A Heuristic for Solving the Bottleneck Traveling Salesman Problem

John LaRusic
Bachelor of Computer Science candidate
Honours in theory and computation
University of New Brunswick

Supervisors:
Dr. Eric Aubanel
UNB Faculty of Computer Science

Dr. Abraham Punnen
UNB Saint John Faculty of Mathematical Sciences

Abstract

The Bottleneck Traveling Salesman Problem (BTSP) asks us find the Hamiltonian cycle in a graph whose largest edge cost (weight) is as small as possible. Studies of this problem have been limited to a few preliminary results. A heuristic algorithm was constructed using well known Traveling Salesman Problem (TSP) heuristics. Experimentally, it was found that computing the bottleneck biconnected spanning subgraph (BBSSP) for the problem coupled with a single call with the Lin-Kernigan (LK) TSP heuristic was sufficient to solve BTSP to optimality for the majority of graphs in the TSPLIB problem library as well as for random problems. Otherwise, the BBSSP and LK heuristic provided a lower bound and upper bound on the solution respectfully. A binary search was then performed, finding Hamiltonian cycles using the LK heuristic, to converge to a solution, although not necessarily to optimality. It was also found that introducing randomness into the costs of our graphs provided us with better results with the LK heuristic. These results allowed us to solve BTSP on all but four problems from the TSPLIB problem library.

Table of Contents

continued…

Table of Figures

1. Introduction

Many readers will be familiar with the Traveling Salesman Problem (TSP) - a problem that, in addition to resisting political correctness, has been very well studied in the field of combinatorial optimization. In layman's terms, a salesman wants to find the shortest tour through a number of cities (which will presumably save him time and money). A related problem is the Bottleneck Traveling Salesman Problem (BTSP) where our salesman wants to spend as little time traveling between any pair of cities (perhaps he gets carsick easily).

We formulate BTSP as a graph problem on a complete graph $G = (V, E)$ and a cost matrix C where we wish to find the Hamiltonian cycle whose largest edge cost (weight) is minimized. A more formal definition of this problem (and supplementary theory) is given in section 2.

The origins of BTSP are fuzzy, but Gilmore and Gomory [8] discussed a specific case of the original problem. Garfinkel and Gilbert [7] discussed the general problem in relation to machine scheduling. Certainly, BTSP has not been as well researched as TSP. Whereas TSP has enjoyed the attention of many researchers in solving various problems, BTSP results have been limited to problem sizes less than or equal to 200 vertices [4, 7].

2. Background and Theory

This section covers some basic definitions and theorems that are important for studying our problem. We assume the reader has some basic graph theory behind them. Knowledge of problem complexity is also helpful.

We have three problems we need to define:

Definition 2.1: A *Hamiltonian cycle* (sometimes known as a Hamiltonian circuit or tour) is a cycle $\pi = (v_1, v_2, ..., v_n, v_1)$ through a graph $G = (V, E)$ that visits every vertex in V exactly once.

Definition 2.2: Given a graph $G = (V, E)$, a cost matrix C, and a collection of Hamiltonian cycles Π of G, the *Traveling Salesman Problem* (TSP) solution is the cycle $\pi = (v_1, v_2, ..., v_n, v_1) \in \Pi$ such that:

$$\min\{C(v_n, v_1) + \sum_{i=1}^{n-1} C(v_i, v_{i+1})\}$$

(The TSP solution to a graph is the Hamiltonian cycle whose total cycle cost is minimized.)

Definition 2.3: Given a graph $G = (V, E)$, a cost matrix C, and a collection of Hamiltonian circuits Π of G, the *Bottleneck Traveling Salesman Problem* (BTSP) solution is the cycle $\pi = (v_1, v_2, ..., v_n, v_1) \in \Pi$ such that:

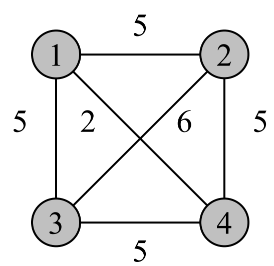
$$\min\{\max\{C(v_n, v_1) \cup \{C(v_i, v_{i+1}) \text{ for } i = 0,1,...,n-1\}\}\}$$

(The BTSP solution to a graph is the Hamiltonian cycle whose largest edge cost is minimized.)

This paper refers to the largest edge cost in a BTSP solution as its bottleneck value. It is also known as the objective value of BTSP.

BTSP (and TSP for that matter) applies to both directed (digraphs) and undirected graphs. In literature, BTSP on undirected graphs is known as Symmetric BTSP, and on directed graphs it is known as Asymmetric BTSP. We will be concentrating solely on Symmetric BTSP for this report, but much of what is discussed here can be extended to solving Asymmetric BTSP.

