Solving the Traveling Salesman Problem Using Ordered-Lists

THESIS

Petar D. Jackovich, 1st Lt, USAF

AFIT-ENS-MS-19-M-127

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENS-MS-19-M-127

SOLVING THE TRAVELING SALESMAN PROBLEM USING ORDERED LISTS

THESIS

Presented to the Faculty

Department of Operational Sciences

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Operations Research

Petar D. Jackovich, B.S.

1st Lt, USAF

21 March 2019

AFIT-ENS-MS-19-M-127

SOLVING THE TRAVELING SALESMAN PROBLEM USING ORDERED LISTS

THESIS

Petar D. Jackovich, B.S.
1st Lt, USAF

Committee Membership:

Lt Col Bruce A. Cox, PhD
Chair

Dr. Raymond R. Hill, PhD
Member

AFIT-ENS-MS-19-M-127

Abstract

The arc-greedy heuristic is a constructive heuristic utilized to build an initial, quality tour for the Traveling Salesman Problem (TSP). There are two known sub-tour elimination methodologies utilized to ensure the resulting tours are viable. This thesis introduces a third novel methodology, the Greedy Tracker (GT), and compares it to both known methodologies. Computational results are generated across multiple TSP instances. The results demonstrate the GT is the fastest method for instances below 400 nodes while Bentley's Multi-Fragment maintains a computational advantage for larger instances.

A novel concept called Ordered-Lists is also introduced which enables TSP instances to be explored in a different space than the tour space and demonstrates some intriguing properties. While computationally more demanding than its tour space counterpart, the solution quality advantages, as well as a possibly higher proportion of optimal occurrences, when optimality is achievable via the ordered-list space, warrants further investigation of the space. Three meta-heuristics that leverage the ordered-list space are introduced. Testing results indicate that while at a severe iteration disadvantage, these methodologies benefit from using the ordered-list space which yields a higher per iteration improvement rate.

Acknowledgements

First, I would like to thank my dog, Modi, for tolerating the long hours I spent not giving him attention while I was at school or in my home office working on this thesis. I would like to thank my parents for listening to my weekly ramblings and emotional vents about this thesis. I would like to thank Dr. Raymond Hill for convincing me to pursue this thesis topic. Finally, I'd like to thank my research advisor, Lt Col Bruce Cox, for putting up with my varying levels of motivation and excitement throughout the course of this thesis.

Petar D. Jackovich

Table of Contents

Page

List of Figures

List of Tables

I. Introduction

This thesis presents a novel sub-tour tracking and elimination methodology, the Greedy Tracker (GT), which ensures feasible solutions to the Traveling Salesman Problem during the implementation of the arc-greedy constructive heuristic. The GT is compared to other currently accepted sub-tour elimination methodologies to examine situational computational advantages. The paper then utilizes constructive heuristics to develop and explore a novel meta-heuristic that seeks to find an optimal, or near optimal, tour utilizing a novel concept called Ordered-Lists.

1.1 Motivation

Linear programming problems fall under the mathematical topic of optimization; they seek to optimize a linear function representing a measure of merit while minding linear equality and or inequality constraints on the systems performance [4]. The term linear programming was coined by economist and mathematician T.C. Koopmans based on work that George B. Dantzig was doing as a mathematical advisor to the United States Air Force during the late 1940s. Dantzig later developed the "simplex method" to solve these linear programs which became widely accepted due to its ability to model important and complex management decision problems and its capability for producing solutions to many important linear programs in a reasonable amount of time. However, the simplex method was not able to solve all LPs in a reasonable amount of time, leading mathematicians to seek an understanding on the types of problems that proved intractable for the method. Combinatorial opti-

mization problems are a subset of discrete linear programs that involve finding an optimal set from a finite set of solutions. While these problems theoretically have fewer possible solutions than a traditional linear program, they break the underlying continuity assumptions used in the simplex method thus preventing its usage. Other direct solution approaches to combinatorial optimization problems have also proved intractable, due to their exponential computational growth as problem size increases. One such combinatorial optimization problem that has long captured the interest of mathematicians is the traveling salesman problem.

1.2 The Traveling Salesman Problem

Applegate et al [5] describes the traveling salesman problem as, "Given a set of cites along with the cost of travel between each pair of them, the traveling salesman problem, or TSP for short, is the problem of finding the cheapest way of visiting all the cities and returning to the starting point." It can also be mathematically defined as, given a complete undirected graph $G = (V, E)$, cities are represented via the graph vertices, and edges represent the paths between the cities where the edge weights are the distances between each city. In terms of a graph the problem can be posed as: What is the shortest tour that visits all vertices once and returns to the starting vertex? One of the earliest examples of a similar graph problem was that of Euler's bridge conundrum in Konigsberg. The city of Konigsberg consisted of four land areas separated by two branches of the river Pregel but connected by seven bridges. Euler analyzed the challenge of finding a way to cross all the bridges exactly once and return to the origin. This problem differs from the TSP as it seeks to travel each arc once, and return to the starting node. However, while different, Euler's problem established much of the graph theory that is utilized to define problems like the TSP. The exact origins of the TSP are not known and there are many examples of

Figure 1. Konigsberg Bridges [1]

other similar early concepts. The first recorded use of the phrase "Traveling Salesman Problem" occurred in 1949 by Julia Robinson in her paper *On the Hamiltonian game (a traveling salesman problem)* [5]. When traditional linear programming methods were applied to the TSP, intractability issues arose [5]. It has since been shown this is because the TSP falls into a class of known computationaly 'hard' problems called NP-Complete [6]. As a result, nontraditional methods such as heuristics are often used when solving the TSP.

A heuristic is a "method which, on the basis of experience or judgment, seems likely to yield a reasonable solution to the problem, but which cannot be guaranteed to produce the mathematically optimal solution" [7]. This is the key difference between a heuristic and an algorithm. An algorithm guarantees optimality, whereas a heuristic does not. Heuristics have many advantages over algorithms, especially when it comes to the class of NP-Complete problems. Evans and Zanakis [8] present a multitude of these reasons, but considering the intractability of the TSP, the primary reason is that while "An exact method is available. It it is computationally unattractive due to excessive time and or storage requirements. Large real-world complex problems

may prevent an optimizer from finding an optimal or even a feasible solution within a reasonable effort. Heuristics, on the other hand, can produce at least feasible solutions with minimal time and storage requirements."

Many heuristics utilize a greedy-type methodology, where the best choice according to some predefined parameter is selected at each step of the method. An example of a greedy-type method for the TSP is the arc-greedy constructive heuristic, where the shortest available arc is added to the tour. However, this greedy heuristic runs the risk of generating sub-tours. These sub-tours are disconnected tours of less than size N (where N is the number of nodes present in the graph) that prevent a single continuous tour from being formed. Some research has been completed to develop methodologies that avoid sub-tours when utilizing the arc-greedy heuristic [9][10].

1.3 Research Questions

1. This paper introduces a novel sub-tour elimination methodology for the arc-greedy heuristic that is compared to two known sub-tour elimination methodologies. Computational results are generated across multiple TSP instances for each method.

2. A novel concept called Ordered-Lists is introduced which enables TSP instances to be explored in a different space than the tour space. This concept demonstrates some intriguing properties which we leverage in some novel meta-heuristics.

1.4 Outline

In Chapter 2, various methods used to solve the TSP are reviewed. In Chapter 3, two known arc-greedy sub-tour tracking and elimination methodologies are introduced with pseudo code, examples, and theoretical advantages. This chapter also introduces a novel sub-tour elimination method, the Greedy Tracker. Chapter 4

summarizes each of these methodologies performance across different instances of the TSP focusing on run-time comparisons and identifying run time trends due to underlying instance structure. In Chapter 5, a novel concept for viewing TSP instances, Ordered-Lists, is introduced and a novel TSP meta-heuristic utilizing this concept is proposed. In Chapter 6, results from the proposed meta-heuristic are summarized followed by Chapter 7 where, all results are concluded with recommendations on future research of utilizing the arc-greedy methodology on the TSP and other combinatorial optimization problems.

II. Literature Review

In this chapter several well known algorithms and heuristics used to solve the TSP are introduced. The chapter starts with a brief overview of NP-Hardness and is followed by the Linear Programming formulation of the TSP, an overview of heuristic methods, and an introduction to popular construction and meta-heuristic's specifically used on the TSP. Understanding these motivates the sub-tour elimination methods for arc-greedy constructive heuristics as well as methodologies used in the Ordered-List meta-heuristics introduced later in this paper.

2.1 NP-Hardness

The TSP is an NP-Complete problem [6]. NP-complete is one of the classes of computational complexity. The other classes P, NP, and NP-hard along with their currently understood relationships are found in Figure 2. Briefly, the class P consists

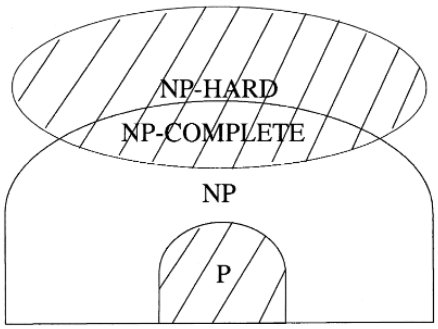

Figure 2. Computational Complexity [2]

of problems solvable in polynomial time [2], the class NP consists of problems whose solutions can be verified in polynomial time. It is an open question if the class P is equivalent to class NP [11]. In additions, the class NP-hard can be informally thought of as the class of problems that are "at least as hard as the hardest problems in NP." The intersection of NP-Hard problems , and NP problems is called NP-complete. No

one has yet developed an efficient method for solving large instances of NP-complete problems to optimality [12]. The inclusion of the TSP in the set of NP-complete problems motivates the usage of other solving techniques such as LPs with cuts and heuristics.

2.2 LP Relaxation

The LP formulation for the the TSP as initially described by Dantzig et al [13] is given as:

$$\begin{aligned} &\text{minimize} && c^T x && \\ &\text{subject to} && 0 \leq x_e \leq 1 && for\ all\ edges\ e, \\ & && \sum (x_e \ : \ v\ is\ an\ end\ of\ e) = 2 && for\ all\ cities\ v. \end{aligned} \tag{1}$$

In this formulation, the decision variable x_e represents the choice of including edge e in the tour. The objective function associates a cost matrix with this decision variable, while the constraints ensure that each edge is used at most once and that each node has "two edges". The authors, however, go on to discuss that this formulation is not the actual problem they want to solve, but is instead the problem they can solve. The formulation above is a relaxation of the actual problem which allows for a solution containing sub-tours, as well as solutions that partially assign edges. For instance, Figure 3 shows a allowable solution to (1), as it satisfies all constraints however it does not produce a single continuous tour. The answer found from the relaxation is however still useful as it provides a lower bound objective value for a TSP instance which can then be used to grade the quality of a proposed tour found by heuristics, or as a foundation for cutting plane algorithms.

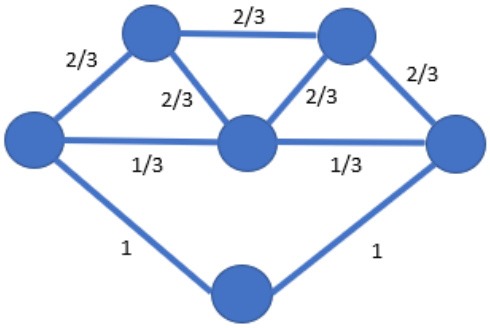

Figure 3. TSP Relaxation Solution

2.3 Cutting Plane Method

Research from Heller [14] and Kuhn [15] indicated it may be possible to define beforehand a finite list of inequalities to add to the LP relaxation to exactly define the feasible region. However, the full list of inequalities could be far too large for any linear programming solver to handle directly. One methodology proposed by Applegate et al [5] to utilize this list is to implement a series of iterative cuts to remove infeasible solutions. A cut, or cutting plane, is a linear inequality that constricts the convex hull of the feasible region. The process of adding these cuts involves solving the LP relaxation, examining the solution to determine if it is a feasible tour, determining which additional inequalities are necessary to break any sub-tours, adding them and resolving the problem. This continues until a feasible, and thus optimal solution, to the TSP is found. Iterative cutting is possible because not every inequality needs to be added to the LP to find the optimal solution. Therefore, by solving multiple smaller LPs and iteratively adding cutting planes to remove infeasible intermediate tour solutions, an optimal solution can be found. However, the time to solve grows exponentially depending on the number of cuts that may be necessary. Because of this, a heuristic solution methodology appears to be the best way to quickly produce good, if not optimal tours [9].

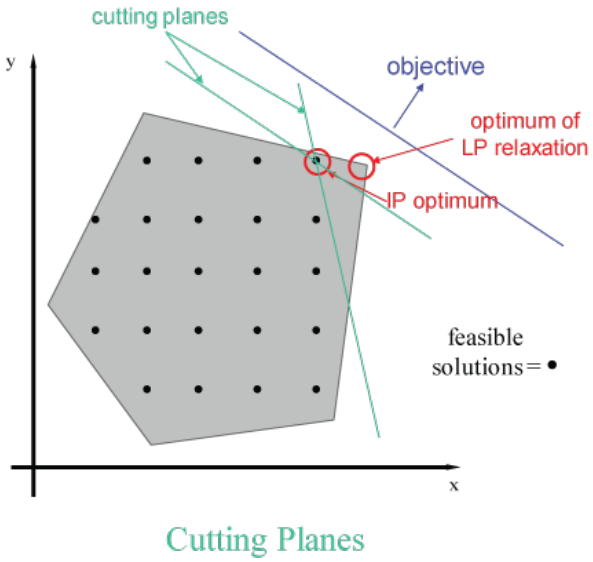

Figure 4. Cutting Plane [3]

2.4 Heuristics

The difficulties encountered in applying cutting planes motivate the usage of heuristic methodologies to solve the TSP. The earliest recorded use of heuristics traces all the way back to ancient Greek mathematical literature. The name heuristic comes from the Greek verb "heurskein" meaning "to find". From then to now people have been applying creative methodologies to solve difficult problems. As the name implies, some of the earliest examples of the TSP were records of various Salesman discussing the idea that more thought should be put into how they organize their journeys, or tours, to neighboring cities. A excerpt from the Commis-Boyageur, a 1830s German traveling salesman handbook [5], was brought to the attention of the TSP research community by Heiner Muller-Merbach in 1983 which translated to, "The main thing to remember is always to visit as many localities as possible without having to touch them twice." This excerpt indicates that as early as the 1800s, a salesman was cognizant that his routes should be planned as to minimize the number of places he visits more than once.

There are many desired qualities that make a good heuristic. Evans and Zanakis [8] give a list of characteristics they feel defines a good heuristic:

- Simplicity,
- Reasonable core storage requirements,
- Speed,
- Accuracy,
- Robustness,
- Acceptable to multiple starting points,
- Produce multiple solutions,
- Good stopping criteria,
- Statistical estimation, and
- Interactive.

While many of these are intuitive, some may require further explanation. Because a heuristic does not necessarily converge to the optimal solution like a algorithm, the starting point, or initial solution, is very important. Different feasible initial solutions start at different locations within the feasible region and can often converge to different local optima. By making a heuristic acceptable to multiple starting solutions, it has a better chance to test and explore more of the feasible region. As it's Greek root implies, A heuristic also needs to have a good stopping mechanism to determine when it has "found" a suitable solution. This ensures that the heuristic does not run for a unreasonably long time searching for answers without improvement. It also ensures the the heuristic does not stop before possibly reaching a very good set, or neighborhood, of new solutions.

Most heuristics can be broken into three categories, construction heuristics, local-search heuristics, and meta-heuristics. In relation to the TSP, construction heuristics

build a tour from scratch, local heuristics improve a given tour, and meta-heuristics apply a combination of constructive and iterative local-search heuristics [16], of particular note is a meta heuristics ability to be interactive. Modern meta-heuristics often include user definable elements, which allow the user to tune the meta-heuristic for the given instance it is solving. These elements often include number of iterations, stopping criteria, number of initial starting solutions generated, and the definition of neighboring solutions, all of which are very important to how the meta-heuristic performs with regards to many of Evans and Zanakis's qualities.

2.5 Greedy-type Construction Heuristics

One of the most common construction heuristic methodologies is the greedy heuristic. A greedy heuristic is one that at each step selects the best decision for a given metric, with no regard to how such choices may effect future decisions. For the TSP, there are three primary greedy construction heuristics; Nearest neighbor (node-greedy), arc-greedy, and Recursive Selection.

Nearest Neighbor (node-greedy heuristic).

The nearest neighbor(NN) heuristic was first applied to the TSP in a 1954 paper by Flood [17] but was introduced as the "next closest city method." The process was later refined by Dacey [18] and coined with its eventual name. The NN starts at an arbitrary city, and successively visits the closest unvisited city. It is important to to note that the nearest neighbor heuristic maintains a single path fragment that originates at the predetermined starting city, and cannot be closed into a cycle until every node has been visited. Therefore the decision of "which arc to add" is limited to only those arcs that leave the current end node of the fragment, this yields an algorithm run time of $O(N^2)$. Future work by Bentley [9] allowed this heuristic to

perform in $O(N \log N)$. This methodology allows NN to quickly create an initial tour which avoids sub tours. However, this approach is extremely sensitive to the choice of starting node, especially in larger instances. This sensitivity leads to a common practice of running NN for all cities as the starting node to provide the best solution, which is never more than $([\log N] + 1)/2$ times the length of the optimal tour[19]. The shortfall of this heuristic is that one can easily create examples that cause the heuristic to produce the worst possible solution. A simple example of a scenario where this occurs can be seen in Figure 5. If Node A is selected as the starting node, the

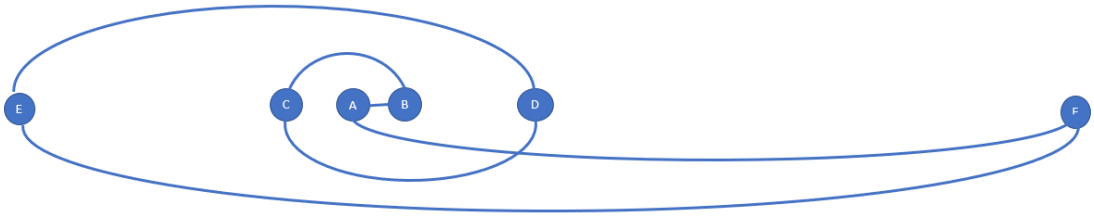

Figure 5. Greedy Worst Solution Example

heuristic is stuck in a situation where it constantly crosses it's own path to connect to the nearest node thus producing the worst possible solution for the given instance. This is a characteristic downfall that is quite common in many greedy heuristics due to the short term framing of the greedy decisions being made.

Arc-greedy Heuristic.

The arc-greedy heuristic was first introduced by Papadimitiou and Steiglitz [20] as a modification of a process first seen in a 1968 paper by Steiglitz and Weiner [21]. The heuristic is a more complex greedy-type TSP heuristic where all edges of the graph are sorted from shortest to longest. Edges are then added to the tour starting with the shortest arc as long as the addition of this arc will not make it impossible to complete a tour. Specifically, this means avoiding adding edges that make early cycles, and also avoiding creation of vertices of degree three. This process,

as originally proposed, required $O(N^2 \log N)$ time. However, Bentley was also able to speed up this process to $O(N \log N)$ [9] in a paper introducing his Multi-Fragment (MF) version. This yields a similar run time to NN while maintaining a similar worst case solution quality. Arc-greedy's tour construction methodology causes the heuristic to only produce a single solution for each instance where NN can arrive at different solutions based on starting point. This is one of the shortcomings of arc-greedy when related to NN; the failure to generate variability in the output tour. On average though, the arc-greedy heuristic tends to outperform NN in tour quality on a instance to instance basis[9], however there are problem instances where the arc-greedy heuristic is significantly outperformed by NN as the scope of the arc-greedy, considering all arcs at any instance, is inherently more greedy than NN, whose decision is bound to a single node. Thus, the arc-greedy heuristic can create situations where the final arcs needed to connect the various fragments into a single tour are of poor quality.

Recursive-Selection Heuristic.

Taking the arc-greedy shortcoming into account, Okano et al [16] introduced a new heuristic known as the Recursive-Selection Heuristic (RS). Rather than sorting all arcs by length, the RS sorts all points by order of the distance between each point and its nearest neighbor and iterates through the list adding points as long as they do not create a degree of three or early cycles. Once it has iterated through the list, if any points still have a degree of one or zero, it will resort the list with the closest available nearest neighbors and iterate through again. No runtime performance was given for the RS, but the RS+2-Opt meta heuristic designed by Okano steadily outperformed the MF+2-Opt through many of the instances tested in [16]. This RS heuristic motivates one of the central research questions of this paper "What is the

best way to order the greedy decisions made when solving the TSP?" It appears that modifying the decision framework can change how well a greedy heuristic performs.

