Leading Methods, Implementations.” *European Journal of Operational Research*. (2011) 211(3):427-441

[21] Yelmewad, Pramod and Talawar, Basavaraj. “Near Optimal Solution for Traveling Salesman Problem Using The GPU” *IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT) Electronics, Computing and Communication Technologies (CONECCT)*, (2018)

[22] Agarwala, R., et al. “A Fast and Scalable Radiation Hybrid Map Construction and Integration Strategy”. *Genome Research* (2000), 10(3):350-364

[23] Katajainen, Jyrki and Nevalainen, Olli. “An Alternative For The Implementation Of Kruskal’s Minimum Spanning Tree Algorithm” *Science of Computer Programming* (1983) 3(2): 205-216

[24] Osipov, V., et al. “The Filter-Kruskal Minimum Spanning Tree Algorithm”. *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)* (2009) 52-61

[25] Katsigiannis, A., Anastopoulos, N., Nikas, K., & Koziris, N. (2012). “An Approach To Parallelize Kruskal’s Algorithm Using Helper Threads”. International Symposium on Parallel and Distributed Processing Symposium Workshops & PhD Forum (pp. 1601-1610). Shanghai: IEEE.

[26] O'Neil, M. A., et al. “A Parallel GPU Version of the Traveling Salesman Problem”. *Department of Computer Science, Texas State University.* (2015)

[27] Kruskal, J.B.. “On The Shortest Spanning Subtree Of A Graph And The Traveling Salesman Problem” *Proceedings of the American Mathematical Society* (1956)

[28] NVIDIA CUDA Programming Guide (2020, August 13) http://docs.nvidia.com/cuda/index.html

[29] Reinelt, Gerhard. “TSPLIB A Traveling Salesman Problem Library” *ORSA Journal on Computing.* Fall91, Vol. 3 Issue 4, p376

[30] Bai, Hongtao, et al. “MAX-MIN Ant System on GPU with CUDA” *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)* :801-804 (2009)

[31] Delévacq, Audrey, et al. “Parallel Ant Colony Optimization on Graphics Processing Units” *Metaheuristics on GPUs*, *Journal of Parallel and Distributed Computing*. January 2013 73(1):52-61

[32] Skinderowicz, Rafał. ”Implementing a GPU-based parallel MAX–MIN Ant System” *Future Generation Computer Systems.* May 2020, Vol. 106, p277-295.

[33] Skinderowicz, Rafał. “The GPU-based parallel Ant Colony System” *Journal of Parallel and Distributed Computing*. December 2016 98:48-60

[34] Chen, Shyi-Ming and Chien, Chih-Yao. “Parallelized genetic ant colony systems for solving the traveling salesman problem” *Expert Systems With Applications* (2011) 38(4):3873-3883

[35] *TSP_{GPU} v1.1 and v2.3* (2020, August 22) Retrieved from The Efficient Computing Laboratory: https://userweb.cs.txstate.edu/~burtscher/research/TSP_GPU/

[36] Jiening, W., et al. “Implementation of Ant Colony Algorithm Based on GPU.” *Sixth International Conference on Computer Graphics, Imaging and Visualization* (2009), pp. 50-53.

[37] Cecilia, J.M., et al. “Parallelization Strategies for Ant Colony Optimisation on GPUs” *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, (2011)339-346

[38] Fu, Jie, et al. “A Parallel Ant Colony Optimization Algorithm With Gpu-Acceleration Based On All-In-Roulette Selection” *Third International Workshop on Advanced Computational Intelligence Advanced Computational Intelligence (IWACI),* (2010):260-264

[39] You, Y.S. “Parallel Ant System for Traveling Salesman Problem on GPUs.” *Eleventh Annual Conference on Genetic and Evolutionary Computation (2009)*

[40] Fujimoto, N. and Tsutsui, S. “A Highly-Parallel TSP Solver for a GPU Computing Platform.” *Lecture Notes in Computer Science (2011),* 6046:pp. 264-271.

[41] Dorigo, M. and Gambardella, L.M. “Ant colony system a cooperative learning approach to the traveling salesman problem” *IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, pp. 53-66. April 1997*

[42] CUB Documentation (2020, August 26) https://nvlabs.github.io/cub/index.html