TSP and BTSP solutions are not necessarily unique. Finding all the TSP and BTSP solutions to a graph is a tedious and probably pointless exercise, so we are happy to limit ourselves to one solution. We also note that the TSP solution and BTSP solutions are not necessarily equivalent. Here is a simple example where we see the TSP tour differs from the BTSP tour:



TSP Tour: 1, 4, 2, 3, 1 (length: 18)

BTSP Tour: 1, 2, 4, 3, 1 (length: 20)

Figure 2.1: A comparison of a TSP tour to a BTSP tour

Let us now say something about the complexity of all three problems introduced at the beginning of this section. For readers who are not familiar with the idea of complexity classes, the general idea is that solving any of these problems is expensive in terms of the total number of operations that need to be performed. As the graphs we want to consider become larger, the number of operations needed grows exponentially. As a result, naïve algorithms for tacking these sorts of problems will only finish for small graphs.

> **Theorem 2.4**: TSP belongs to the NP-complete complexity class.
> **Theorem 2.5**: BTSP belongs to the NP-complete complexity class.
> **Theorem 2.6**: The problem of finding all the Hamiltonian cycles in a sparse graph belongs to the NP-complete complexity class.

For proofs of all three theorems, please see appendix B in Gutin and Punnen's book on TSP [17] for the appropriate proofs.

For our purposes we will assume that all the graphs we study, unless otherwise noted, are complete.

> **Definition 2.7**: A *complete graph* is a graph $G = (V, E)$ such that there exists an edge $(u, v) \in E$ for all $u, v \in V \quad u \neq v$.
> **(**Each pair of vertices in the graph is connected by an edge.)

To make a graph complete, simply add edges to a graph with a very large edge cost between unconnected pairs of vertices until the graph is complete. Any TSP or BTSP solution will avoid those edges if it can. If one of these edges exists in the final TSP or BTSP solution then no Hamiltonian cycles exist in the original graph.

> **Corollary 2.8**: For a graph with *n* vertices there are $(n-1)!/2$ Hamiltonian cycles in an undirected complete graph.
> *Proof*: There are $\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)!}{2}$ ways to pick *n* pairs of vertices.
> Because we are dealing with a cycle of *n* vertices, the order does not matter so we can divide out a factor of *n*.

It might seem alarming that the number of Hamiltonian cycles we could consider rapidly grows as we add another vertex to the complete graph. However, the techniques we will later develop do not depend on the number of edges or number of candidate Hamiltonian cycles in a graph, but solely on the number of vertices. Therefore, it is convenient to deal with complete graphs. Even if the original non-complete graphs had only a handful of Hamiltonian cycles, finding such cycles is a hard problem anyways (as proven by theorem 2.8), so we gain no advantages by dealing with sparse graphs.

3. Algorithms

As mentioned in the previous section, we are concentrating on solving BTSP on undirected, complete graphs. In this section we will discuss the TSP algorithms we will use in solving BTSP, algorithms for finding an upper and lower bound on the bottleneck solution of BTSP, and finally an algorithm that attempts to find BTSP solutions. The word "heuristic" is thrown around quite a bit as you will see. For our purposes, it indicates an algorithm that does not guarantee an optimal solution. Heuristics can be though of as good guesses for a particular problem.

Pseudo-code is used to detail how each algorithm works. All arrays are 0-indexed (that is they start counting at 0 instead of 1) as is common in C like programming languages. The notation used often refers to a set of vertices and a cost matrix. The set of vertices can be thought of as being the numbers from 0 to n-1, where n is the number of vertices in the graph. Since we are working with complete graphs, except where noted, we ignore the set of edges that normally accompany graph structures. We instead rely extensively on a cost matrix C, where the entry $C[u, v]$ is the edge cost between the pair of vertices u and v.

As is convention when dealing with the complexity of graph algorithms, n will equal the number of vertices in the graph, while m will equal the number of edges in the graph. Since we are mostly dealing with complete graphs, $m = n^2$ except where noted. As well, all logarithm functions are assumed to be in base 2.

3.1 TSP Algorithms

TSP has been a very popular problem of study for years. As a result many good algorithms have been developed to tackle the problem. One of the advantages of our approach to solving BTSP is that we attempt to leverage this work. This next section gives a brief overview of a heuristic for approximating TSP tours and an exact algorithm for definitively solving TSP tours.

3.1.1 The Lin-Kernighan Heuristic

In their 1973 paper [12], Lin and Kernighan detailed a popular algorithm that today is considered to be one of the best heuristics for finding near-optimal TSP solutions [9]. It has been used in finding optimal solutions of up to 24,978 vertices [1] and has produced estimates that are within 0.068% of the optimal tour for a problem with 1,904,711 vertices [6].