2.6 Greedy-type Construction Heuristic Modifications

Minimizing the Variance of Distance Matrix Greedy.

A recent modification to the arc based greedy heuristic utilizes work from a 1970 paper by Held and Karp [22] to produce an arbitrary real vector, π, which transforms the distance matrix D to D' by stretching and manipulating the distances between each node [10]. In general, the optimal tour of both distance matricies are the same. This allows for the possibility of finding a vector π such that when the arc based greedy heuristic is implemented on D' a better solution is produced versus when the same heuristic runs on D. Further research by Wang et al [10] showed that the performance of the arc based greedy heuristic was significantly negatively correlated to the variance of D'. This motivates the remainder of the paper, finding a vector π that minimizes the variance of the distance matrix D', thus producing better greedy solutions. The authors identify the fact that minimizing the variance of the distance matrix mitigates a key disadvantage of the arc-based greedy heuristic; the disadvantage being that the last few edges added are often very inefficient due to the non-forward-thinking, greedy nature of the methodology.

Greedy with Regret.

A greedy heuristic with regret modifies a greedy heuristic so that it may reconsider past decisions to possibly improve the final solution. Hassin and Keinan applied this methodology to the TSP utilizing the Cheapest Insertion Heuristic [23]. Adding regret allows the greedy heuristic to correct one of its biggest faults, selecting the best decision at the present moment with no regard to what happens to future moves. Has-

sin and Keinan create a deletion step which allows the heuristic to delete a previously added edge to the sub-tour if it is more expensive than the current decision.

2.7 Meta-Heuristics

This section discusses three meta-heuristics that can incorporate greedy type elements into their solution methodologies. As discussed earlier, a meta-heuristic combines both constructive and, sometimes multiple, local-search heuristics with tunable elements to achieve near, if not optimal, solutions.

Simulated Annealing.

Simulated annealing (SA) is a local search type heuristic, modeled after the annealing process that occurs in metal and glass making. The heuristic was first introduced in 1953 by Metropolis et al [24] as a numerical simulation. This heuristic was then applied to specific combinatorial optimization problems in 1983 by Kirkpatrick et al [24], and finally the TSP, two years later in a paper by Cerny [25]. Additional tunable elements and advantages were added to a later iteration by Eglese [26] who noted that the crux of SA was the ability to tune its temperature parameters to probabilistically accept worse solutions in order to avoid the heuristic getting stuck in a local minima. This is accomplished by a 'temperature' control parameter that assigns a probability to accepting a worse solution when considering any neighbor solution. Generally, SA starts with a warm temperature, corresponding to a high probability of accepting worse neighboring solutions, and cools over time. Reheating functions can be applied so that the heuristic can climb out of local minima. The biggest weakness of SA is the difficulty in tuning the heuristic to different instances, the proper stopping criteria, the proper set of neighbors, and the fact that ideal heating and cooling functions can change drastically from instance to instance.

Genetic Algorithm.

Genetic Algorithms (GA) are modeled after the evolutionary process. This idea was first conceived in 1950 by A.M. Turing [27]. He came up with the following list of connections which he believed could be incorporated into a computerized process modeling hereditary evolution.

- Structure of the child machine = hereditary material,
- Changes of the child machine = mutation,
- Natural selection = judgment of the experimenter.

D. B. Fogel [28] first applied this methodology to the TSP in 1988. The GA follows a Darwinian "Survival of the Fittest" type mentality by first generating a random initial population. A percentage of the population is then selected and evolved through mutation and/or reproduction. This continues until a set termination criterion is met and the newly created individuals are then evaluated against a fitness parameter. A new population is generated from individuals with a specified fitness level and the population is once again evolved. Generally, GAs perform very well due to their ability to explore many solutions simultaneously and identify quality schema utilizing a concept known as intrinsic parallelism. Reeves describes schema as a "subset of a space in which all the strings share a particular set of defined values [29]. In the case of the TSP, schema may be tours that have a certain number of common values in a row. For example, if we have a 10 node TSP, the group of tours that have a common connection of 3-4-5-6 would be a schema. If those connections are efficient and occur in many of the higher fitness population a GA identifies the string as a quality schema. Intrinsic parallelism is the idea that information on many schemata can be processed in parallel [30]. The difficulty of GAs in relation to the TSP is that special precautions have to be taken to ensure that mutations do not cause incomplete tours. Multiple

methodologies ensuring feasibility of solutions via mutation and combinations of tours are discussed by Merz and Freisleben [31].

2.8 Lin-Kernighan Algorithm

The Lin-Kernighan (LK) heuristic was published in 1972 [32]. Various iterative improvements have been made to the LK since its conception, some of the most recent advances can be found in a paper by Rego et al [6] documenting LK variants as well as state-of-the-art data structures which play a key role in many of the improvements. The core of this heuristic involves an adaptive k-opt swap methodology that allows for a variable number of swaps to occur at each iteration.

2-Opt.

An example of a k-opt swap, the 2-OPT routine incrementally considers pairs of arcs for a swap. In order to perform a thorough local search, the 2-OPT routine increments through each node along the tour and considers all possible arc pair swaps at that point. One methodology for performing such a swap is to replace the intermediate tour between two nodes with its reverse order. If the swap is shown to reduce to total tour cost, the swap is saved (but not executed) and compared against all other swaps in the current iteration. At the end of the iteration if an improvement has been saved, the improved tour is executed and becomes the new tour, and the process starts over. Generally k-Opt methodologies need to have a good starting solution to be effective. One of the best starting solutions for a 2-Opt is the arc-greedy heuristic [16]. Thus one popular methodology is the arc-greedy+2-Opt. Pseudocode for this process can be seen in Algorithm 1.

Algorithm 1 Arc-Greedy+2-Opt Pseudocode

```
1: Initialize Variables
2: Generate arc-greedy tour
3: BestCost & SaveCost = arc-greedy tour cost
4: BestTour & SaveTour = arc-greedy tour
5: while Stop <1 do
6:     i = 0
7:     while i <Size-1 do
8:         i = i + 1
9:         j = i + 1
10:        while j <Size do
11:            TESTtour = replace tour between i and j with reverse
12:            Calculate TESTtourCost
13:            if TESTtourCost <BestCost then
14:                BestTour = TESTtour
15:                BestCost = TESTtourCost
16:            end if
17:            j = j + 1
18:        end while
19:    end while
20:    if BestCost <SaveCost then
21:        SaveTour = BestTour
22:        SaveCost = BestCost
23:    else
24:        Stop = 1
25:    end if
26: end while
```

Concorde.

The Concorde is a heuristic LP-type solver designed by Applegate et al [5] that incorporates various separation routines into a primary cutting-plane loop. It orders the routines by rough estimates of their computational requirements. Utilizing a controller type program cuts from a routine are added to the LP relaxation and the problem is solved. If the LP bound for the entire round of cutting planes is above a threshold value, the round is broke off, column generation is applied, and the code returns to the start of the loop. If the total improvement is less than the threshold, additional cuts from the next separation routine are added and the problem is solved

again. This continues until a total improvement bound is less than a designated threshold.

III. Arc-Greedy Subtour Elimination Methodologies

This chapter provides detailed explanations, examples, and pseudo-code for two known sub-tour elimination methodologies for the arc greedy TSP constructive heuristic as well as a third novel sub-tour elimination method. The arc based greedy heuristic gradually constructs a TSP tour by adding to the tour the shortest arc available at each iteration that does not cause a node to have a degree of more than 2 (see Figure 6). However, this degree constraint alone does not prevent sub-tours. The heuristic must also verify that a tour of less than size N, a premature partial circuit, called a sub-tour is not created. For example, consider the following tour construction utilizing an arc greedy constructive heuristic methodology on a 5 node TSP instance. After

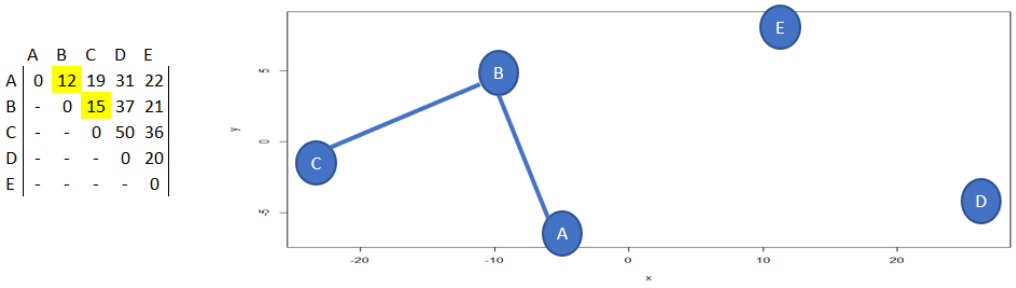

Figure 6. Greedy Subtour 1

adding the first two shortest arcs A-B and B-C, we can see from the distance matrix that arc A-C is the next shortest and still ensures that all nodes in the graph do not exceed a degree of 2. However, adding this arc creates a sub-tour, which would prevent the heuristic from ever constructing a feasible TSP tour (see Figure 7. There are two known methodologies for preventing sub-tours while using an arc greedy heuristic, namely Bentley's Multi-fragment method [9], and an exhaustive loop test. This paper introduces a third novel method for eliminating sub-tours while using an arc greedy constructive heuristic. Each of the following methodologies were reproduced in R adhering strictly to the source descriptions and pseudocode.

	A	B	C	D	E
A	0	12	19	31	22
B	-	0	15	37	21
C	-	-	0	50	36
D	-	-	-	0	20
E	-	-	-	-	0

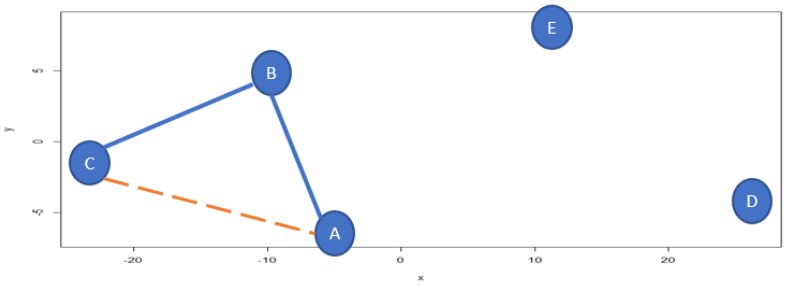

Figure 7. Greedy Subtour 2

3.1 Exhaustive Loop

The Exhaustive Loop (EL), is a methodology for preventing sub-tours while using the arc greedy constructive heuristic. This method is not well documented in academic literature but is often simply referenced as "the standard way." A literature review yielded no scholarly articles on this methodology. EL exhaustively cycles through every edge connected to the most recently added edge. Once a edge e_{ij} is added to the partial tour, node i will be identified as the "start node" and node j will be set as "current node." A trace along the current partial tour then begins. At each step of the trace the "current node," node j, is checked to see if it is connected to another node k via edge e_{jk} in the partial tour. If it is, then node k becomes the new "current node." If the trace returns back to the "start node" in under N steps. Where N is the number of nodes in the instance, then the added edge e_{ij} has created a sub-tour and is an illegal edge. If no edge leaves the "current node" the addition of edge e_{ij} is valid and the current portion of the tour is still a fragment. Each time a edge is added, a count is incremented and the process continues until N-1 edges have been added upon which the last two endpoints are then connected.

When applied to the earlier example, after adding edge A-C the heuristic identifies node A as the starting node and Node C as the current node (seen in Figure 8). The heuristic then looks at Node C and sees it has a degree of 2 and finds the other

	A	B	C	D	E
A	0	12	19	31	22
B	-	0	15	37	21
C	-	-	0	50	36
D	-	-	-	0	20
E	-	-	-	-	0

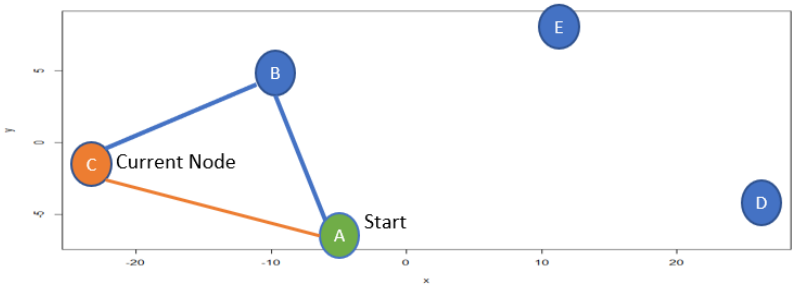

Figure 8. EL Subtour 1

connected arc C-B. Node B becomes the current node and the heuristic verifies that the current node is not the same as the start node (seen in Figure 9). Once again, the

	A	B	C	D	E
A	0	12	19	31	22
B	-	0	15	37	21
C	-	-	0	50	36
D	-	-	-	0	20
E	-	-	-	-	0

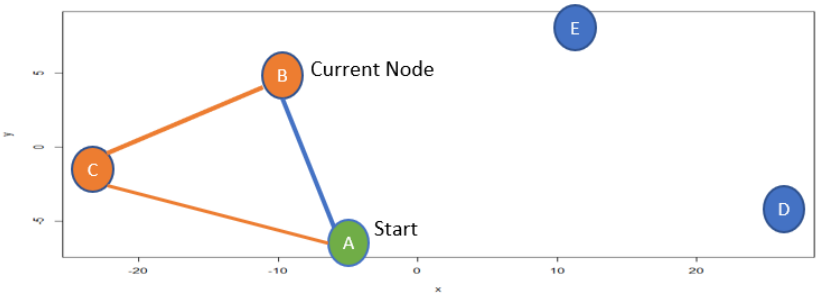

Figure 9. EL Subtour 2

heuristic looks at the new current node, Node B, and identifies that it has a degree of 2 and finds the other connected arc B-A, and updates the current node to Node A. This time when the heuristic checks the current and start node, it realizes they are the same (Figure 10). It then sees how many edges have been added to the tour. Since the number is less than N, the heuristic marks that a subtour has formed and that arc C-A is not valid. Pseudocode for this methodology can be found in Algorithm 2.

Directional vs. Non-Directional.

The methodology above can be described as non-directional, where the direction of travel for each arc does not matter during tour construction. This methodology can only be used with symmetric TSP instances where the distance to travel from node to

Algorithm 2 Exhaustive Loop Pseudocode

```
Initialize Variables
Sort edges: Shortest to Longest
while Nodes.Visited <Size-1 do
    if Both nodes of current edge have degree <2 then
        Set Start = Tail of current edge
        Set Current = Head of current edge
        while Continue = True do
            if Current is Tail to Another Edge then
                Set Next Node = Head of found edge
                if Next Node = Start then
                    Subtour Formed — Remove Edge
                    Continue = False
                else
                    Current = Next
                end if
            else
                Continue = False
                Set edge as part of tour
                Nodes.Visited = Nodes.Visited + 1
            end if
        end while
    end if
    Next Edge in List
end while
Connect Hamilton Path
```

	A	B	C	D	E
A	0	12	19	31	22
B	-	0	15	37	21
C	-	-	0	50	36
D	-	-	-	0	20
E	-	-	-	-	0

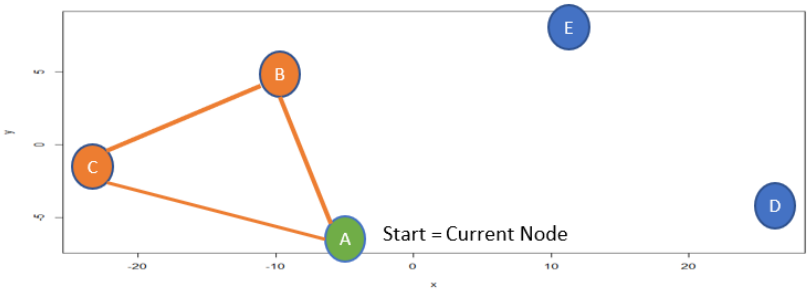

Figure 10. EL Subtour 3

node is equal in both directions. This poses some computational advantages as only $n * (n + 1)/2$ arcs need to be initially sorted. The EL can also be modified to handle a directional methodology which can be used on both symmetric and asymmetric instances when the direction of the arcs is either of importance to the final solution and/or takes different distances to travel in each direction. In this directional scenario, all arcs of each direction $n^2 - n$, are sorted from shortest to longest and rather than tracking the total degree of each node, the connections are split into a T (To) and F (From) array. Utilizing these data structures ensures that each node is only entered once and left once ensuring a continuous direction throughout the tour.

3.2 Multi-Fragment

The Multi-Fragment heuristic described in Bentley's [9] paper utilizes a unique non-directional methodology for eliminating subtours by focusing only on the ends (tails) of each tour fragment. The following structures are utilized in this methodology:

- An array, Degree, that keeps track of each nodes degree
- An array, Tail, that keeps track of the opposite tail of each fragment

All nodes are initialized as their own tail and given a degree of zero when the heuristic begins. As each arc is added, the tails of the nodes and fragment ends are updated.

While Bentley's paper and pseudocode made no mention of how to update these tails, through testing, four possible scenarios were identified.

The first scenario is that the degree of both nodes of the added edge are 0, which is the same as 2 nodes being connected to form a new fragment. In this case, the heuristic sets the tail of each node equal to the node at the opposite end of the edge. Continuing with the 5 node example, this type of update occurs when the first edge is added. As seen in figure 11, when fragment A-B is added the tails for each node

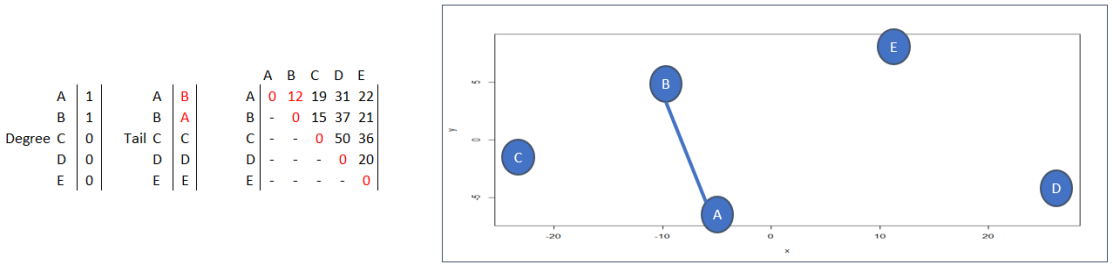

Figure 11. MF Subtour 1

simply becomes the other node, and the degree of each is incremented. With respect to the graph, this scenario is just connecting two nodes.

The second and third scenario are fundamentally the same and occur when an added edge has one node with a degree of 1 and the other node has a degree of zero (for coding purposes they are separate scenarios dependent on which node node has a degree of 0 and which node has a degree of 0). With respect to the graph, this scenario is synonymous with a node being connected to a existing fragment. Figure 12 shows this scenario as node C is connected to the fragment made up of A and B. Node B's degree is updated to be blank indicating that it is in the middle of a fragment. To update the other tail values, the heuristic must reference the tail B, which was A, and update it to show a tail of C, and then update the tail of C to what the tail of B previously was, or A.

The final scenario for updating the tails occurs when two fragments are being

A	1
B	2
Degree C	1
D	0
E	0

A	C
B	-
Tail C	A
D	D
E	E

	A	B	C	D	E
A	0	12	19	31	22
B	-	0	15	37	21
C	-	-	0	50	36
D	-	-	-	0	20
E	-	-	-	-	0

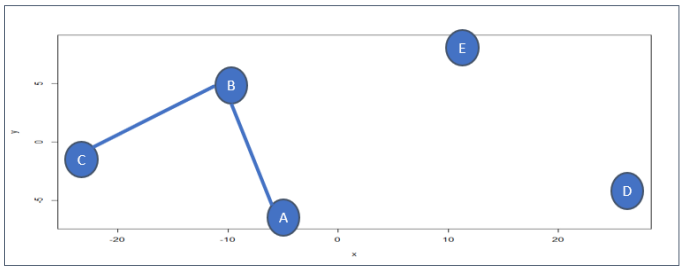

Figure 12. MF Subtour 2

connected by a new edge. See Figure 13, adding edge A-E utilizes a methodology

A	2
B	2
Degree C	1
D	1
E	2

A	-
B	-
Tail C	D
D	C
E	-

	A	B	C	D	E
A	0	12	19	31	22
B	-	0	15	37	21
C	-	-	0	50	36
D	-	-	-	0	20
E	-	-	-	-	0

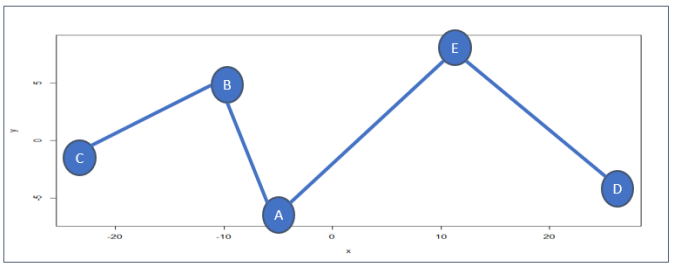

Figure 13. MF Subtour 3

where the tail of each node that makes up the edge must have its tail's tail updated to be the value of the opposite nodes tail. So in this case, node A's tail, which was node C, must have its tail value updated to the tail value of node E, which is node D. The same updating must occur in respect to the other end of the fragment. Pseudocode for MF is included in Algorithm 3.

The description and pseudocode above depicts a non-directional methodology on a symmetric instance for constructing TSP tours using Benteley's MF heuristic. It is possible to modify this methodology to function directionally on both symmetric and asymmetric instances. To do this, the degree array would be split into a To and From array as described when converting EL to a directional variant (see Section 3.1).

Algorithm 3 Multi-Fragment Pseudocode

```
1: Initialize Variables
2: Sort edges(i,j): Shortest to Longest
3: while Nodes.Visited <Size-1 do
4:     if both nodes of current edge have degree ¡ 2 & Tail[i] is not j then
5:         Add edge(i,j)
6:         if Degree[i]=0 & Degree[j]=0 then
7:             tempTaili = Tail[i]
8:             tempTailj = Tail[j]
9:             Tail[i] = tempTailj
10:            Tail[j] = tempTaili
11:        else if Degree[i]=1 & Degree[j]=0 then
12:            tempTaili = Tail[i]
13:            Tail[tempTaili] = Tail[j]
14:            Tail[j] = tempTaili
15:            Tail[i] = 0
16:        else if Degree[i]=0 & Degree[j]=1 then
17:            tempTailj = Tail[j]
18:            Tail[tempTailj] = Tail[i]
19:            Tail[i] = tempTailj
20:            Tail[j] = 0
21:        else if  Degree[i]=1 & Degree[j]=1 then
22:            tempTaili = Tail[i]
23:            tempTailj = Tail[j]
24:            Tail[tempTaili] = tempTailj
25:            Tail[tempTaili] = tempTailj
26:            Tail[i] = 0
27:            Tail[j] = 0
28:        end if
29:        Degree[i] = Degree[i] + 1
30:        Degree[j] = Degree[j] + 1
31:        Nodes.Visited = Nodes.Visited + 1
32:    end if
33:    Next Edge in List
34: end while
35: Connect Hamilton Path
```

3.3 Greedy Tracker

The first original contribution this thesis makes is through the introduction of a novel way to track the progress of the arc greedy construction heuristic, and ensure sub-tours are not created. This new method is called the"greedy tracker" (GT). Conceptually, the GT serves as a methodology to track a nodes communication with other nodes when constructing a TSP tour. While GT can operate on both symmetric and asymmetric instances, it is conceptually easier to visualize the GT using its directional methodology on a symmetric instance and then generalizing the process for asymmetric instances or to the non-directional methodology on symmetric instances. Because of this, the following introduction to the GT utilizes the directional methodology on a symmetric matrix and is accomplished using the following structures:

- X = binary n by n matrix of x_{ij}
- F = binary n by 1 array of f_i
- T = binary n by 1 array of t_i
- $x_{ij} = 0$ if arc from i to j is eligible, greater than 0 if not eligible
- f_i = binary for whether node i has been left
- t_i = binary for whether node i has been entered

These structures track each move, and in doing so, prevent Hamilton cycles and sub tours. The process by which this is accomplished can be seen in Figure 14:

The X (identity), F (From), and T (To) structures can be seen above on the left and a distance matrix from the TSP can be seen on the right. The 1s loaded on the diagonal of the X matrix (where i=j) signal that these moves are ineligible. Note that the diagonal on the distance matrix has been colored red to correspondingly show these ineligible arcs. Looking at the distance matrix it can be seen that the shortest arc is either from A to B or vice versa, thus arc A to B is selected. The X,

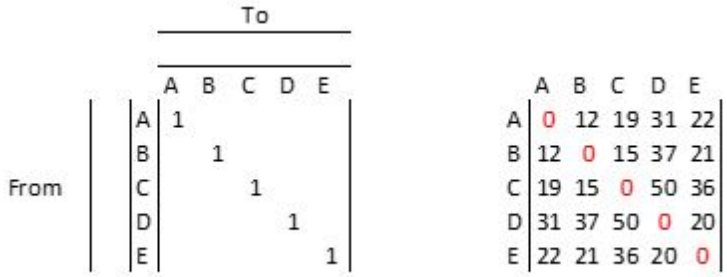

Figure 14. Greedy Tracker 1

F, and T matrices are updated with 1s to indicate this move, this is shown in Figure 15.

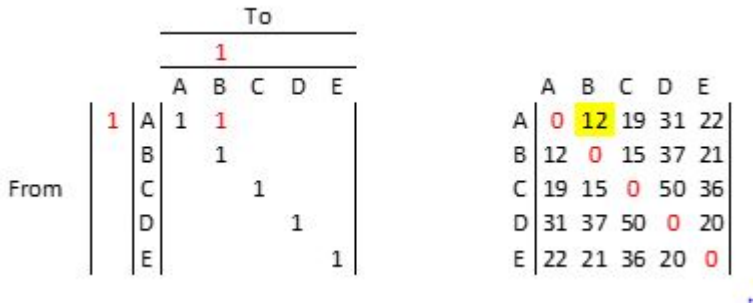

Figure 15. Greedy Tracker 2

Then, the column of the X matrix associated with the new arc is processed. Every row where a 1 appears is combined with the From row of the created arc. Figure 16 illustrates this operation. As seen in Figure 16, since Row 2 has a 1 in the same

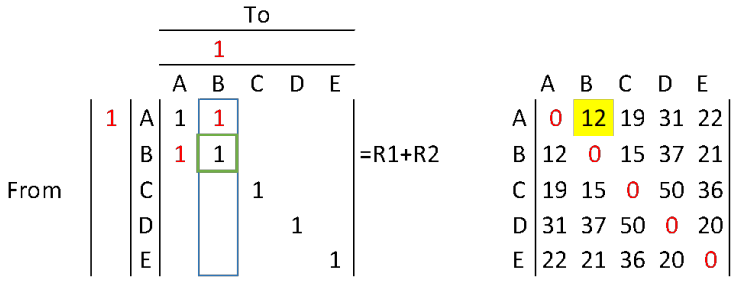

Figure 16. Greedy Tracker 3

column as our new arc, the two rows were combined so that any 1s that were in the Row 1 are now also in Row 2. Note that for the example we only show values of 1 so

as to not detract from their purpose of referring to an ineligible move, however in the code the values in each row will actually be added and values of greater than 1 will appear. For ease of reference in this example ineligible values in the distance matrix are turned red (Figure 17). As can be seen in Figure 17, distances that correspond

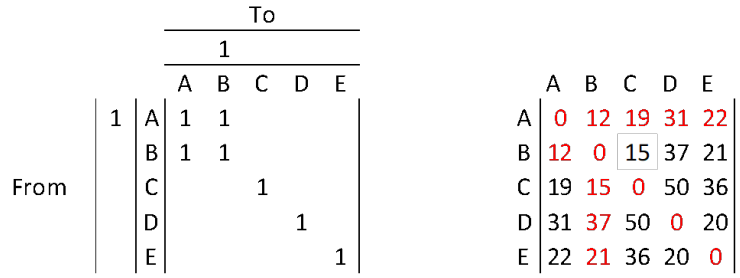

Figure 17. Greedy Tracker 4

with a 1 in the X matrix have been made ineligible moves. Note that any row or column that has a 1 in the T or F arrays will also be marked as an ineligible move. This information will be utilized in the first step of the next iteration where the shortest available arc is identified. As seen in Figure 18, the shortest available arc is B-C and once again the X,F, and T matrices are updated with 1s to indicate the move. Once again the column of X associated with the "To" node of the new arc is

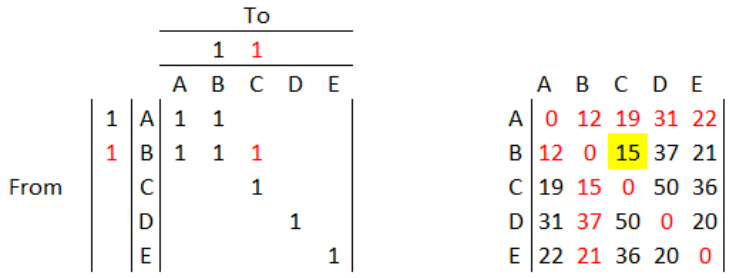

Figure 18. Greedy Tracker 5

processed and every row where a 1 appears is combined with the "From" row of the created arc which can be seen in Figure 19. All the distances that correspond with a 1 in the X matrix are marked as ineligible moves in the distance matrix, as well as any distances associated with a 1 in the T and F arrays. The resulting step can

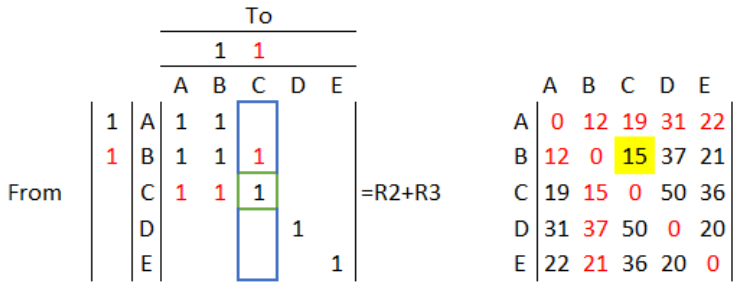

Figure 19. Greedy Tracker 6

be seen in Figure 20. The red in the distance matrix indicates that adding arc A-C

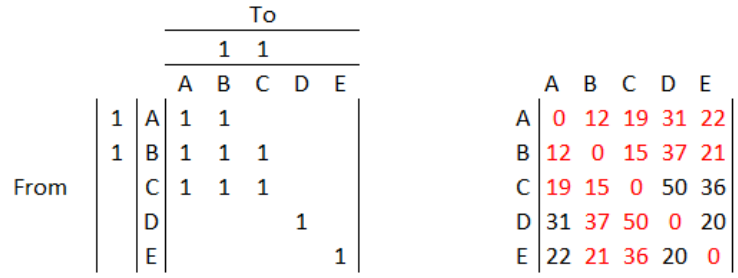

Figure 20. Greedy Tracker 7

is no longer possible because node C already has an edge entering it. This process thus removes the formation of the sub-tour shown earlier. The process shown above continues to iterate until all nodes have been visited which creates a Hamiltonian Path. The final connection to complete the tour is made using the T and F arrays as each should have one index that is still empty. Pseudo code for this methodology is in Algorithm 4.

This methodology can also be used on asymmetric instances as described, or may also be modified to handle a Non-Directional methodology for symmetric TSP instances. For this methodology, the T (To) and F (From) arrays are changed to a Degree array similar to the one used in MF. The row addition loop must also occur twice, once for every 1 in the column of the added edge (i, j), and once for every 1 in the column for the opposite edge (j, i). This nuance makes the GT quite inefficient

Algorithm 4 Greedy Tracker Pseudocode

```
1: Initialize Variables
2: Sort edges(i,j): Shortest to Longest
3: while Nodes.Visited <Size-1 do
4:     if To[j]=0 & XMatrix[i,j]=0 then
5:         Nodes.Visited + 1
6:         XMatrix[i,j]=1
7:         From[i]=1
8:         To[j]=1
9:         for k = 1 to size do
10:            if XMatrix[k,j]=1 then
11:                XMatrix[k,]=XMatrix[k,]+Xmatrix[i,]
12:            end if
13:        end for
14:        Next Edge in List
15:    end if
16: end while
17: Connect Hamilton Path
```

when utilized non-directionally as it doubles the computational time.

GT Improvements.

Certain adjustments to the GT methodology can be made to reduce the total number of operations that occurs within each iteration. These adjustments involve removing the addition of values in specific columns and rows as each node has been left and entered. This process decreases the dimensionality of the GT as the tour is constructed. This is possible because once a node has been entered, or left, no more arcs may enter that node or leave that node. Therefore, it is unnecessary to track what arcs could produce a sub-tour by entering or leaving that node. Consider the the same 5 by 5 instance used earlier, after completing row additions after adding arc A-B, column B can be deleted. Figure 21 shows the resulting GT and distance matrix. As can be seen, all moves in column B, or to Node B, are in eligible because Node B already has an arc entering it. Therefore, it is unnecessary to track and conduct row additions in this column. When working with a non-directional instance

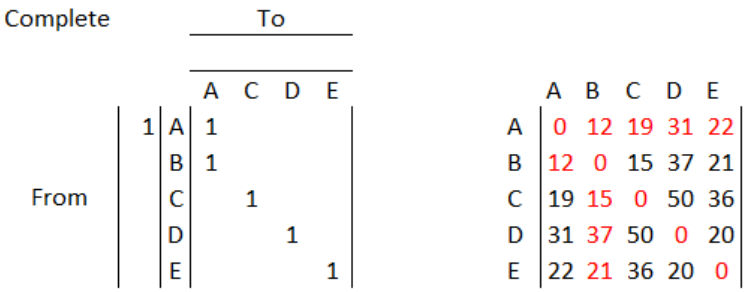

Figure 21. GT Row Delete 1

a column would be deleted after the node had a degree of 2.

R struggles to re-dimensionalize matrices in an efficient fashion. Thus, modifications to this methodology were made. Breaking down the process of the row and column delete methodology in greater detail yields the realization that only one necessary value, the tail of the current fragment, is being transferred to a new node. The GT is thus very similar to Bentley's MF. When the diagonal is populated with 1s, the X matrix is initializing all nodes as their own tail and for the remainder of the tour construction the tail is passed as fragments are connected. In the case of the Directional GT only one value is passed because a directional fragment can only reattach to itself in one direction. This is why the non-directional GT requires two sweeps as opposed to the directional GT's one. Consider the example below on the modified GT. Figure 22 shows a similar initialization to the original GT with the exception that the added arc is no longer reflected in the X matrix. After this step

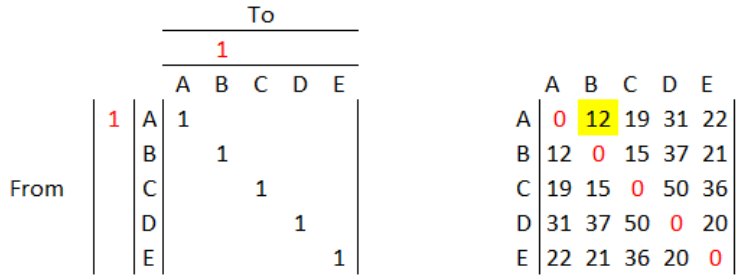

Figure 22. GT modified 1

is performed the "To" column of the arc is scanned for a 1 that coincides with an

empty, or 0 value in the "From" array. Figure 23 shows that this occurs in row B. The next step in the iteration is to find the value in row A that coincides with an

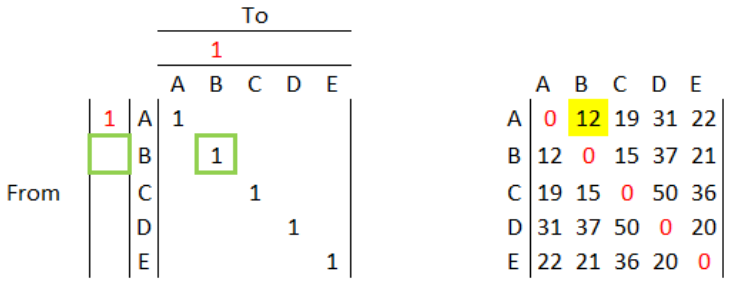

Figure 23. GT modified 2

empty value in the "To" array. Figure 24 shows that this value occurs at A. Thus

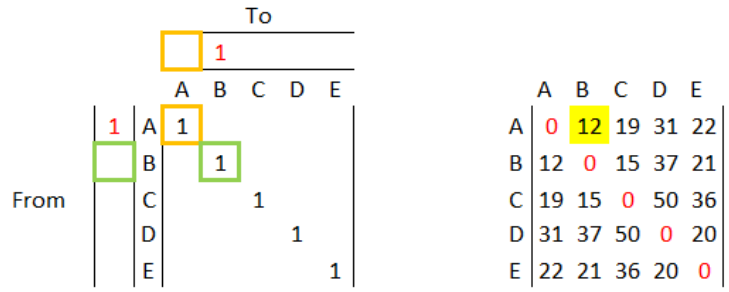

Figure 24. GT modified 3

the next step is to set the intersection of the column identified in the previous step to the row identified directly before to a value of 1. In this case, the intersection of row B and column A is set to 1 as seen in Figure 25. In this first iteration the tail of A,

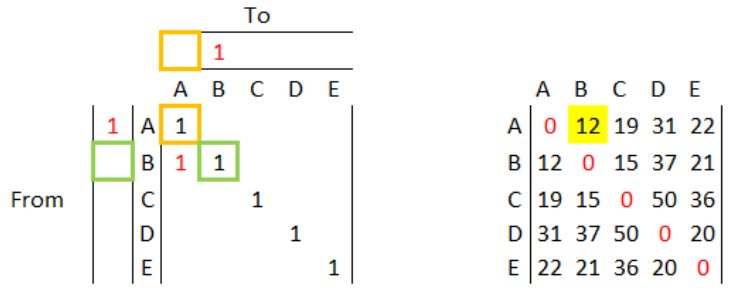

Figure 25. GT modified 4

which was itself, is passed to B, exactly as it would have been in the MF heuristic.

This process continues until a Hamiltonian path is formed. The pseudocode for this modified GT is in Algorithm 5.

Algorithm 5 Greedy Tracker modified Pseudocode

```
1: Initialize Variables
2: Sort edges(i,j): Shortest to Longest
3: while Nodes.Visited <Size-1 do
4:     if To[j]=0 & XMatrix[i,j]=0 then
5:         Nodes.Visited + 1
6:         XMatrix[i,j]=1
7:         From[i]=1
8:         To[j]=1
9:         Row = Intersect(which(X[,j]==0,which(From==0))
10:        Column = Intersect(which(X[,j]==0,which(From==0))
11:        XMatrix[Row,Column]=1
12:        Next Edge in List
13:    end if
14: end while
15: Connect Hamilton Path
```

IV. Greedy Sub-tour Elimination Results

This section covers the TSP instances used, and testing methods employed, along with results from all three of the sub-tour elimination methodologies demonstrated in the prior chapter.