The LK heuristic is complicated, and a thorough discussion of its workings is beyond the scope of this paper. A good resource that details the heuristic as well as submits an implementation is Helsgaun's paper [9]. It should be noted that the quality of the output (that is to say, how close the result is to the optimal solution) is affected by the input.

Helsgaun [9] performed some experimental studies on the LK heuristic and found an average running time complexity of $O(n^{2.2})$. However, the running time is not strictly dependent on the number of vertices but also the structure of the graph. For example, cost matrices that satisfy the triangle inequality (that is to say for $\forall x, y, z \in V$, $C[x,z] \leq C[x,y] + C[y,z]$ for a given set of vertices V and cost matrix C) seem to require more time to solve, but their results are more accurate. To reflect this uncertainty, we'll parameterize the time complexity of algorithms that use the LK heuristic with an oracle (e.g. $O(\log_2 n \cdot LK)$).

For our purposes we will pass the LK heuristic a set of vertices and a cost matrix. It will return a tour and its length:

Algorithm LK-Heuristic(V, C):
Inputs: A set of vertices V and a cost matrix C.
Outputs: An ordered pair (T, l) where T is the best Hamiltonian cycle the Hamiltonian cycle the heuristic could find of length l.

3.1.2 Integer Linear Programming with Branch-and-Cut techniques

Solving TSP to optimality is a computationally intensive problem as proven by the fact it belongs to the class of NP-complete problems. A great deal of research has been done to try and find the best way to yield optimal results while minimizing number of calculations. The technique that seems to have had the most success is formulating TSP as a linear programming problem using what's known as Branch-and-Cut techniques. Its origins can be traced back to 1952 with the research of Dantzig, Fulkerson, and Johnson who solved a 52-vertex problem by hand [5]. The branch-and-cut technique was most recently used by Applegate, Bixby, Chvátal, Cook, and Helsgaun in 2004 to confirm a 24,978-vertex problem [1].

Much like the LK heuristic, we will refer the reader to Naddef's chapter [15] for a complete discussion of the method. Because it as an algorithm that produces an optimal result, we expect the same result no matter the input.

3.2 Lower Bound Heuristics

There are two lower bound heuristics we will examine, the largest of which shall be a lower bound on our problem. Both rely on the idea that a Hamiltonian cycle for a graph will have two edges incident on every vertex. The proof of this is left as an exercise to the reader.

3.2.1 2-Max Bound Heuristic

For this heuristic, described by Kabadi and Punnen [11], we simply calculate the second smallest cost incident on every vertex and take the largest of all these costs. In the context of BTSP, the Hamiltonian cycle of a graph will use at best the smallest and second smallest cost edge incident on every vertex to form a cycle. A lower bound on the bottleneck value will therefore be the largest of these edges. This is known as the 2-Max Bound (2MB). The algorithm is given below and clearly runs in $O(n^2)$ time for complete graphs.

Algorithm 2-Max-Bound(V, C):

Inputs: A set of vertices V and a cost matrix C.
Output: A lower bound on the bottleneck value for BTSP

```
max ← −∞
alpha ← +∞            // smallest edge
beta ← +∞             // 2nd smallest edge
for all u ∈ V
    for all v ∈ V \ {u}
        if C[u,v] < alpha then
            beta ← alpha
            alpha ← C[u,v]
        else if C[u,v] < beta
            beta ← C[u,v]
        end if
    end for
    if max < beta then
        max ← beta
    end if
end for
return max
```

3.2.2 Biconnected Bottleneck Spanning Sub Graph Heuristic

A graph is biconnected if there is no vertex that exists such that its removal will disconnect the graph. A tour, by definition, is biconnected, so finding the minimum edge cost that still allows for a biconnected graph will be a lower bound on the BTSP solution. We refer to this algorithm at the Biconnected Bottleneck Spanning Sub Graph Problem (BBSSP). A simple way of solving this problem was introduced by Parker and Rardin [16] and will be the implementation discussed here.

To find what this cost is, we perform a binary search over an ordered array of unique edge costs. Taking the median value b we see if the graph is biconnected if we consider only edges of cost less than or equal to b. If the graph is biconnected at that value, then we lower the upper bound to b and repeat. If the graph is not biconnected, then we raise the lower bound to the next cost after b (as we've already shown that b is no good, so we try the next lowest cost as the lower bound).

The algorithm for testing biconnectivity of a graph is well known, so we leave the implementation up to the reader. For those unfamiliar with how biconnectivity is tested, we refer to Dave Mount's excellent lecture on the subject [14]. Please note this is the only time where we deal with a set of edges and ignore the cost matrix.

Algorithm Graph-Is-Biconected(V, E):
    Inputs: A set of vertices V and a set of edges E.
    Output: True if the graph is biconnected, false if not.