4.1 TSP Instances

TSP data for multiple instances and variations is available in an online library, TSPLIB, maintained by Ruprecht-Karls-Universitat Heidelberg located in Baden-Wurttemberg, Germany. The data from TSPLIP is available via one of two formats in an .atsp file type. A picture of the data's raw format for these files can be seen in Figure 26. The first file type consists of a distance matrix containing a string of distances from node to node for all edges in the instance. However, the file is not properly formatted to be imported into R. To make this file type usable, the information was saved as a text string and then processed to place the information in matrix form. The second file type contains a series of coordinates for each node which can be utilized to form a distance matrix. The distances for every edge can be calculated via a euclidean distance formula (Equation 2) and placed into a distance matrix in R.

$$Euclidean\ Distance\ = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2}$$

The values also must be rounded. TSPLIB provides the best known optimal tours and scores for heuristic testing. For the purposes of this research testing was performed on the instances seen in Table 1, where the alpha prefix is an identifier and the numerical suffix indicates the instance size (in number of nodes).

```
NAME: ftv170
TYPE: ATSP
COMMENT: Asymmetric TSP (Fischetti)
DIMENSION: 171
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: FULL_MATRIX
EDGE_WEIGHT_SECTION
    100000000           9          15          19          39          50
           67          55          75          58          67          85
           99         100          95         111         125         110
          103          91         100         109         123         134
          156         130         143         153         140         126
          144         132         122         121         111          99
          112          84          78          88         100         134
          159         119         125         115         126         137
          150          51          65          80          91         105
          148         127         167         180         167          98
           85         100         188         134         153         132
          119         117         104         115         140          67
           47          35          49          68          79          22
           35          29          22          13          40          82
           94         133         140         127         141         123
          130         106          97          85         115         131
          128          95         114         109         127         143
          131         143          89          82          74          61
           67          49          29          21          39          58
           72          77          90         101         109         127
          137         136         152         165         111          96
           85          78         140         131          91          91
           46          59          69         106         128         138
          118         132         134          85          86         111
          122         176         159         128         142         115
          140         106          91         153         112         127
          107         114         117         115         123         147
          158         122          67         109          76         124
           60          68          45           9   100000000           6
           26          46          57          74          62          82
           65          74          76          90         104         102
          118         132         114          94          82          91
          100         114         125         147         137         134
          144         131         117         135         123         113
          112         102          90         103          75          69
           79          91         125         150         110         116
          106         117         128         141          42          56
           71          82          96         139         118         158
          171         158          89          76          91         179
          125         144         123         110         108          95
          118         131          58          38          26          40
           59          86          13          26          38          31
           22          34          91         103         136         143
          136         150         132         139         115         106
           94         124         140         137         104         123
```

Figure 26. Raw Data Snapshot

Table 1. TSP Instances

TSP Instances	
Symmetric	Asymmetric
bays29	br17
gr48	ry48p
gr51	ft53
berlin52	ft70
pr76	kro124p
kroa100	rgb323
gr120	rgb358
gr130	rgb403
gr195	rgb443
ts225	
pma343	
pcb442	
pr1002	

4.2 Testing

Initial tests verify that each sub-tour method (MF, EL, GT, Modified GT) produced the same tour and distance for all TSP instances. These tests were conducted with both directional and non-directional versions of codes on symmetric TSP instances. In addition, the directional code versions were run on asymmetric TSP instances. Directional and non-directional codes were tested on symmetric TSP instances as they generally produce different solutions, and have different run times due to the number of arcs that must be considered.

Once testing verified each method produced identical greedy tours; that is all directional code variants produced identical tours, and all non-directional codes produced identical tours, the remaining testing focused on computational run-time comparisons. Each methodology was placed in the same arc-greedy heuristic shell so that testing would fairly compare the speed of the three sub-tour tracking and elimination methodologies. Bentley [9] and Wang [10] each utilized advanced computer techniques (k-d trees) and additional data structures to speed up the process of finding the next shortest arc available. However, since neither of these effect the speed of the sub-tour tracking and elimination methodologies they were not utilized.

Speed tests were conducted utilizing the R package "microbenchmark." This package allows testing of multiple R codes simultaneously. Microbenchmark randomly generates run order to handle possible CPU speed fluctuation during testing. The package also reports a variety of statistics to summarize run results. A sample of this output is in Figure 27. Microbenchmark output the minimum, mean, median,

```
Unit: milliseconds
       expr      min       lq     mean   median       uq      max neval
    EL (nd) 30.66862 34.19942 36.05064 35.22033 36.05233  69.70968   100
    MF (nd) 43.48405 46.76142 48.73286 47.54998 48.37478 101.12933   100
    GT (nd) 34.92552 38.35299 39.42242 39.30940 40.07714  69.35652   100
 GT VM (nd) 26.95199 28.95938 30.74206 30.05586 30.76783  65.08831   100
```

Figure 27. Microbench Output

and maximum run-times as well as the lower and upper quartiles. 100 iterations of each code were run to create these summary statistics on 13 different symmetric TSP instances and 9 asymmetric instances. Density plots of runtimes were also reported utilizing the Microbenchmark and ggplot2 R packages, an example of which are in Figure 28. Both symmetric and asymmetric instances were tested to determine if

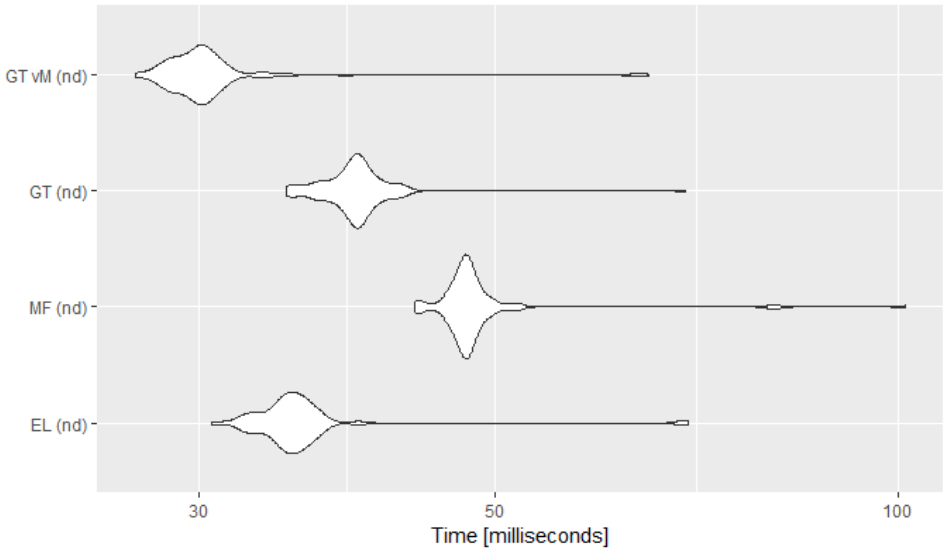

Figure 28. Microbench Output Plot

symmetry effected run time.

4.3 Symmetric Instance Results

Mean run times for a variety of symmetric TSP instances utilizing each of the methodologies can be seen in Table 2. When looking at the directional methodologies, the GT and modified GT tend to be the fastest methodologies on small instances followed by EL and MF. Once instance size reaches around 442, MF takes over as the fastest method for eliminating sub-tours. This largely is due to it's linear growth in operation count as instance size grows. For larger instances, the heuristic is conducting the same number of operations at each step. While the operations are slower for small instances, once the problem becomes larger it proves to be the most efficient.

Table 2. Greedy Sub-tour Methodology Run Times (Symmetric)

Milliseconds	Directional				Non-Directional			
Instance	EL	MF	GT	GT vM	EL	MF	GT	GT vM
BAYS29	23.2	52.0	22.6	22.4	29.7	44.2	29.8	24.4
GR48	25.4	56.1	25.0	25.0	30.5	46.8	33.0	25.8
EIL51	26.7	53.2	24.1	24.4	32.3	50.4	34.8	28.1
BERLIN52	26.6	53.1	24.0	24.1	31.4	47.9	338.0	26.3
PR76	30.8	56.9	29.7	28.5	32.8	46.9	35.8	29.9
KROA100	36.8	60.8	33.4	34.5	36.5	50.4	39.1	30.8
GR120	41.9	66.5	42.4	43.1	40.5	52.2	44.6	34.9
CH130	48.9	70.4	45.6	43.7	43.3	52.9	45.7	35.7
RAT195	76.8	98.6	78.0	73.6	54.5	65.0	61.5	50.3
TS225	97.8	113.9	100.4	95.4	61.8	68.1	86.9	57.3
PMA343	193.7	193.9	193.4	177.0	121.2	111.9	157.4	107.7
PCB442	363.7	312.8	340.2	317.2	177.0	166.1	253.8	180.5
PR1002	1.595(s)	1.308(s)	1.480(s)	1.374(s)	836.3	637.3	1082.3	761.8

We see that the modified GT and original GT tend to perform very similar for smaller instances but once the instance size grows the elimination of the row addition operations in the modified methodology gives the modified GT a computational advantage. The reduction in operations is still not enough to keep GT faster than MF as the searching procure utilized by the modified GT is still a computationally demanding process as instance size grows.

These performance trends are continued when looking at the non-directional code variants applied to these same symmetric instances as seen on the right half of Figure 2, with the exception of the original GT. In the non-directional instance, the dual row sweeping doubles the operations at each step, which gives the modified GT and EL a speed advantage. However, once again MF becomes the fastest methodology from the 442 node instance and larger.

4.4 Asymmetric Instance Results

The directional variants of each sub-tour elimination codes were also run on Asymmetric TSP instances to compare runtimes to determine if any trends changed. The mean runtimes are in Table 3.

There was greater variability between methods for some of the asymmetrical instances. This could be due to the how the edges fluctuate in each direction which

Table 3. Greedy Sub-tour Methodology Run Times (Asymmetric)

Milliseconds	Directional			
Instance	EL	MF	GT	GT vM
br17	22.2	52.5	21.5	21.1
ry48p	24.5	53.9	23.2	23.2
ft53	25.6	54.0	24.2	24.7
ft70	28.5	56.5	27.1	27.4
kro124p	35.9	64.4	33.1	34.6
rgb323	230.3	195.5	220.6	199.1
rgb358	242.7	219.7	231.6	231.3
rgb403	311.9	275.2	303.2	274.5
rgb443	366.5	315.3	350.3	330.5

causes more searching to find edges to complete the tour. Prior overall trends remain, where modified GT is competitive for small to medium sized asymmetric instances, but MF is fastest for larger instances.

4.5 Future Improvements

The portion of the Modified GT most susceptible to computational growth is the search to identify what tail is necessary to move. If this search process growth can be limited, it is possible that the modified GT could outperform MF for larger instances as well. Some possible methodologies to limit computational growth include a better implementation of the row and column delete methodology in conjunction with a new row and column generation methodology. Size of the search operations could also be reduced drastically especially during the early iterations of the arc-greedy heuristic by only generating nodes and tails as needed. This is accomplished by storing a list of indices, call them *Tnodes* and *Fnodes*, for what arcs values are necessary for tail storage and transfer. The following example explains this methodology using the modified GT.

This proposed methodology starts with an empty X matrix. The To and From arrays are populated as usual, but the values searched when a tail is being moved, is limited to the indices contained within the subsets *Tnodes* and *Fnodes*. Therefore, for visual purposes, only nodes within these indicies show their values in the figures,

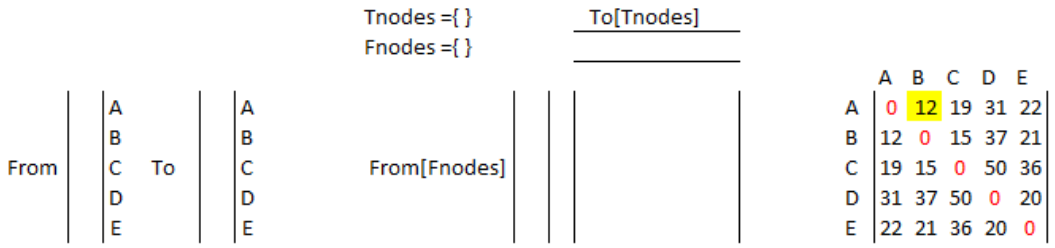

Figure 29. Proposed Future GT 1

represented as To[*Tnodes*] and From[*Fnodes*]. As with previous examples the first arc added is arc A-B. Figure 30 shows A and B are added to *Tnodes* and *Fnodes*, which generates a respective row and column for each to track the tail generated by the addition of the arc. This generation technique is possible because all unconnected

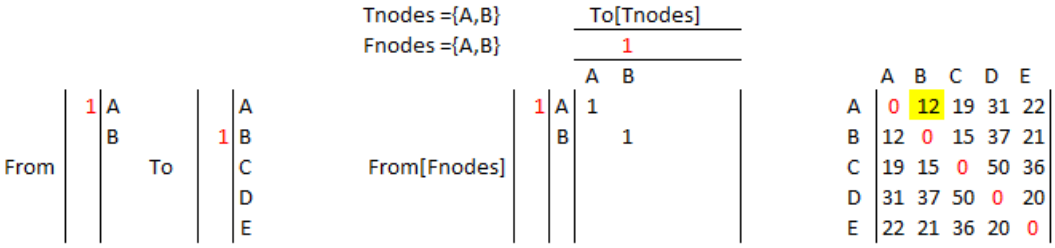

Figure 30. Proposed Future GT 2

nodes initialize as their own tail. Utilizing the Modified Greedy Methodology any 1s in the "To" column of the added arc that correspond with a 0 in the "From" array will identify what row the tail will be transferred to. This is followed by searching the Row associated with the "From" node of the current arc and identifying any nodes in this row that correspond with a 0 in the To array. This step can be seen in Figure 31. Once this step is completed, both rows and columns that correspond 1s in the "From" or "To" are deleted from the matrix and removed from the subsets *Tnodes* or *Fnodes*(as seen in Figure 32). This process could drastically decrease the size of each iterations search larger TSP instances. This methodology along with coding in a more advanced computer language could help GT to maintain is performance gap over MF in larger instances.

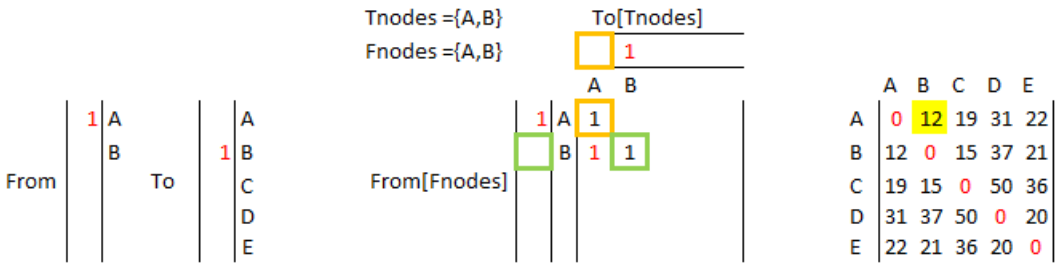

Figure 31. Proposed Future GT 3

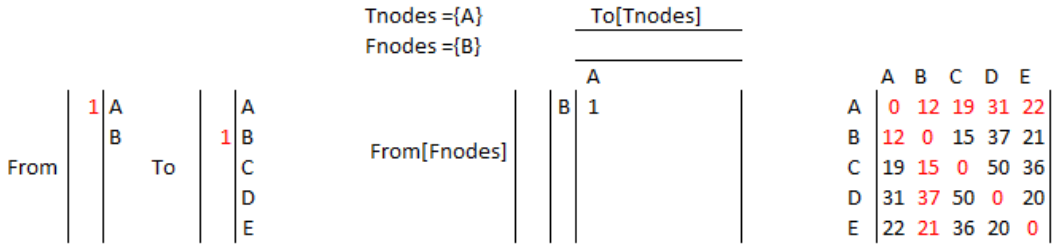

Figure 32. Proposed Future GT 4

V. Ordered-Lists Methodology

This section introduces a novel constructive heuristic called Ordered-Greedy. This is followed by a comparison of tour quality between the ordered-greedy output resulting from 1000 random generated lists, versus viewing the lists as tours. This comparison is performed for 13 symmetric instances, the outcome of which motivates the development of a new meta-heuristic based on Ordered-Lists.

5.1 Ordered-Greedy Heuristic

Given the sub-tour tracking abilities of the aforementioned methodologies, there are some interesting alterations to the arc-greedy heuristic that can be made. One such change is to utilize one of the elements realized by the Recursive Selection heuristic, where the order in which greedy decisions are made is taken into consideration. This concept motivated the development of a novel constructive heuristic called the Ordered-Greedy (OG) heuristic. The OG heuristic is a node-greedy heuristic that takes as input a complete list of nodes. Starting at the top of the list, each node is considered in turn and the available set of choices is limited to the feasible arcs originating at that node. What differentiates the OG from NN, another node-greedy heuristic, is that multiple fragments may exist during the tour construction.

The motivation for this heuristic is to apply a more structured approach to what nodes should be given priority in connecting to their nearest neighbors. Nodes higher in the list have maximum flexibility with minimal concern for node degree or sub-tours and thus typically choose better arcs than nodes later in the list. The quality of the solution found is heavily dependent on the order of the list.

To introduce the methodology of the OG heuristic, consider the following example. In this example an ordered list of D,E,C,B,A has been, through some unspecified

fashion, predetermined. This ordering of the nodes list is reflected in the matrix on the right-side of Figure 33 whose rows are now sorted according to this list order. The constructive heuristic now makes greedy decisions starting at the top of this list and working down. The first greedy decision is made with respect to node D. The

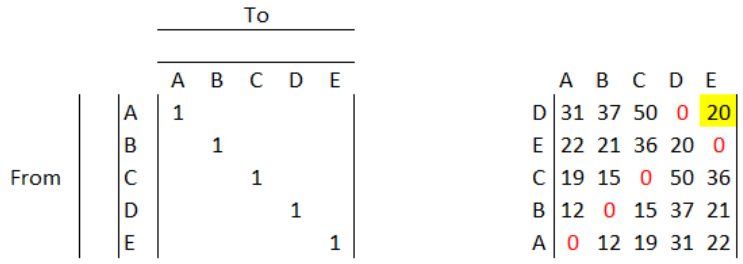

Figure 33. Ordered-Greedy 1

greediest, or shortest arc, from node D is arc D-E as indicated above. This arc and its associated node is tracked by the GT so that the next decision can be made. The next decision is made with respect to node E. This is not due to node E being the head of the previous arc added, but rather because it is second in the provided ordered list: D,E,C,B,A. Looking at the row in the Distance matrix associated with node E along with the GT output that captures ineligible moves (as seen in Figure 34), it can be seen that the shortest legal arc available is arc E-B. This process continues row by

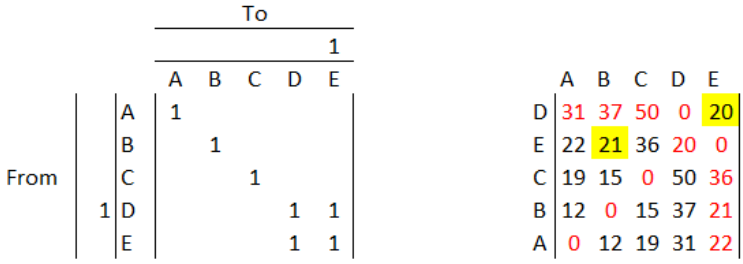

Figure 34. Ordered-Greedy 2

row until the final row is reached which is where the T and F arrays are scanned to find the final legal arc as seen in Figure 35. After adding arc A-D, the resulting tour becomes A-D-E-B-C-A which is also the optimal tour for this TSP instance. This

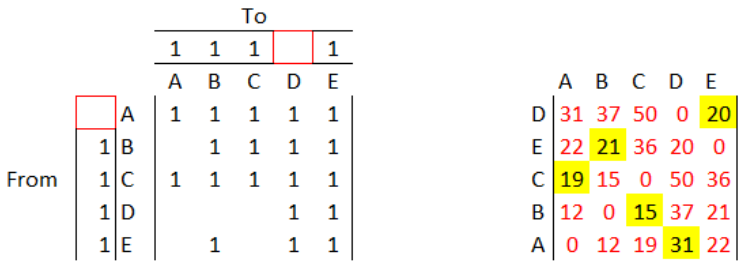

Figure 35. Ordered-Greedy 3

result motivated the creation of the concept of a *Perfect-Ordered List*. Pseudocode for the OG is in Algorithm 6.

Algorithm 6 Ordered-Greedy Pseudocode

```
Initialize Variables
Generate Order
Nodes.Visited = 0
while Nodes.Visited <Size-1 do
    Moves = arcs leaving Order[Nodes.Visited+1]
    Moves[To==True]= Inf
    Moves[XMatrix[Order[Nodes.Visited+1],]]= Inf
    minmove = min(Moves)
    Get First index i of Moves where Moves[i]=minmove
    Add Arc(Order[Nodes.vistied+1],i) to tour
    Track moves with Greedy Tracker
    Nodes.Visited = Nodes.Visited+1
end while
Connect Hamilton Path
```

The OG non-directional and OG directional methodologies yield the same solutions because of how the OG handles connections to nodes that already have a degree of one. If a node attempts to connect to another node with a degree of 1, the connection will only be accepted if that node has already occurred in the order. This is because if the node has not yet occurred in the order, and the connection is allowed, the node will have a degree of 2 before its turn in the ordered-list. Thus, when its turn does come, it would not be able to make a connection. This behavior causes the non-directional to treat these nodes as if they were of the opposite direction, causing

the two methodologies to produce the same solution.

5.2 Perfect-Ordered List

A perfect ordered list (POL) is simply an ordered-list which, when iterated through using the ordered greedy heuristic described above, will yield the optimal solution. Most, but not all, optimal solutions can be associated with a POL (the reasoning for which is discussed later). To find whether a POL exists for a given optimal solution, following methodology based on the GT is used.

First, initialize by identifying all arcs in the optimal solution, and the shortest available Arc (using lowest index to break ties) for each node. Figure 36 shows the completion of this initialization. The next step is to then identify all, so called, Tier 1

Tier		A	B	C	D	E	Min	Arc
	A	0	12	19	31	22	12	A-B
	B	12	0	15	37	21	12	B-A
	C	19	15	0	50	36	15	C-B
	D	31	37	50	0	20	20	D-E
	E	22	21	36	20	0	20	E-D

		To				
		A	B	C	D	E
From	A	1				
	B		1			
	C			1		
	D				1	
	E					1

Figure 36. Perfect-Order 1

nodes. These are nodes where the optimal arc is the same as the shortest arc available. During this iteration (seen in Figure 37), the only arc in the optimal solution that matches its shortest arc is Arc D-E.

Tier		A	B	C	D	E	Min	Opt
	A	0	12	19	31	22	12	A-B
	B	12	0	15	37	21	12	B-A
	C	19	15	0	50	36	15	C-B
1	D	31	37	50	0	20	20	D-E
	E	22	21	36	20	0	20	E-D

		To				
						1
		A	B	C	D	E
From	A	1				
	B		1			
	C			1		
1	D				1	1
	E				1	1

Figure 37. Perfect-Order 2

This move is updated in the GT and the distance matrix is reanalyzed to determine the remaining shortest legal arc available for each node. The second iteration identifies any nodes that now match their shortest available arc with the optimal solution arc. These nodes are labeled as Tier 2 node. The intuition is that Tier 2 nodes derive their optimal arc matches as a result of the greedy decisions made by the Tier 1 nodes. As seen in Figure 38, the Tier 1 move effected the shortest available legal move for node E and is marked as a Tier 2 node. This process continues until either all nodes are

Tier		A	B	C	D	E	Min	Opt
	A	0	12	19	31	22	12	A-B
	B	12	0	15	37	21	12	B-A
	C	19	15	0	50	36	15	C-B
1	D	31	37	50	0	20		
2	E	22	21	36	20	0	21	E-B

			To				
				1			1
			A	B	C	D	E
From		A	1				
		B		1		1	1
		C			1		
	1	D				1	1
	1	E		1		1	1

Figure 38. Perfect-Order 3

given a Tier as seen in Figure 39 or no greediest legal arcs match their optimal arc in an iteration. If the later occurs then no POL exists for the given optimal tour. If the

Tier		A	B	C	D	E	Min	Opt
4	A	0	12	19	31	22	31	A-D
4	B	12	0	15	37	21	15	B-C
3	C	19	15	0	50	36		
1	D	31	37	50	0	20		
2	E	22	21	36	20	0		

			To				
			1	1			1
			A	B	C	D	E
From		A	1		1		
		B		1		1	1
	1	C	1		1		
	1	D				1	1
	1	E		1		1	1

Figure 39. Perfect-Order 4

process does run to completion, then the order of the nodes are sorted with respect to their Tier. In the case of the example provided, the POL would be D,E,C followed by either A,B or B,A.

Order within the tiers does not effect the resulting tour, which reveals an interesting insight into solving the TSP using ordered-lists. More than one ordered list

corresponds to a single tour. Since the total number of permutations of nodes is equal for tours and ordered lists, we can deduce that certain feasible tours cannot be reached via the ordered list solution space. This information is cause for concern as it leads one to question whether the optimal tour is always achievable within the ordered list solution space. Tests on the 13 symmetric instances initially yielded POs for only 8 of the instances. However further testing on the GR48 instance revealed there exists multiple optimal tours. The images in Figure 40 show the difference between two unique optimal tours, one of which (left) cannot be represented by a Ordered List (i.e. cannot be found utilizing the OG Heuristic) and the other (right) can be found by the OG. Similar situations may exist for the larger instances but it

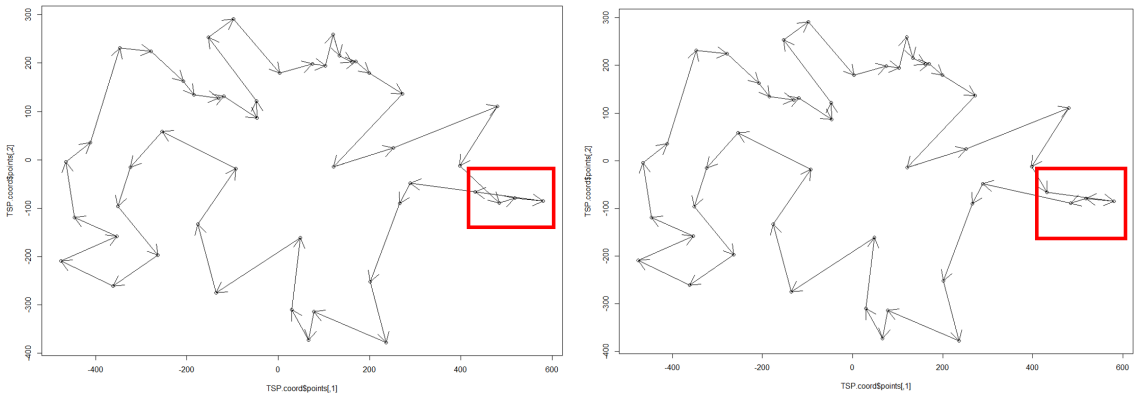

Figure 40. gr48 Optimal Tours

is nontrivial to find additional optimal tours for these large instances to verify. This line of inquiry is left as a question for future research.

5.3 Ordered-Lists vs. Tour Order

Since not all valid tours for an instance have an analog in the Ordered-List solution space, it is important to compare solution quality of each space. Exhaustive testing comparing all tour permutations and all ordered list permutations could only be completed on examples smaller than 10x10. Some 5x5 test instances were generated

by selecting the first 5 nodes, and associated distances between them, for 5 of the symmetric and 2 of the asymmetric instances. Then each of these 5x5 instances had one additional node added to generate the 6x6 instance and so on until the 9 by 9 instance. This provided some comparison as to how the solution space for each instance was effected by the addition of a single node. The summary of the results for these tests are in Table 4.

Table 4. 5 by 5 to 9 by 9 List vs. Tour Comparison

Symmetric		5 by 5		6 by 6		7 by 7		8 by 8		9 by 9	
		Tour	List	Tour	List	Tour	List	Tour	List	Tour	List
eil51	avg	131.5	116.9	155.6	131.4	191.7	156.4	209.4	162.7	241.3	174.7
	max	157	124	190	157	241	196	264	213	316	239
	# opt	10	16	12	84	14	232	16	2940	18	18180
gr120	avg	1526.0	1387.4	1800.0	1603.2	2109.0	1815.7	2301.4	1894.4	2476.0	1981.1
	max	1756	1756	2240	1935	2645	2369	2884	2480	3267	2677
	# opt	10	16	12	84	14	146	16	638	18	3156
rat195	avg	105.5	93.9	142.8	109.9	192.7	141.1	246.9	168.9	309.0	193.3
	max	123	123	184	163	249	202	331	258	410	310
	# opt	20	20	48	96	42	488	176	3386	198	13586
TS225	avg	5000.0	4233.3	7000.0	5422.2	9333.3	6646.0	12000.0	7893.8	15000.0	9153.5
	max	6000	6000	9000	8000	12000	10000	16000	12000	20000	14000
	# opt	40	96	96	504	224	3082	512	21880	1152	177218
PMA343	avg	30.0	25.4	42.0	32.5	56.0	39.9	72.0	47.4	90.0	54.9
	max	36	36	54	48	72	60	96	72	120	84
	# opt	40	96	96	504	224	3082	512	21880	1152	177218
Asymmetric		5 by 5		6 by 6		7 by 7		8 by 8		9 by 9	
		Tour	List	Tour	List	Tour	List	Tour	List	Tour	List
br17	avg	174.5	122.5	170.4	100.5	167.7	94.9	165.1	85.7	163.3	82.1
	max	245	130	274	142	274	142	290	166	295	166
	# opt	20	26	48	124	56	100	128	3894	288	34316
ft70	avg	3595.5	3192.0	4195.2	3666.4	4851.8	4237.0	5667.9	4945.9	6488.0	5328.1
	max	4194	3662	4933	4198	5766	4935	6607	5793	7613	6378
	# opt	5	32	6	86	7	420	8	1176	9	9504

As seen in Table 4, the Ordered-List feasible solutions outperformed Tour feasible solutions in all measures of merit for tour quality, producing shorter average tours, with shorter maximum tour lengths and exhibiting a higher number of occurrences of the optimal solution. This is due to the indifference to order within tiers. So if a PO exists, it appears that the many variants of orders makes for a higher chance of finding an optimal solution through the use of lists.

While it is not computationally possible to exhaustively test the feasible space of larger instances, the spaces can be sampled to see if the trend of the ordered list space providing generally better solutions continues. To this end, 1000 random tours were generated for all 13 large symmetric instances and 9 asymmetric instance considered previously and compared to the performance of solutions generated by their ordered list counterparts. These tests results are in Table 5.

The main takeaway from these results is that in all symmetric instances the tour generated by the random ordered-list out-performed the randomly generated string tours. This indicates that the quality of solutions resulting from ordered-lists are superior to their associated random tours. This is not a surprising discovery as it is more computationally demanding to calculate a tour from an ordered list when compared to calculating the distance associated with a random string tour. However, the quality of the solutions of ordered-list generated tours appears to warrant this additional time.

The results pertaining to asymmetric instances show mixed results when comparing tours generated by an ordered-list and randomly generated string tours. Specifically, the randomly generated string tours either performed relatively equal to or better than ordered-lists for all four rgb instances. This may indicate that this specific instance type. The results of these tests can be seen in Table 6.

These results motivate the development of the Perfect List Random Greedy Search (PLGRS) Meta-heuristic, which seeks to initialize an Ordered-List for a given TSP instance and then improve the solution by making alterations to the list.

5.4 Perfect List Random Greedy Search

PLGRS is a meta-heuristic methodology that focuses on improving an instance tour by randomly searching the ordered-list solution space. Unlike many other meta-

Table 5. Tour Distance Comparisons (Symmetric)

Symmetric		1000 Runs Each	
Size		Tour	List
bays29	mean	5,985.87	2,755.86
	min	4655	2131
	max	7404	3366
gr48	mean	20,983.00	7,014.35
	min	16327	5622
	max	24666	8516
eil51	mean	16,151.13	590.82
	min	1292	483
	max	1934	727
berlin52	mean	29,880.31	11,163.22
	min	23,277	8,416
	max	34,877	13,915
pr76	mean	573,383.10	158,281.70
	min	480,146	133,273
	max	649,969	192,408
kroa100	mean	171,463.70	32,202.41
	min	143,985	27,074
	max	197,441	38,948
gr120	mean	52,195.33	10,235.26
	min	45,651	8,725
	max	59,170	12,796
ch130	mean	46,382.67	9,016.88
	min	41,597	7,725
	max	51,354	11,005
rat195	mean	22,737.86	3,405.78
	min	19,752	2,933
	max	25,597	3,889
ts225	mean	1,594,118.00	208,465.70
	min	1,461,395	185,067
	max	1,717,514	232,953
pma343	mean	36,136.86	2,309.17
	min	32,554	2,004
	max	39,631	2,705
pcb442	mean	773,506.70	79,800.80
	min	724,800	71,967
	max	819,674	90,000
pr1002	mean	6,448,828.00	393,001.10
	min	6,172,884	369,705
	max	6,719,736	432,164

heuristic, PLGRS iteratively utilizes a constructive heuristic (OG) to improve the solution. All variants of PLGRS operate by generating an initial tour utilizing the Non-Directional Greedy heuristic and then deconstructing the tour to generated the associated ordered-list. After this is completed PLGRS seeks to alter the ordered-list to improve the solution. Three versions of PLGRS are described below.

Table 6. Tour Distance Comparisons (Asymmetric)

Asymmetric		1000 Runs Each	
Size		Tour	List
br17	mean	246.77	78.07
	min	60	39
	max	424	409
ry48p	mean	54,760.99	19,374.83
	min	42,406	16,303
	max	65,788	23,059
ft53	mean	26,102.12	11,699.97
	min	20,825	8,812
	max	29,156	14,771
ft70	mean	72,282.37	46,788.46
	min	66,004	42,252
	max	78,146	51,749
kro124p	mean	190,819.40	49,022.63
	min	163,892	42,315
	max	213,777	56,054
rgb323	mean	6,203.35	6,153.28
	min	5,770	5,859
	max	6,557	6,484
rgb358	mean	6,919.96	7,182.20
	min	6,372	6,908
	max	7,428	7,522
rgb403	mean	7,691.37	7,967.02
	min	7,192	7,663
	max	8,078	8,286
rgb443	mean	8,259.27	8,700.73
	min	7,839	8,253
	max	8,734	9,027

PLGRS - Random Swaps.

The first form of PLGRS attempts to improve tour quality by altering the associated ordered list utilizing a random swap of nodes within the order. This swap occurs by simply selecting 2 nodes in the ordered list and swapping their indices giving each a different selection of available arcs in the OG heuristic. The total number of iterations, and the number of swaps that occur at each iteration are tuneable parameters which allows the user to tune the heuristic to the size of the TSP instance. Pseudocode for the Random Swap PLGRS code is in Algorithm 7.

Algorithm 7 PLGRS - Random Swaps Pseudocode

```
1: M = # of iterations
2: n = # of swaps at each iteration
3: tour = Find Arc-Greedy tour
4: bestScore = score
5: bestTour = tour
6: order = Deconstruct Arc- Greedy tour to Ordered-List
7: bestOrder = order
8: for i = 1 to M do
9:     for j = 1 to n do
10:        swap1 = sample(size,1)
11:        swap2 = sample(size,2)
12:        temp = order[swap2]
13:        order[swap2] = order[swap1]
14:        order[swap1] = order[swap2]
15:    end for
16:    Ordered Greedy(order)
17:    if score ≤ bestScore then
18:        bestScore = score
19:        bestTour = tour
20:        bestOrder = order
21:    else
22:        order = bestOrder
23:    end if
24: end for
```

PLGRS - Bad Arc Targeting.

In order to attempt more educated alterations to improve an ordered list, the Bad Arc Targeting (BAT) methodology was conceived. Wang et al [10] indicated that generally the reason that the greedy heuristic performed poorly was due to the final arcs added as they typically were the worst in the tour. This BAT methodology attempts to target these arcs and move their respective nodes higher in the list to improve the solutions. The heuristic works by starting considering only the worst arcs in the present best solution, and then narrowing the scope of neighboring solutions it is considering. After a user specified number of iterations with no improvement this scope expands to include slight better arcs. If an improvement is found then the

scope is reset to only consider the worst arcs in the current solution. The reasoning for this methodology is the worst arcs to the solution may completely change based on a slight alteration to the ordered list. Thus, the heuristic is greedily attempting to fix the worst arcs first and then expanding to consider a growing number of better arcs until another improvement is found. Arcs are identified as "Bad" utilizing tuneable criterion that can change as the Heuristic progresses. The tuneable criterion are:

- $\alpha = x$times min arc value for current node considered "Bad",
- $start_\alpha = \alpha$ value that starts search,
- $change_\alpha =$ amount α decrease by indicated, and
- $intensify_\alpha =$ number of iterations with no improvement before decreasing α.

After the heuristic has generated an ordered-list from the arc-greedy tour, the minimum value arc for each node is found. Bad arcs are identified using α and and the shortest arc available for each node (Equation 3).

$$Bad\ Arc\ Threshold\ =\ (minimum\ arc\ value) + \alpha * (minimum\ arc\ value) \quad (3)$$

The number of bad arcs must be equal to at least 2 to ensure some variety in attempted moves for the period until alpha is adjusted. If less than 2 arcs are considered bad then alpha is decreased by $change_\alpha$. Bad arcs are randomly selected and inserted higher into the ordered-list in an attempt to improve the solution. After a certain number of iterations, $intensify_\alpha$ with no improvement α will be reduced by $change_\alpha$. If an improvement is found, or α has reached 0 for $intensify_\alpha$ iterations, α is reset to the $start_\alpha$. Pseudocode for this process is in Algorithm 8.

Algorithm 8 PLGRS - Bad Arc Targeting Pseudocode

```
M = # of iterations
tour = Find Arc-Greedy tour
bestScore = score of Arc-Greedy tour
bestTour = tour
order = Deconstruct Arc- Greedy tour to Ordered-List
bestOrder = order
minvals = calculate min arc distance for all nodes
for i = 1 to M do
    arcvals = value of current arcs for each node
    haveArc = False
    while haveArc = False do
        badarc = which(arcvals ≥ minvals + minvals*α)
        if length(badarc) <2) OR αcount ≥ intensify_α then
            α = α − change_α
            αcount=0
        else
            movenode = sample(badarc,1)
            havearc = True
        end if
    end while
    Move movenode to random new location in order
    Perform Ordered Greedy(order)
    if score ≤ bestScore then
        if score <bestScore then
            α = start_α
            αcount=0
        end if
        bestScore = score
        bestTour = tour
        bestOrder = order
    else
        order = bestOrder
        αcount = αcount + 1
    end if
end for
```

PLGRS - Bad Arc Targeting & Good Node.

The "Good Node" methodology identifies quality candidate nodes to be moved later in an ordered list. This methodology uses the same alpha parameters used in

BAT but in addition to moving nodes up the list, alpha is also used to generate a list of candidate nodes to move down in the ordered list. This is accomplished by using $Start_\alpha$ to generate a number of arcs within α percent length of the best available arc for each node. Then the nodes with the greatest number of arcs within this threshold will be considered to be swapped with one of the nodes identified by the BAT methodology. Pseudocode for PLGRS- Bad Arc Targeting & Good Node is in Algorithm 9.

PLGRS - ALL.

The last version of PLGRS, PLGRS - All, utilizes all the methodologies described above and randomly selects one methodology to alter the current order at each iteration. Pseudocode for PLGRS-All is in Algorithm 10.

Algorithm 9 PLGRS - Bad Arc Targeting & Good Node Pseudocode

```
M = # of iterations
tour = Find Arc-Greedy tour
bestScore = score of Arc-Greedy tour
bestTour = tour
order = Deconstruct Arc- Greedy tour to Ordered-List
bestOrder = order
minvals = calculate min arc distance for all nodes
for p = 1 to Size do
    goodnodes[p] = length(which(ArcLengths[p,] ≤ minvals[p]*(1/start_α)
    maxgood = max(goodnodes)
end for
for i = 1 to M do
    arcvals = value of current arcs for each node
    haveArc = False
    Perform BAT to identify node to move up
    swap1 = node found by BAT
    movenode = sample(which(goodnodes≥ α/start_α*maxgood),1)
    swap2 = which(order=movenode)
    temp = order[swap2]
    order[swap2]=order[swap1]
    order[swap1]=temp
    Perform Ordered Greedy(order)
    if score ≤ bestScore then
        if score <bestScore then
            α = start_α
            αcount=0
        end if
        bestScore = score
        bestTour = tour
        bestOrder = order
    else
        order = bestOrder
        αcount = αcount + 1
    end if
end for
```

Algorithm 10 PLGRS - ALL Pseudocode

```
M = # of iterations
tour = Find Arc-Greedy tour
bestScore = score of Arc-Greedy tour
bestTour = tour
order = Deconstruct Arc- Greedy tour to Ordered-List
bestOrder = order
minvals = calculate min arc distance for all nodes
for p = 1 to Size do
    goodnodes[p] = length(which(ArcLengths[p,] ≤ minvals[p]*(1/start_α)
    maxgood = max(goodnodes)
end for
for i = 1 to M do
    arcvals = value of current arcs for each node
    haveArc = False
    type = sample(3,1)
    if type = 1 then
        Perform PLGRS Random Swaps
    else if type = 2 then
        Perform PLGRS Bad Arc Targeting
    else if type = 3 then
        Perform PLGRS Bad Arc Targeting & Good Node
    end if
    Perform Ordered Greedy(order)
    if score ≤ bestScore then
        if score <bestScore then
            α = start_α
            αcount=0
        end if
        bestScore = score
        bestTour = tour
        bestOrder = order
    else
        order = bestOrder
        αcount = αcount + 1
    end if
end for
```

VI. PLGRS Results

The same 13 symmetric instances introduced in section 4.1 were used to compare all meta-heuristics. Microbenchmark was used to test each heuristic. 10 iterations of each instance size were run, with the exception of the 1002 size instance which was only run three times due to computational requirements. Percent deviation from optimality for each iteration was collected as well as run-times to summarize the performance of each heuristic.

6.1 Greedy+2-Opt Comparison

The first test was conducted comparing all three PLGRS heuristics against a arc-greedy+2-Opt heuristic. The arc-greedy+2-opt heuristic was selected due to its deterministic nature which causes it to always converges to the same solution. Therefore, the arc-greedy+2-Opt does not contain features such as randomness or tuneable elements. While each aspect can be advantageous in a heuristic methodology, if used improperly they can also be a hindrance. Thus, the arc-greedy+2-opt gives a good baseline computational time and final solution for which to compare the PLGRS codes against. Since the arc-greedy+2-opt heuristic always converges to the same solution we limited the number of iterations provided to the PLGRS code so it was not given an advantage. If run indefinitely, most randomized meta-heuristics, while not guaranteed to reach optimality, will approach it. Thus, by limiting the number of iterations assigned to PLGRS, it was ensured that all heuristics found a solutions within a similar amount of time. To accomplish this, the PLGRS - Random Swaps code was run with varying numbers of iterations until a time close to the runtime of the arc-greedy+2-Opt was achieved. This runtime threshold determination was accomplished for all instances with the exception of the Bays29 and gr48 instances, for

which PLGRS could not run a single iteration in the time it took the arc-greedy+2-opt to run to completion. For those instances, PLGRS was given 50 iterations. Results for these runs are in Table 7.

Table 7. Greedy 2-Opt vs. PLGRS Comparison

	% within opt					Runtimes (seconds)				
Instance	PLGRS RS	PLGRS BAT	PLGRS BATGN	PLGRS ALL	Greedy 2OPT	PLGRS RS	PLGRS BAT	PLGRS BATGN	PLGRS ALL	Greedy 2OPT
bays29	7.67%	7.33%	9.16%	8.42%	6.58%	47.34(ms)	59.05(ms)	48.81(ms)	50.13(ms)	46.93(ms)
gr48	16.37%	15.00%	15.87%	13.50%	14.76%	155.14(ms)	181.84(ms)	180.33(ms)	204.53(ms)	186.26(ms)
eil51	4.46%	8.59%	11.46%	7.25%	3.99%	801.68(ms)	827.81(ms)	808.39(ms)	867.58(ms)	812.01(ms)
berlin52	16.40%	18.95%	13.76%	11.32%	17.62%	217.85(ms)	243.38(ms)	252.19(ms)	280.55(ms)	219.95(ms)
pr76	9.78%	7.17%	11.33%	9.14%	26.79%	692.57(ms)	721.6(ms)	741.53(ms)	739.92(ms)	771.68(ms)
kroa100	10.02%	11.58%	10.07%	10.48%	11.73%	1061.89(ms)	991.71(ms)	995.93(ms)	1053.59(ms)	1033.26(ms)
gr120	11.64%	14.75%	13.28%	13.73%	16.02%	1.59	1.60	1.65	1.68	1.75
ch130	8.97%	10.16%	8.64%	8.33%	15.24%	2.02	2.19	2.08	2.14	2.16
rat195	10.76%	10.89%	12.44%	10.12%	5.25%	6.21	6.17	6.52	6.66	6.92
ts225	4.95%	5.38%	5.19%	5.06%	5.03%	8.44	10.08	9.50	9.00	10.40
pma343	13.30%	16.01%	13.74%	13.23%	16.59%	37.69	36.81	37.22	37.34	37.92
pcb442	9.44%	16.89%	9.01%	9.10%	12.68%	66.25	82.54	73.19	70.42	79.95
pr1002	7.72%	11.34%	11.92%	8.65%	15.59%	32.95(ms)	33.21(ms)	34.09(ms)	34.48(ms)	38.32(ms)

The results do not clearly indicate any heuristic being truly dominant. It appears the most notable trend is that the PLGRS heuristics tend to perform better, relative to the Greedy-2-Opt, as instance size grow, with the exception of instances 195 and 225. Given that the PLGRS heuristic is an iterative constructive heuristic, each iteration of PLGRS takes longer to complete than an iteration of the arc-greedy+2-Opt. Thus PLGRS is able to consider significantly less iterations/solution. The advantages of the ordered-list space, however seem to largely counteract this, thus while PLGRS considers less solutions, they tend to be of higher quality.

6.2 Simulated Annealing Comparison

The simulated annealing comparison meta-heuristic also generates an initial greedy tour, each iteration then considers a random 2-opt where good moves are accepted and bad moves are probabilisticaly accepted based on a temperature function. For these runs, SA was tested to determine a suitable number of iterations until noticeable

stagnation began to occur the associated run time for stagnation was noted. Then the PLGRS codes were run to determine the number of iterations associated with this runtime threshold and were limited in testing to this number of iterations. Table 8 is a summary of the SA vs PLGRS runs.

Table 8. SA 2-Opt vs. PLGRS Comparison

	% within opt					Runtimes (seconds)				
Instance	PLGRS RS	PLGRS BAT	PLGRS BATGN	PLGRS ALL	SA	PLGRS RS	PLGRS BAT	PLGRS BATGN	PLGRS ALL	SA
bays29	2.38%	4.06%	3.37%	3.96%	0.89%	699.08(ms)	713.24(ms)	731.4(ms)	727.05(ms)	759.29(ms)
gr48	3.45%	8.26%	5.09%	5.45%	2.26%	1.27	1.32	1.28	1.37	1.25
eil51	4.58%	6.83%	4.48%	4.77%	2.39%	1.28	1.36	1.34	1.38	1.37
berlin52	4.39%	5.54%	3.30%	2.21%	4.14%	3.19	3.08	3.51	3.01	2.94
pr76	5.11%	6.39%	4.83%	5.08%	7.40%	3.69	3.77	3.83	3.87	3.84
kroa100	8.69%	10.43%	8.34%	8.86%	2.52%	4.97	4.77	4.94	4.97	4.72
gr120	8.95%	10.44%	9.65%	8.59%	4.29%	5.53	5.71	5.68	5.82	5.84
ch130	4.26%	7.22%	3.93%	4.71%	5.66%	7.93	8.84	8.34	8.31	8.09
rat195	9.64%	10.20%	10.42%	8.78%	10.50%	9.16	9.59	9.86	9.69	8.83
ts225	4.54%	5.38%	4.76%	4.72%	1.84%	12.70	13.33	12.85	12.32	12.47
pma343	14.33%	15.06%	13.60%	13.23%	8.26%	22.14	22.67	22.93	22.61	23.52
pcb442	10.73%	15.94%	9.43%	9.54%	6.89%	56.10	65.75	60.49	58.96	65.05
pr1002	13.26%	12.81%	12.70%	12.11%	6.68%	183.57	183.65	184.29	184.88	203.92

For a majority of instances, SA demonstrated markedly lower optimality gaps then the PLGRS codes. This issue is exacerbated by larger instances, which highlights an issue with utilizing a iterative constructive heuristic methodology. As instance size grows, the time for each iteration also grows with the PLGRS codes. So when comparing PLGRS to a fast pseudo-random heuristic such as SA, and confining each to similar run-times, PLGRS is at an extreme disadvantage. A principle factor is likely the number of iterations each heuristic code accomplished over the fixed runtime by instance (Table 9), as the SA employs many more iterations.

Starting at the Bays29 instance, the SA 2-Opt code completes roughly 30 times as many iterations as the PLGRS code, and this ratio steadily rises to the 1002 instance where SA can complete nearly 6700 times more iterations than PLGRS. Considering the optimality gaps in this testing evidence strongly points toward searching the Ordered-List subspace providing advantages on a per iteration basis, however the

Table 9. SA 2-Opt vs. PLGRS Iterations Comparison

	Iterations	
Instance	PLGRS Codes	SA
bays29	1300	40000
gr48	1250	60000
eil51	1150	70000
berlin52	2500	150000
pr76	1750	180000
kroa100	1500	210000
gr120	1350	265000
ch130	1700	360000
rat195	1100	360000
ts225	1200	500000
pma343	1020	800000
pcb442	1600	2000000
pr1002	1200	4000000

time it takes to do so severely limits the methodology's potential.

6.3 Future Improvement

When comparing the results of each of the PLGRS variants it appears that PLGRS - ALL generally outperformed the other two variants, although there were cases where PLGRS-RS was the best performing methodology. This suggests some benefit in the strategic approach of targeting bad arcs and swapping their location in an ordered-list with arcs that have a high number of relatively good connections. However, future research should consider the direct effects of making such moves, whether there is a more informed way of performing such operations, and then performing those moves computationally cheaper. With such enhancements, PLGRS could maximize its conceptual advantages to improve the tour.

The PLGRS methodology also lends well to utilizing parallel processing which could provide vast improvements to its computational time. If multiple lists could be tested simultaneously at each iteration, always tracking and attempting to improve upon the best solution, would allow PLGRS to close the iteration count gap it is experiencing in relation to these other heuristics. This could give further motivation for using the ordered-list space when solving the TSP.

VII. Conclusion

As a hard combinatorial optimization problem, the TSP is often solved via heuristic methodologies. One of the biggest considerations when constructing solutions is avoiding sub-tours, or a loop of interconnected nodes that prevents a single continuous tour amongst all cities within the instance. This paper introduced a novel sub-tour elimination methodology for the arc-greedy heuristic that is compared to two known sub-tour elimination methodologies. Computational results were generated across multiple TSP instances for each method. A novel concept called Ordered-Lists was also introduced which enables TSP instances to be explored in a different space than the tour space. The Ordered-List tour space demonstrates some unique properties. We propose some novel meta-heuristics that seek to utilize this new space.

7.1 Sub-tour Elimination

When utilizing an arc-greedy heuristic, additional steps must be taken to ensure that sub-tours are avoided and resulting tour is a valid TSP solution. This paper recognized two accepted arc-greedy sub-tour elimination methodologies, the Exhaustive loop and Bentley's Multi-fragment, and compared them to a novel methodology, the Greedy Tracker. The comparison utilized both directional and non-directional variants of each code on 13 symmetric TSP instances and the directional variants on 9 asymmetric instances.

The results of the comparison between each of these arc-greedy sub-tour elimination methodologies showed that the GT was the fastest tracking methodology for small to medium sized instances. However, Bentley's MF still maintains the computational advantage for larger instances and thus most, if not all, instances that would be solved utilizing a heuristic methodology.

However, these results also indicated that given a more efficient coding implementation of methodology used for the X Matrix, the GT could become the preferred methodology for all instance sizes. For future research, the GT should be modified to handle a new row/column generation and delete technique to minimize the computational time utilized in the searching portions of the GT.

7.2 Ordered-Lists

Any improvement upon any of these sub-tour elimination methodologies would also provide direct computational improvement to the novel heuristic methodology, the Ordered Greedy, which in turn would give greater efficiency to searching the Ordered-List solution space.

While computationally more demanding than its tour list counterpart, the solution quality advantages, as well as a possibly higher number of optimal occurrences, when a Perfect Order exists, seems to indicated that further investigation of the space may be worthwhile to the TSP community.

The novel meta-heuristic methodologies introduced in this paper sought to leverage the advantages of the Ordered list space. Testing results indicate that while at a severe iteration disadvantage, the PLGRS methodologies benefited from using the ordered-list space which yields a higher per iteration improvement rate. For future research, the PLGRS methodologies could benefit from parallel processing and a more efficient methodology for targeting what list modifications should be made. Deeper investigation of the Ordered-List space would also be worthwhile to fully investigate its relation to the tour order space.

Appendix A. R Code

Greedy Tracker

```
library(readxl)
TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test1002.csv",header = FALSE)
#Turn it into a usable matrix
Data<-as.matrix(TSP_Data)
a = Sys.time()
#Get the data sets size
#dim initial vars
size<-dim(Data)[1]
listarcs = matrix(0,size^2,3)
tours = matrix(0,size,size)
to = rep(0,size)
from = rep(0,size)
trails = matrix(0,size,size)
diag(trails)=1
#generate list of arcs
count = 1
for (i in 1:size) {
  for (j in 1:size) {
    listarcs[count,1] = Data[i,j]
    listarcs[count,2] = i
    listarcs[count,3] = j
    if (i==j) {listarcs[count,1]=Inf}
    count= count + 1
  }
}
#sort list
listarcs = listarcs[order(listarcs[,1],decreasing = FALSE), ]

opcount=0
num.visited = 0
count = 1

#While statement
while (num.visited<size-1) {
  #check all greedy tracker structures to see if current arc is valid if not loop to next arc
  if ((from[listarcs[count,2]]==0)&&(to[listarcs[count,3]]==0)&&(trails[listarcs[count,2],listarcs[count,3]]==0)) {
    #add arc
    num.visited = num.visited+1
    trails[listarcs[count,2],listarcs[count,3]]=1
    tours[listarcs[count,2],listarcs[count,3]]=1
    from[listarcs[count,2]]=1
    to[listarcs[count,3]]=1

    #greedy tracker
    #find all rows with >0
    listarc = which(trails[,listarcs[count,3]]>0)
    #add current row to rows with column value >0
    for (i in 1:length(listarc)) {
      trails[listarc[i],]=trails[listarc[i],]+trails[listarcs[count,2],]
      opcount = opcount +1
    }
  }
  count = count +1
}
#Connect hamilton path start to finish
tours[which(from==0),which(to==0)]=1

b = Sys.time()
score=sum(tours*Data)
```

```
60
61 print(score)
62 print(b - a)
```

Greedy Tracker Modified

```
1  library(readxl)
2  TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test1002.csv",header
       = FALSE)
3  #Turn it into a usable matrix
4  Data<-as.matrix(TSP_Data)
5  a = Sys.time()
6  #Get the data sets size
7  #dim initial vars
8  size<-dim(Data)[1]
9  listarcs = matrix(0,size^2,3)
10 tours = matrix(0,size,size)
11 to = rep(0,size)
12 from = rep(0,size)
13 trails = matrix(0,size,size)
14 diag(trails)=1
15 #generate list of arcs
16 count = 1
17 for (i in 1:size) {
18   for (j in 1:size) {
19     listarcs[count,1] = Data[i,j]
20     listarcs[count,2] = i
21     listarcs[count,3] = j
22     if (i==j) {listarcs[count,1]=Inf}
23     count= count + 1
24   }
25 }
26 #sort list
27 listarcs = listarcs[order(listarcs[,1],decreasing = FALSE), ]
28 num.visited = 0
29 count = 1
30
31 #While statement
32 while (num.visited<size-1) {
33   #check all greedy tracker structures to see if current arc is valid if not loop to
         next arc
34   if ((from[listarcs[count,2]]==0)&&(to[listarcs[count,3]]==0)&&(trails[listarcs[
         count,2],listarcs[count,3]]==0)) {
35     #add arc
36     num.visited = num.visited+1
37     #trails[listarcs[count,2],listarcs[count,3]]=1
38     tours[listarcs[count,2],listarcs[count,3]]=1
39     from[listarcs[count,2]]=1
40     to[listarcs[count,3]]=1
41
42     #greedy tracker
43
44     #add current row to rows with column value >0
45     listarc = intersect(which(trails[,listarcs[count,3]]>0),which(from==0))
46     listarc2 = intersect(which(trails[listarcs[count,2],]>0),which(to==0))
47     #add current row to rows with column value >0
48     trails[listarc,listarc2]=1
49   }
50   count = count +1
51 }
52 #Connect hamilton path start to finish
53 tours[which(from==0),which(to==0)]=1
54
55 b = Sys.time()
```

```
score=sum(tours*Data)

print(score)
print(b - a)
```

Ordered Greedy

```
library(readxl)
TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test7by7.csv",header
    = FALSE)

#Turn it into a usable matrix
Data<-as.matrix(TSP_Data)
#Read in Dataset named Lab_Data

#Get the data sets size
size<-dim(Data)[1]
order = rep(1:size,1)
a = Sys.time()

  #Initialize Variables
  num.visited = 0
  to = rep(0,size)
  from = rep(0,size)
  trails =  matrix(0,size,size)
  diag(trails)=1
  tours = matrix(0,size,size)

  #While statement
  while (num.visited<size-1) {
    current.distances<-Data[,order[num.visited+1]]
    eligible = intersect(which(to==0),which(trails[order[num.visited+1],]==0))
    nextTownToVisit = eligible[as.integer(which(current.distances[eligible]==min(
        current.distances[eligible]),arr.ind = T,useNames = F)[1])]#In case of ties,
        take just the first

    nextTownToVisit = c(order[num.visited+1], nextTownToVisit)
    trails[nextTownToVisit[1],nextTownToVisit[2]]=1
    tours[nextTownToVisit[1],nextTownToVisit[2]]=1
    from[nextTownToVisit[1]] = 1
    to[nextTownToVisit[2]]= 1

    #row addition Trails code
    listarc = which(trails[,nextTownToVisit[2]]==1)
    for (i in 1:length(listarc)) {
      trails[listarc[i],]=trails[listarc[i],]+trails[nextTownToVisit[1],]
    }
    num.visited = num.visited + 1
  }
  #Set last arc to finalize tour
  tours[which(from==0),which(to==0)]=1
  score=sum(tours*Data)


tourcheck = which(tours==1,arr.ind = T,useNames = F)
tour=rep(0,size)
start = 1
current = 1
current = tourcheck[current,1]
count = 1
while (current!=start) {
  tour[count]=current
  current = tourcheck[current,1]
  count = count+1
```

```
}
tour[count]=current

b=Sys.time()
print(score)
print(b - a)
print(best_order)
print(tour)
```

PLGRS RS

```
TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test48.csv",header =
    FALSE)
#Turn it into a usable matrix
Data<-as.matrix(TSP_Data)

#Get the data sets size
size<-dim(Data)[1]

iter = 50
startalpha =4
changealpha =.5
intensifycrit = 5

greedytour = rep(1:size,1)
#***********PART 1: get Greedy Tour******************

  listarcs = matrix(0,size*(size+1)/2,3)
  tours = matrix(0,size,size)
  Degree = rep(0,size)
  Tail = rep(1:size)
  taili=0
  tailj=0
  temptaili = 0
  temptailj = 0
  #generate list of arcs column 1 is length, column two is tail, column 3 is head
  count = 1
  for (i in 1:size) {
    for (j in i:(size)) {
      listarcs[count,1] = Data[i,j]
      listarcs[count,2] = i
      listarcs[count,3] = j
      if (i==j) {listarcs[count,1]=Inf}
      count= count + 1
    }
  }
  #sort the list by length
  listarcs = listarcs[order(listarcs[,1],decreasing = FALSE), ]
  #initialize more variables
  num.visited = 0
  count = 1
  #While statement (create hamilton path)
  while (num.visited<size-1) {
    #node leaving does not have a arc leaving and node going to does not have an arc
        entering
    if ((Degree[listarcs[count,2]]<2)&&(Degree[listarcs[count,3]]<2)&&(Tail[listarcs[
        count,2]]!=listarcs[count,3])) {
      #add arc
      tours[listarcs[count,2],listarcs[count,3]]=1
      tours[listarcs[count,3],listarcs[count,2]]=1
      #if both are 0 degree
      if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==0)) {

        taili =Tail[listarcs[count,2]]
```

```
        tailj =Tail[listarcs[count,3]]
        Tail[listarcs[count,2]]=tailj
        Tail[listarcs[count,3]]=taili

      } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==0)) {

        taili =Tail[listarcs[count,2]]
        Tail[taili]=Tail[listarcs[count,3]]
        Tail[listarcs[count,3]]=taili
        Tail[listarcs[count,2]]=0

      } else if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==1)) {

        tailj =Tail[listarcs[count,3]]
        Tail[tailj] =Tail[listarcs[count,2]]
        Tail[listarcs[count,2]]=tailj
        Tail[listarcs[count,3]]=0

      } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==1)) {

        taili =Tail[listarcs[count,2]]
        tailj =Tail[listarcs[count,3]]
        Tail[taili]=tailj
        Tail[tailj]=taili
        Tail[listarcs[count,2]]=0
        Tail[listarcs[count,3]]=0
      }
      #set start to tail and current to head
      Degree[listarcs[count,2]]=Degree[listarcs[count,2]]+1
      Degree[listarcs[count,3]]=Degree[listarcs[count,3]]+1
      num.visited = num.visited+1
    }
    count = count +1
  }
  #connect hamilton path start to finish
  tours[which(Degree<2)[1],which(Degree<2)[2]]=1
  tours[which(Degree<2)[2],which(Degree<2)[1]]=1
  score=sum(tours*Data)/2

  previousnode = 0
  currentnode = 1
  for (j in 1:size) {
    nodes = which(tours[currentnode,]==1)
    if (nodes[1]!=previousnode) {
      greedytour[j] = nodes[1]
      previousnode=currentnode
      currentnode=nodes[1]
    }  else {
      greedytour[j] = nodes[2]
      previousnode=currentnode
      currentnode=nodes[2]
    }
  }

  TSP_Tour=greedytour
  TSP_Tour = c(TSP_Tour,greedytour[1])
  Greedy_Tour = matrix(0,size,1)
  prev = TSP_Tour[1]
  for (i in 1:size+1) {
    Greedy_Tour[prev] = TSP_Tour[i]
    prev = TSP_Tour[i]
  }

#**********PART 2: Get Greedy Order with tiering***********

  #Initialize Variables
```

```
  num.visited = 0
  to = rep(T,size)
  from = rep(T,size)
  trails = matrix(T,size,size)
  diag(trails)=F
  tours = matrix(0,size,size)
  greedloop = rep(0,size)
  GreedyOrder = rep(0,size)
  tiering = rep(0,size)
  tier = 1
  while (num.visited<size-1) {
    current.distances<-Data[]
    current.distances[!from] = Inf
    current.distances[,!to] = Inf
    current.distances=ifelse(trails==F,Inf,current.distances)
    numintier = 0
    #go though every node
    for (i in 1:size) {
      #if node hasnt been left yet
      if (from[i]==T) {
        #find the min distance arcs
        availmin = which(current.distances[i,]==min(current.distances[i,]),arr.ind =
            T, useNames = F)[1]
        #if only min distance arc AND same as in opt tour (this can probably just be
            made availmin[1] and length removed)
        if ((availmin == Greedy_Tour[i])&&((numintier+num.visited)<(size-1))) {
          #update number in tier
          numintier = numintier + 1
          #store connection
          greedloop[numintier]= i
        }
      }
    }
    #loop through connections in tier
    for (j in 1:numintier) {
      #Perfrorm greed tracker
      trails[greedloop[j],Greedy_Tour[greedloop[j]]]=F
      tours[greedloop[j],Greedy_Tour[greedloop[j]]]=1
      from[greedloop[j]] = F
      to[Greedy_Tour[greedloop[j]]] = F

      #row addition Trails code
      for (i in 1:size) {
        if (trails[i,Greedy_Tour[greedloop[j]]]==F ) {
          trails[i,]=trails[i,]&trails[greedloop[j],]
        }
      }

      num.visited = num.visited + 1
      #stroe perfect orde4r list
      GreedyOrder[num.visited] = greedloop[j]
      tiering[num.visited] = tier
    }
    tier = tier+1

  }
  #Set last arc to finalize tour
  tours[which(from==T),which(to==T)]=1
  #sometimes cause error due to looping structure
  GreedyOrder[num.visited+1]=which(from==T)
  tiering[num.visited+1]=tier

#***********   PART 3: Greedy Random Search    **********
  graphvector<-matrix()
  order = GreedyOrder

```

```
181   best_order = order
182   best_score = score
183   a = Sys.time()
184   #Number of list order swaps at each iteration
185   numswaps = 1
186
187   for (k in 1:iter) {
188     #Initialize Variables
189     num.visited = 0
190     to = rep(T,size)
191     from = rep(T,size)
192     trails =  matrix(T,size,size)
193     diag(trails)=F
194     tours = matrix(0,size,size)
195
196     for (j in 1:numswaps) {
197       swap1 = sample(size,1)
198       swap2 = sample(size,1)
199       temp = order[swap2]
200       order[swap2]=order[swap1]
201       order[swap1]=temp
202     }
203
204     #While statement
205
206     while (num.visited<size-1) {
207       current.distances<-Data[,order[num.visited+1]]
208       current.distances[!to]=Inf
209       current.distances[!trails[order[num.visited+1],]]=Inf
210       nextTownToVisit = as.integer(which(current.distances==min(current.distances),
              arr.ind = T,useNames = F)[1])#In case of ties, take just the first
211
212       #current.distances.notVisited<-Data[,order[num.visited+1]][to][trails]
213       #shortestDistance = min(current.distances.notVisited)
214       # The exclamation mark was not added in V1
215       #current.distances[!to] = NA #Any towns visited set to NA so they can't be
              matched in next line
216       #nextTownToVisit = as.integer(which(current.distances == shortestDistance)[1])
              #In case of ties, take just the first
217       ################################
218       nextTownToVisit = c(order[num.visited+1], nextTownToVisit)
219       trails[nextTownToVisit[1],nextTownToVisit[2]]=F
220       tours[nextTownToVisit[1],nextTownToVisit[2]]=1
221       from[nextTownToVisit[1]] = F
222       to[nextTownToVisit[2]]= F
223
224       #row addition Trails code
225       listarc = which(trails[,nextTownToVisit[2]]==F)
226       for (i in 1:length(listarc)) {
227         trails[listarc[i],]=trails[listarc[i],]&trails[nextTownToVisit[1],]
228       }
229       num.visited = num.visited + 1
230     }
231     #Set last arc to finalize tour
232     tours[which(from==T),which(to==T)]=1
233     score=sum(tours*Data)
234
235
236
237     if (score<=best_score) {
238       best_score=score
239       best_tour=tours
240       best_order=order
241     } else {
242       order = best_order
243     }
```

```
244
245   }
```

PLGRS BAT

```
1
2  TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test48.csv",header =
       FALSE)
3  #Turn it into a usable matrix
4  Data<-as.matrix(TSP_Data)
5
6  #Get the data sets size
7  size<-dim(Data)[1]
8
9
10 iter = 50
11 startalpha =4
12 changealpha =.5
13 intensifycrit = 5
14
15 greedytour = rep(1:size,1)
16
17 #***********PART 1: get Greedy Tour*******************
18
19   listarcs = matrix(0,size*(size+1)/2,3)
20   tours = matrix(0,size,size)
21   Degree = rep(0,size)
22   Tail = rep(1:size)
23   taili=0
24   tailj=0
25   temptaili = 0
26   temptailj = 0
27   #generate list of arcs column 1 is length, column two is tail, column 3 is head
28   count = 1
29   for (i in 1:size) {
30     for (j in i:(size)) {
31       listarcs[count,1] = Data[i,j]
32       listarcs[count,2] = i
33       listarcs[count,3] = j
34       if (i==j) {listarcs[count,1]=Inf}
35       count= count + 1
36     }
37   }
38   #sort the list by length
39   listarcs = listarcs[order(listarcs[,1],decreasing = FALSE), ]
40   #initialize more variables
41   num.visited = 0
42   count = 1
43   #While statement (create hamilton path)
44   while (num.visited<size-1) {
45     #node leaving does not have a arc leaving and node going to does not have an arc
         entering
46     if ((Degree[listarcs[count,2]]<2)&&(Degree[listarcs[count,3]]<2)&&(Tail[listarcs[
         count,2]]!=listarcs[count,3])) {
47       #add arc
48       tours[listarcs[count,2],listarcs[count,3]]=1
49       tours[listarcs[count,3],listarcs[count,2]]=1
50       #if both are 0 degree
51       if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==0)) {
52
53         taili =Tail[listarcs[count,2]]
54         tailj =Tail[listarcs[count,3]]
55         Tail[listarcs[count,2]]=tailj
56         Tail[listarcs[count,3]]=taili
```

```
57
58        } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==0)) {
59
60          taili =Tail[listarcs[count,2]]
61          Tail[taili]=Tail[listarcs[count,3]]
62          Tail[listarcs[count,3]]=taili
63          Tail[listarcs[count,2]]=0
64
65        } else if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==1)) {
66
67          tailj =Tail[listarcs[count,3]]
68          Tail[tailj] =Tail[listarcs[count,2]]
69          Tail[listarcs[count,2]]=tailj
70          Tail[listarcs[count,3]]=0
71
72        } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==1)) {
73
74          taili =Tail[listarcs[count,2]]
75          tailj =Tail[listarcs[count,3]]
76          Tail[taili]=tailj
77          Tail[tailj]=taili
78          Tail[listarcs[count,2]]=0
79          Tail[listarcs[count,3]]=0
80        }
81        #set start to tail and current to head
82        Degree[listarcs[count,2]]=Degree[listarcs[count,2]]+1
83        Degree[listarcs[count,3]]=Degree[listarcs[count,3]]+1
84        num.visited = num.visited+1
85      }
86      count = count +1
87    }
88    #connect hamilton path start to finish
89    tours[which(Degree<2)[1],which(Degree<2)[2]]=1
90    tours[which(Degree<2)[2],which(Degree<2)[1]]=1
91    score=sum(tours*Data)/2
92
93    previousnode = 0
94    currentnode = 1
95    for (j in 1:size) {
96      nodes = which(tours[currentnode,]==1)
97      if (nodes[1]!=previousnode) {
98        greedytour[j] = nodes[1]
99        previousnode=currentnode
100       currentnode=nodes[1]
101     }  else {
102       greedytour[j] = nodes[2]
103       previousnode=currentnode
104       currentnode=nodes[2]
105     }
106   }
107
108   TSP_Tour=greedytour
109   TSP_Tour = c(TSP_Tour,greedytour[1])
110   Greedy_Tour = matrix(0,size,1)
111   prev = TSP_Tour[1]
112   for (i in 1:size+1) {
113     Greedy_Tour[prev] = TSP_Tour[i]
114     prev = TSP_Tour[i]
115   }
116
117 #**********PART 2: Get Greedy Order with tiering**********
118   #Initialize Variables
119   num.visited = 0
120   to = rep(T,size)
121   from = rep(T,size)
122   trails = matrix(T,size,size)
```

```
  diag(trails)=F
  tours = matrix(0,size,size)
  greedloop = rep(0,size)
  GreedyOrder = rep(0,size)
  tiering = rep(0,size)
  tier = 1
  while (num.visited<size-1) {
    current.distances<-Data[]
    #current.distances[!from] = Inf
    current.distances[,!to] = Inf
    current.distances=ifelse(trails==F,Inf,current.distances)
    numintier = 0
    #go though every node
    for (i in 1:size) {
      #if node hasnt been left yet
      if (from[i]==T) {
        #find the min distance arcs
        availmin = which(current.distances[i,]==min(current.distances[i,]),arr.ind =
            T, useNames = F)[1]
        #if only min distance arc AND same as in opt tour (this can probably just be
            made availmin[1] and length removed)
        if ((availmin == Greedy_Tour[i])&&((numintier+num.visited)<(size-1))) {
          #update number in tier
          numintier = numintier + 1
          #store connection
          greedloop[numintier]= i
        }
      }
    }
    #loop through connections in tier
    for (j in 1:numintier) {
      #Perfrorm greed tracker
      trails[greedloop[j],Greedy_Tour[greedloop[j]]]=F
      tours[greedloop[j],Greedy_Tour[greedloop[j]]]=1
      from[greedloop[j]] = F
      to[Greedy_Tour[greedloop[j]]] = F

      #row addition Trails code
      for (l in 1:size) {
        if (trails[l,Greedy_Tour[greedloop[j]]]==F ) {
          trails[l,]=trails[l,]&trails[greedloop[j],]
        }
      }

      num.visited = num.visited + 1
      #stroe perfect orde4r list
      GreedyOrder[num.visited] = greedloop[j]
      tiering[num.visited] = tier
    }
    tier = tier+1

  }
  #Set last arc to finalize tour
  tours[which(from==T),which(to==T)]=1
  #sometimes cause error due to looping structure
  GreedyOrder[num.visited+1]=which(from==T)
  tiering[num.visited+1]=tier


#***********   PART 3: Adaptive List    **********
  #Re-Initialize Variables
  graphvector<-matrix()
  order = GreedyOrder

  alpha=startalpha
  alphacount = 0
```

```
187   best_order = order
188   best_score = score
189   a = Sys.time()
190   #Number of list order swaps at each iteration
191
192   minvals = rep(0,size)
193   badarc = rep(0,size)
194
195   for (i in 1:size) {
196     minval = tail(sort(Data[i,],decreasing = F,index.return=T),2)
197     minvals[i] = minval$x[2]
198   }
199
200   for (k in 1:iter) {
201     #Initialize Variables
202     num.visited = 0
203     to = rep(T,size)
204     from = rep(T,size)
205     trails =  matrix(T,size,size)
206     diag(trails)=F
207     arcvals = rowSums(tours*Data)
208     tours = matrix(0,size,size)
209
210     havearc=F
211
212     while (havearc == F) {
213       badarc = which(arcvals>=(minvals+minvals*alpha))
214
215       if ((length(badarc)<2)||(alphacount >= intensifycrit)){
216         alpha = alpha-changealpha
217         alphacount=0
218         if (alpha < 0) {alpha=startalpha}
219       } else {
220         move1 = sample(badarc,1)
221         oldloc=which(order==move1)
222         if (oldloc!=1) {
223           havearc=T
224         }
225       }
226     }
227
228     newloc=sample(oldloc-1,1)
229
230     if (newloc==1) {
231       if (oldloc==size) {
232         temporder = c(move1,order[1:size-1])
233         order=temporder
234       } else {
235       temporder = c(move1,order)
236       order = c(temporder[1:(oldloc)],temporder[(oldloc+2):(size+1)])
237       }
238     } else if (oldloc==size){
239       temporder = c(order[1:newloc-1],move1,order[newloc:size])
240       order = c(temporder[1:(oldloc)])
241     } else {
242       temporder = c(order[1:newloc-1],move1,order[newloc:size])
243       order = c(temporder[1:(oldloc)],temporder[(oldloc+2):(size+1)])
244     }
245
246     #While statement
247     while (num.visited<size-1) {
248
249       current.distances<-Data[,order[num.visited+1]]
250       current.distances[!to]=Inf
251       current.distances[!trails[order[num.visited+1],]]=Inf
252       nextTownToVisit = as.integer(which(current.distances==min(current.distances),
```

```
          arr.ind = T,useNames = F)[1])#In case of ties, take just the first
253
254       nextTownToVisit = c(order[num.visited+1], nextTownToVisit)
255       trails[nextTownToVisit[1],nextTownToVisit[2]]=F
256       tours[nextTownToVisit[1],nextTownToVisit[2]]=1
257       from[nextTownToVisit[1]] = F
258       to[nextTownToVisit[2]]= F
259
260       #row addition Trails code
261       listarc = which(trails[,nextTownToVisit[2]]==F)
262       for (i in 1:length(listarc)) {
263         trails[listarc[i],]=trails[listarc[i],]&trails[nextTownToVisit[1],]
264       }
265       num.visited = num.visited + 1
266     }
267     #Set last arc to finalize tour
268     tours[which(from==T),which(to==T)]=1
269     score=sum(tours*Data)
270
271     if (score<=best_score) {
272       if (score<best_score) {
273         alpha=startalpha
274         alphacount=0
275       }
276       best_score=score
277       best_tour=tours
278       best_order=order
279     } else {
280       order = best_order
281       alphacount = alphacount + 1
282     }
283
284   }
```

PLGRS BATGN

```
1  TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test48.csv",header =
       FALSE)
2  #Turn it into a usable matrix
3  Data<-as.matrix(TSP_Data)
4
5  #Get the data sets size
6  size<-dim(Data)[1]
7
8
9  iter = 50
10 startalpha =4
11 changealpha =.5
12 intensifycrit = 5
13 greedytour = rep(1:size,1)
14
15 #***********PART 1: get Greedy Tour******************
16
17   listarcs = matrix(0,size*(size+1)/2,3)
18   tours = matrix(0,size,size)
19   Degree = rep(0,size)
20   Tail = rep(1:size)
21   taili=0
22   tailj=0
23   temptaili = 0
24   temptailj = 0
25   #generate list of arcs column 1 is length, column two is tail, column 3 is head
26   count = 1
27   for (i in 1:size) {
```

```
    for (j in i:(size)) {
      listarcs[count,1] = Data[i,j]
      listarcs[count,2] = i
      listarcs[count,3] = j
      if (i==j) {listarcs[count,1]=Inf}
      count= count + 1
    }
  }
  #sort the list by length
  listarcs = listarcs[order(listarcs[,1],decreasing = FALSE), ]
  #initialize more variables
  num.visited = 0
  count = 1
  #While statement (create hamilton path)
  while (num.visited<size-1) {
    #node leaving does not have a arc leaving and node going to does not have an arc entering
    if ((Degree[listarcs[count,2]]<2)&&(Degree[listarcs[count,3]]<2)&&(Tail[listarcs[count,2]]!=listarcs[count,3])) {
      #add arc
      tours[listarcs[count,2],listarcs[count,3]]=1
      tours[listarcs[count,3],listarcs[count,2]]=1
      #if both are 0 degree
      if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==0)) {

        taili =Tail[listarcs[count,2]]
        tailj =Tail[listarcs[count,3]]
        Tail[listarcs[count,2]]=tailj
        Tail[listarcs[count,3]]=taili

      } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==0)) {

        taili =Tail[listarcs[count,2]]
        Tail[taili]=Tail[listarcs[count,3]]
        Tail[listarcs[count,3]]=taili
        Tail[listarcs[count,2]]=0

      } else if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==1)) {

        tailj =Tail[listarcs[count,3]]
        Tail[tailj] =Tail[listarcs[count,2]]
        Tail[listarcs[count,2]]=tailj
        Tail[listarcs[count,3]]=0

      } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==1)) {

        taili =Tail[listarcs[count,2]]
        tailj =Tail[listarcs[count,3]]
        Tail[taili]=tailj
        Tail[tailj]=taili
        Tail[listarcs[count,2]]=0
        Tail[listarcs[count,3]]=0
      }
      #set start to tail and current to head
      Degree[listarcs[count,2]]=Degree[listarcs[count,2]]+1
      Degree[listarcs[count,3]]=Degree[listarcs[count,3]]+1
      num.visited = num.visited+1
    }
    count = count +1
  }
  #connect hamilton path start to finish
  tours[which(Degree<2)[1],which(Degree<2)[2]]=1
  tours[which(Degree<2)[2],which(Degree<2)[1]]=1
  score=sum(tours*Data)/2

  previousnode = 0
```

```
  currentnode = 1
  for (j in 1:size) {
    nodes = which(tours[currentnode,]==1)
    if (nodes[1]!=previousnode) {
      greedytour[j] = nodes[1]
      previousnode=currentnode
      currentnode=nodes[1]
    }  else {
      greedytour[j] = nodes[2]
      previousnode=currentnode
      currentnode=nodes[2]
    }
  }

  TSP_Tour=greedytour
  TSP_Tour = c(TSP_Tour,greedytour[1])
  Greedy_Tour = matrix(0,size,1)
  prev = TSP_Tour[1]
  for (i in 1:size+1) {
    Greedy_Tour[prev] = TSP_Tour[i]
    prev = TSP_Tour[i]
  }

##########PART 2: Get Greedy Order with tiering##########
  #Initialize Variables
  num.visited = 0
  to = rep(T,size)
  from = rep(T,size)
  trails = matrix(T,size,size)
  diag(trails)=F
  tours = matrix(0,size,size)
  greedloop = rep(0,size)
  GreedyOrder = rep(0,size)
  tiering = rep(0,size)
  tier = 1
  while (num.visited<size-1) {
    current.distances<-Data[]
    #current.distances[!from] = Inf
    current.distances[,!to] = Inf
    current.distances=ifelse(trails==F,Inf,current.distances)
    numintier = 0
    #go though every node
    for (i in 1:size) {
      #if node hasnt been left yet
      if (from[i]==T) {
        #find the min distance arcs
        availmin = which(current.distances[i,]==min(current.distances[i,]),arr.ind =
            T, useNames = F)[1]
        #if only min distance arc AND same as in opt tour (this can probably just be
            made availmin[1] and length removed)
        if ((availmin == Greedy_Tour[i])&&((numintier+num.visited)<(size-1))) {
          #update number in tier
          numintier = numintier + 1
          #store connection
          greedloop[numintier]= i
        }
      }
    }
    #loop through connections in tier
    for (j in 1:numintier) {
      #Perfrorm greed tracker
      trails[greedloop[j],Greedy_Tour[greedloop[j]]]=F
      tours[greedloop[j],Greedy_Tour[greedloop[j]]]=1
      from[greedloop[j]] = F
      to[Greedy_Tour[greedloop[j]]] = F

```

```
156       #row addition Trails code
157       for (l in 1:size) {
158         if (trails[l,Greedy_Tour[greedloop[j]]]==F ) {
159           trails[l,]=trails[l,]&trails[greedloop[j],]
160         }
161       }
162
163       num.visited = num.visited + 1
164       #stroe perfect orde4r list
165       GreedyOrder[num.visited] = greedloop[j]
166       tiering[num.visited] = tier
167     }
168     tier = tier+1
169
170   }
171   #Set last arc to finalize tour
172   tours[which(from==T),which(to==T)]=1
173   #sometimes cause error due to looping structure
174   GreedyOrder[num.visited+1]=which(from==T)
175   tiering[num.visited+1]=tier
176
177 #***********   PART 3: Adaptive List    **********
178   #Re-Initialize Variables
179   order = GreedyOrder
180
181   #tuneable parameters
182
183   numswaps=1
184   goodnodes = rep(0,size)
185   alpha=startalpha
186   alphacount = 0
187   best_order = order
188   best_score = score
189   a = Sys.time()
190   #Number of list order swaps at each iteration
191
192   minvals = rep(0,size)
193   badarc = rep(0,size)
194
195   for (i in 1:size) {
196     minval = tail(sort(Data[i,],decreasing = F,index.return=T),2)
197     minvals[i] = minval$x[2]
198   }
199
200   for (p in 1:size) {
201     goodnodes[p]=length(which(Data[p,]<=minvals[p]+minvals[p]*(1/startalpha)))
202   }
203   maxgood = max(goodnodes)
204
205
206   ##################
207   for (k in 1:iter) {
208     #Initialize Variables
209     num.visited = 0
210     to = rep(T,size)
211     from = rep(T,size)
212     trails =  matrix(T,size,size)
213     diag(trails)=F
214     arcvals = rowSums(tours*Data)
215     tours = matrix(0,size,size)
216
217     havearc=F
218     while (havearc == F) {
219       badarc = which(arcvals>=(minvals+minvals*alpha))
220       if ((length(badarc)<2)||(alphacount >= intensifycrit)){
221         alpha = alpha-changealpha
```

```
        alphacount=0
        if (alpha < 0) {alpha=startalpha}
      } else {
        move1 = sample(badarc,1)
        swap1=which(order==move1)
        havearc=T
      }
    }
    move2 = sample(which(goodnodes>=alpha/startalpha*maxgood),1)
    swap2 = which(order==move2)
    temp = order[swap2]
    order[swap2]=order[swap1]
    order[swap1]=temp

    #While statement
    while (num.visited<size-1) {

      current.distances<-Data[,order[num.visited+1]]
      current.distances[!to]=Inf
      current.distances[!trails[order[num.visited+1],]]=Inf
      nextTownToVisit = as.integer(which(current.distances==min(current.distances),
          arr.ind = T,useNames = F)[1])#In case of ties, take just the first

      nextTownToVisit = c(order[num.visited+1], nextTownToVisit)
      trails[nextTownToVisit[1],nextTownToVisit[2]]=F
      tours[nextTownToVisit[1],nextTownToVisit[2]]=1
      from[nextTownToVisit[1]] = F
      to[nextTownToVisit[2]]= F

      #row addition Trails code
      listarc = which(trails[,nextTownToVisit[2]]==F)
      for (i in 1:length(listarc)) {
        trails[listarc[i],]=trails[listarc[i],]&trails[nextTownToVisit[1],]
      }
      num.visited = num.visited + 1
    }
    #Set last arc to finalize tour
    tours[which(from==T),which(to==T)]=1
    score=sum(tours*Data)



    if (score<=best_score) {
      if (score<best_score) {
        alpha=startalpha
        alphacount=0
      }
      best_score=score
      best_tour=tours
      best_order=order
    } else {
      order = best_order
      alphacount = alphacount + 1
    }

  }
  PLGRS_BATGA[countBATGA]=best_score
  countBATGA =countBATGA+1
```

PLGRS ALL

```

TSP_Data <- read.csv("C:\\Users\\petar\\Documents\\R\\R Studio\\test48.csv",header =
    FALSE)
```

```
#Turn it into a usable matrix
Data<-as.matrix(TSP_Data)

#Get the data sets size
size<-dim(Data)[1]


iter = 50
startalpha =4
changealpha =.5
intensifycrit = 5
greedytour = rep(1:size,1)

#***********PART 1: get Greedy Tour******************

  listarcs = matrix(0,size*(size+1)/2,3)
  tours = matrix(0,size,size)
  Degree = rep(0,size)
  Tail = rep(1:size)
  taili=0
  tailj=0
  temptaili = 0
  temptailj = 0
  #generate list of arcs column 1 is length, column two is tail, column 3 is head
  count = 1
  for (i in 1:size) {
    for (j in i:(size)) {
      listarcs[count,1] = Data[i,j]
      listarcs[count,2] = i
      listarcs[count,3] = j
      if (i==j) {listarcs[count,1]=Inf}
      count= count + 1
    }
  }
  #sort the list by length
  listarcs = listarcs[order(listarcs[,1],decreasing = FALSE), ]
  #initialize more variables
  num.visited = 0
  count = 1
  #While statement (create hamilton path)
  while (num.visited<size-1) {
    #node leaving does not have a arc leaving and node going to does not have an arc
        entering
    if ((Degree[listarcs[count,2]]<2)&&(Degree[listarcs[count,3]]<2)&&(Tail[listarcs[
        count,2]]!=listarcs[count,3])) {
      #add arc
      tours[listarcs[count,2],listarcs[count,3]]=1
      tours[listarcs[count,3],listarcs[count,2]]=1
      #if both are 0 degree
      if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==0)) {

        taili =Tail[listarcs[count,2]]
        tailj =Tail[listarcs[count,3]]
        Tail[listarcs[count,2]]=tailj
        Tail[listarcs[count,3]]=taili

      } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==0)) {

        taili =Tail[listarcs[count,2]]
        Tail[taili]=Tail[listarcs[count,3]]
        Tail[listarcs[count,3]]=taili
        Tail[listarcs[count,2]]=0

      } else if ((Degree[listarcs[count,2]]==0)&&(Degree[listarcs[count,3]]==1)) {

        tailj =Tail[listarcs[count,3]]
```

```
        Tail[tailj] =Tail[listarcs[count,2]]
        Tail[listarcs[count,2]]=tailj
        Tail[listarcs[count,3]]=0

      } else if ((Degree[listarcs[count,2]]==1)&&(Degree[listarcs[count,3]]==1)) {

        taili =Tail[listarcs[count,2]]
        tailj =Tail[listarcs[count,3]]
        Tail[taili]=tailj
        Tail[tailj]=taili
        Tail[listarcs[count,2]]=0
        Tail[listarcs[count,3]]=0
      }
      #set start to tail and current to head
      Degree[listarcs[count,2]]=Degree[listarcs[count,2]]+1
      Degree[listarcs[count,3]]=Degree[listarcs[count,3]]+1
      num.visited = num.visited+1
    }
    count = count +1
  }
  #connect hamilton path start to finish
  tours[which(Degree<2)[1],which(Degree<2)[2]]=1
  tours[which(Degree<2)[2],which(Degree<2)[1]]=1
  score=sum(tours*Data)/2

  previousnode = 0
  currentnode = 1
  for (j in 1:size) {
    nodes = which(tours[currentnode,]==1)
    if (nodes[1]!=previousnode) {
      greedytour[j] = nodes[1]
      previousnode=currentnode
      currentnode=nodes[1]
    }  else {
      greedytour[j] = nodes[2]
      previousnode=currentnode
      currentnode=nodes[2]
    }
  }

  TSP_Tour=greedytour
  TSP_Tour = c(TSP_Tour,greedytour[1])
  Greedy_Tour = matrix(0,size,1)
  prev = TSP_Tour[1]
  for (i in 1:size+1) {
    Greedy_Tour[prev] = TSP_Tour[i]
    prev = TSP_Tour[i]
  }

#**********PART 2: Get Greedy Order with tiering**********

  #Initialize Variables
  num.visited = 0
  to = rep(T,size)
  from = rep(T,size)
  trails = matrix(T,size,size)
  diag(trails)=F
  tours = matrix(0,size,size)
  greedloop = rep(0,size)
  GreedyOrder = rep(0,size)
  tiering = rep(0,size)
  tier = 1
  while (num.visited<size-1) {
    current.distances<-Data[]
    #current.distances[!from] = Inf
    current.distances[,!to] = Inf
```

```
133     current.distances=ifelse(trails==F,Inf,current.distances)
134     numintier = 0
135     #go though every node
136     for (i in 1:size) {
137       #if node hasnt been left yet
138       if (from[i]==T) {
139         #find the min distance arcs
140         availmin = which(current.distances[i,]==min(current.distances[i,]),arr.ind =
              T, useNames = F)[1]
141         #if only min distance arc AND same as in opt tour (this can probably just be
              made availmin[1] and length removed)
142         if ((availmin == Greedy_Tour[i])&&((numintier+num.visited)<(size-1))) {
143           #update number in tier
144           numintier = numintier + 1
145           #store connection
146           greedloop[numintier]= i
147         }
148       }
149     }
150     #loop through connections in tier
151     for (j in 1:numintier) {
152       #Perfrorm greed tracker
153       trails[greedloop[j],Greedy_Tour[greedloop[j]]]=F
154       tours[greedloop[j],Greedy_Tour[greedloop[j]]]=1
155       from[greedloop[j]] = F
156       to[Greedy_Tour[greedloop[j]]] = F
157
158       #row addition Trails code
159       for (l in 1:size) {
160         if (trails[l,Greedy_Tour[greedloop[j]]]==F ) {
161           trails[l,]=trails[l,]&trails[greedloop[j],]
162         }
163       }
164
165       num.visited = num.visited + 1
166       #stroe perfect orde4r list
167       GreedyOrder[num.visited] = greedloop[j]
168       tiering[num.visited] = tier
169     }
170     tier = tier+1
171
172   }
173   #Set last arc to finalize tour
174   tours[which(from==T),which(to==T)]=1
175   #sometimes cause error due to looping structure
176   GreedyOrder[num.visited+1]=which(from==T)
177   tiering[num.visited+1]=tier
178
179 #***********   PART 3: Adaptive List    **********
180   #Re-Initialize Variables
181   order = GreedyOrder
182
183   #tuneable parameters
184
185   numswaps=1
186   goodnodes = rep(0,size)
187   alpha=startalpha
188   alphacount = 0
189   best_order = order
190   best_score = score
191   a = Sys.time()
192   #Number of list order swaps at each iteration
193
194   minvals = rep(0,size)
195   badarc = rep(0,size)
196
```

```
197  for (i in 1:size) {
198    minval = tail(sort(Data[i,],decreasing = F,index.return=T),2)
199    minvals[i] = minval$x[2]
200  }
201
202  for (p in 1:size) {
203    goodnodes[p]=length(which(Data[p,]<=minvals[p]+minvals[p]*(1/startalpha)))
204  }
205  maxgood = max(goodnodes)
206
207
208  ###################
209  for (k in 1:iter) {
210    #Initialize Variables
211    num.visited = 0
212    to = rep(T,size)
213    from = rep(T,size)
214    trails =  matrix(T,size,size)
215    diag(trails)=F
216    arcvals = rowSums(tours*Data)
217    tours = matrix(0,size,size)
218
219    type = sample(3,1)
220
221    if (type == 1) {
222
223      havearc=F
224      while (havearc == F) {
225        badarc = which(arcvals>=(minvals+minvals*alpha))
226        if ((length(badarc)<2)||(alphacount >= intensifycrit)){
227          alpha = alpha-changealpha
228          alphacount=0
229          if (alpha < 0) {alpha=startalpha}
230        } else {
231          move1 = sample(badarc,1)
232          oldloc=which(order==move1)
233          if (oldloc!=1) {
234            havearc=T
235          }
236        }
237      }
238      newloc=sample(oldloc-1,1)
239      if (newloc==1) {
240        if (oldloc==size) {
241          temporder = c(move1,order[1:size-1])
242          order=temporder
243        } else {
244          temporder = c(move1,order)
245          order = c(temporder[1:(oldloc)],temporder[(oldloc+2):(size+1)])
246        }
247      } else if (oldloc==size){
248        temporder = c(order[1:newloc-1],move1,order[newloc:size])
249        order = c(temporder[1:(oldloc)])
250      } else {
251        temporder = c(order[1:newloc-1],move1,order[newloc:size])
252        order = c(temporder[1:(oldloc)],temporder[(oldloc+2):(size+1)])
253      }
254
255    } else if (type == 2) {
256      for (j in 1:numswaps) {
257        swap1 = sample(size,1)
258        swap2 = sample(size,1)
259        temp = order[swap2]
260        order[swap2]=order[swap1]
261        order[swap1]=temp
262      }
```

```
263     } else if (type == 3) {
264
265       havearc=F
266       while (havearc == F) {
267         badarc = which(arcvals>=(minvals+minvals*alpha))
268         if ((length(badarc)<2)||(alphacount >= intensifycrit)){
269           alpha = alpha-changealpha
270           alphacount=0
271           if (alpha < 0) {alpha=startalpha}
272         } else {
273           move1 = sample(badarc,1)
274           swap1=which(order==move1)
275           havearc=T
276         }
277       }
278       move2 = sample(which(goodnodes>=alpha/startalpha*maxgood),1)
279       swap2 = which(order==move2)
280       temp = order[swap2]
281       order[swap2]=order[swap1]
282       order[swap1]=temp
283     }
284
285     #While statement
286     while (num.visited<size-1) {
287
288       current.distances<-Data[,order[num.visited+1]]
289       current.distances[!to]=Inf
290       current.distances[!trails[order[num.visited+1],]]=Inf
291       nextTownToVisit = as.integer(which(current.distances==min(current.distances),
              arr.ind = T,useNames = F)[1])#In case of ties, take just the first
292
293       nextTownToVisit = c(order[num.visited+1], nextTownToVisit)
294       trails[nextTownToVisit[1],nextTownToVisit[2]]=F
295       tours[nextTownToVisit[1],nextTownToVisit[2]]=1
296       from[nextTownToVisit[1]] = F
297       to[nextTownToVisit[2]]= F
298
299       #row addition Trails code
300       listarc = which(trails[,nextTownToVisit[2]]==F)
301       for (i in 1:length(listarc)) {
302         trails[listarc[i],]=trails[listarc[i],]&trails[nextTownToVisit[1],]
303       }
304       num.visited = num.visited + 1
305     }
306     #Set last arc to finalize tour
307     tours[which(from==T),which(to==T)]=1
308     score=sum(tours*Data)
309
310     if (score<=best_score) {
311       if (score<best_score) {
312         alpha=startalpha
313         alphacount=0
314       }
315       best_score=score
316       best_tour=tours
317       best_order=order
318     } else {
319       order = best_order
320       alphacount = alphacount + 1
321     }
322
323   }
```

Bibliography

1. B. Hopkins and R. J. Wilson, "The Truth about Konigsberg," *The College Mathematics Journal*, vol. 3, no. 1, pp. 198–207, 2004.

2. C. A. Tovey, "Tutorial on computational complexity," *Interfaces*, vol. 32, no. 3, pp. 30–61, 2002.

3. "Mixed-Integer Programming (MIP) - A Primer on the Basics." `http://www.gurobi.com/resources/getting-started/mip-basics`. Accessed: 2018-11-15.

4. M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. Hoboken, New Jersey: John Wiley & Sons, Inc, 4th ed., 2010.

5. D. Applegate, R. Bixby, V. Vhvatal, and W. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton, New Jersey: Princeton University Press, 2006.

6. C. Rego, D. Gamboa, F. Glover, and C. Osterman, "Traveling salesman problem heuristics: Leading methods, implementations and latest advances," *European Journal of Operational Research*, vol. 211, no. 3, pp. 427–441, 2011.

7. E. A. Silver, "An overview of heuristic solution methods," *Journal of the Operational Research Society*, vol. 55, no. 9, pp. 936–956, 2004.

8. S. H. Zanakis and J. R. Evans, "Heuristic "Optimization": Why, When, And How to Use It," *Interfaces*, vol. 11, no. 5, pp. 84–91, 1981.

9. J. J. Bentley, "Fast Algorithms for Geometric Traveling Salesman Problems," *ORSA Journal on Computing*, vol. 4, no. 4, pp. 387–411, 1992.

10. S. Wang, R. Weizhen, and Y. Hong, "A Distance Based Algorithm for Solving the Traveling Salesman Problem," *Operational Research*, 2018.

11. S. Cook, "The Complexity of Theorem Proving Procedures," *Proceedings of the Third Annual ACM Symposium*, pp. 151–158, 1971.

12. A. Khan, M. Khan, and Iqbal Muneeb, "Multilevel Graph Partitioning Scheme To Solve Traveling Salesman Problem," in *2012 Ninth International Conference on Information Technology - New Generations*, (Las Vegas), pp. 458–463, 2012.

13. G. Dantzig, R. Gulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Operations Research*, vol. 2, pp. 393–410, 1952.

14. I. Heller, "On the problem of the shortest path between points.," *Bulletin of the American Mathematical Society*, vol. 59, pp. 551–551, 1953.

15. H. W. Kuhn, "On certain convex polyhedra.," *Bulletin of the American Mathematical Society*, vol. 61, pp. 557–558, 1955.

16. H. Okano, S. Misono, and K. Iwano, "New TSP Construction Heuristics and Their Relationships to the 2-Opt," *Journal of Heuristics*, vol. 5, no. 1, pp. 71–88, 1999.

17. M. M. Flood, "The Traveling Salesman Problem," *Operations Research*, vol. 4, no. 1, pp. 61–75, 1956.

18. M. F. Dacey, "Selection of an Initial Solution for The Traveling-Salesman Problem," *Operations Research*, vol. 8, no. 1, pp. 133–134, 1960.

19. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An Analysis of Several Heuristics For The Traveling Salesman Problem," *SIAM Journal On Computing*, vol. 6, no. 3, pp. 563–581, 1977.

20. C. Papadimitiou and K. Steiglitz, *Combinatorial Optimization*. Englewood Cliffs, NJ: Prentice-hall, 1982.

21. K. Steiglitz and P. Weiner, "Some Improved Algorithms for Computer Solution of the Traveling Salesman Problem," in *6th Annual Allerton Conference on Circuit and Systems Theory*, no. October, (Urbana), pp. 814–821, 1968.

22. M. Held and R. Karp, "The traveling salesman problem and minimum spanning trees," *Operations Research*, vol. 18, no. 6, pp. 1138–1162, 1970.

23. R. Hassin and A. Keinan, "Greedy heuristics with regret, with application to the cheapest insertion algorithm for the TSP," *Operations Research Letters*, vol. 36, no. 2, pp. 243–246, 2008.

24. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

25. V. Černý, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of Optimization Theory and Applications*, vol. 45, no. 1, pp. 41–51, 1985.

26. R. W. Eglese, "Simulated annealing: A tool for operational research," *European Journal of Operational Research*, vol. 46, no. 3, pp. 271–281, 1990.

27. A. M. Turing, "Computing Machinery and Intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

28. D. B. Fogel, "An Evolutionary Approach to the Traveling Salesman Problem," *Biological Cybernetics*, vol. 60, no. 2, pp. 139–144, 1988.

29. C. R. Reeves, "Genetic Algorithms for the Operations Researcher," *INFORMS Journal on Computing*, vol. 9, no. 3, pp. 231–249, 1997.

30. Holland J.H., *Adaption in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press, 1992.

31. P. Merz, H. Hannover, and B. Freisleben, "Memetic Algorithms for the Traveling Salesman Problem," *Complex Systems*, vol. 13, no. 4, pp. 297–345, 2001.

32. S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling-Salesman Problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, 1973.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
21-03-2019	Master's Thesis	Sept 2017 – Mar 2019

4. TITLE AND SUBTITLE
Solving the Traveling Salesman Problem Using Ordered-Lists

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Jackovich, Petar D, 1st Lt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management
(AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENS-MS-19-M-127

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Statement A. Approved for Public Release; distribution unlimited.

13. SUPPLEMENTARY NOTES
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT
The arc-greedy heuristic is a constructive heuristic utilized to build an initial, quality tour for the Traveling Salesman Problem (TSP). There are two known sub-tour elimination methodologies utilized to ensure the resulting tours are viable. This thesis introduces a third novel methodology, the Greedy Tracker (GT), and compares it to both known methodologies. Computational results are generated across multiple TSP instances. The results demonstrate the GT is the fastest method for instances below 400 nodes while Bentley's Multi-Fragment maintains a computational advantage for larger instances.

A novel concept called Ordered-Lists is also introduced which enables TSP instances to be explored in a different space than the tour space and demonstrates some intriguing properties. While computationally more demanding than its tour space counterpart, the solution quality advantages, as well as a possibly higher proportion of optimal occurrences, when optimality is achievable via the ordered-list space, warrants further investigation of the space. Three meta-heuristics that leverage the ordered-list space are introduced. Testing results indicate that while at a severe iteration disadvantage, these methodologies benefit from using the ordered-list space which yields a higher per iteration improvement rate.

15. SUBJECT TERMS
Traveling Salesman Problem, Greedy Heuristic, Multi-Fragment, Ordered-List, Ordered-Greedy

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Lt Col Bruce A. Cox, AFIT/ENS
U	U	U	UU	101	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636 x4510 Bruce.Cox@Afit.edu