Algorithm Biconnected-Spanning-Subgraph(V, C):
    Inputs: A set of vertices V and a cost matrix C.
    Output: A lower bound on the bottleneck value for BTSP.

```
let W be an ordered array of size m consisting of the unique edge
    costs found in C
low ← 0
high ← m − 1
while low ≠ high do
        median ← ((high − low) ÷ 2) + low
        medWeight ← W[median]
        let E be an empty set of edges
        for all u ∈ V
                for all v ∈ V \ {u}
                        if C[u,v] ≤ medWeight then
                                add (u,v) to E
                        end if
                end for
        end for
        if Graph-Is-Biconnected(V, E) then
                high ← median
        else
                low ← median + 1
        end if
end while
return W[low]
```

The algorithm involves ordering the unique edge costs found in C. Given a complete graph with n vertices, there are up to $n^2 / 2$ edge costs to order, so at best the running time for ordering will be $O(n^2 \log n)$. The running time for testing biconnectivity of a graph is $O(n + m)$. The value of m will grow or shrink for each call, depending on whether we are raising the lower bound or lowering the upper bound. Certainly for a complete graph, $m \leq n^2$. Since we are doing a binary search on the ordered edge costs, we will ask the algorithm to make $O(\log n)$ biconnectivity tests. In total, the running time for this bound will be $O(n^2 \log n)$.

It should be noted that there are better ways, asymptotically speaking, of finding the BBSSP solution of a graph. The implementation given is probably the simplest to implement. Punnen and Nair [18] proposed an $O(m + n\log_2 n)$ algorithm, Timofeev [20] an $O(n^2)$ algorithm and, finally, an $O(m)$ algorithm was proposed by Manku [13].

3.2 Upper Bound Heuristics

Just as for lower bounds, we will try and find a tight upper bound on the BTSP solution. The general approach to these algorithms is to build a Hamiltonian cycle and choose the largest edge. The largest edge of any Hamiltonian cycle in a graph will be an upper bound on the bottleneck value for a BTSP solution. As before, the proof of this is left as an exercise for the reader.

3.3.1 Nearest Neighbour Heuristic

The Nearest Neighbour Heuristic (NNH) was one of the first heuristics for approximating a TSP solution. Although the quality of this heuristic is poor with respect to other heuristics available to us, it is simple to implement and runs quickly. We pick a starting node and move to its nearest neighbour, repeating until we form a cycle. The largest edge weight in this cycle will be an upper bound on the bottleneck value. This algorithm clearly runs in $O(n^2)$ time.

Algorithm Nearest-Neighbour(V, C):

Inputs: A set of vertices V and a cost matrix C
Outputs: An upper bound on the bottleneck value for BTSP

```
mark all vertices in V as unvisited
let s be any starting vertex
max ← −∞
u ← s
while there are unvisited vertices in V
    mark u as visited
    min ← +∞
    nn ← NULL
    if there are no unvisited vertices then
        nn ← s          // Connect back with start of tour
    else
        // Find the nearest neighbour
        for all unvisited vertices v ∈ V \ {u}
            if C[u,v] < min then
                min ← C[u,v]
                nn ← v
            end if
        end for
    end if
    if C[u,nn] > max then
        max ← C[u,nn]
    end if
    u ← nn
end while
return max
```

3.3.2 Node Insertion Heuristic

The Node Insertion Heuristic (NIH) attempts to gradually build a tour one random vertex at a time. Starting with a three-vertex cycle, each new randomly chosen vertex is inserted in what is thought to be the best possible place. In an attempt to keep the number of comparisons to a minimum we keep track of the largest and second largest cost in the current tour.

```
Algorithm Node-Insertion(V, C):
        Inputs: A set of vertices V and a cost matrix C
        Outputs: An upper bound on the bottleneck value for BTSP

        mark all vertices in V as unvisited
        let T = {{u,v},{v,w},{w,u}} be a tour of three random vertices
            where u,v,w ∈ V
        alpha ← max{{u,v},{v,w},{w,u}}
        beta ← second largest of{{u,v},{v,w},{w,u}}
        mark u, v, w as visited
        while there are unvisited vertices in V
                let w be a random vertex from V
                minVal ← +∞
                for all {u,v} ∈ T
                        if C[u,v] = alpha then
                                largest ← max{beta, C[u,w], C[w,v]}
                        else
                                largest ← max{alpha, C[u,w], C[w,v]}
                        end if
                        if largest < minVal then
                                minVal ← largest
                                minSpot ← {u,v}
                        end if
                end for
                insert w into tour between the edge {u,v}
                mark w as visited
                if minVal > alpha then
                        beta ← alpha
                        alpha ← minVal
                else if minVal > beta then
                        beta ← minVal
                end if
        end while
        return alpha
```

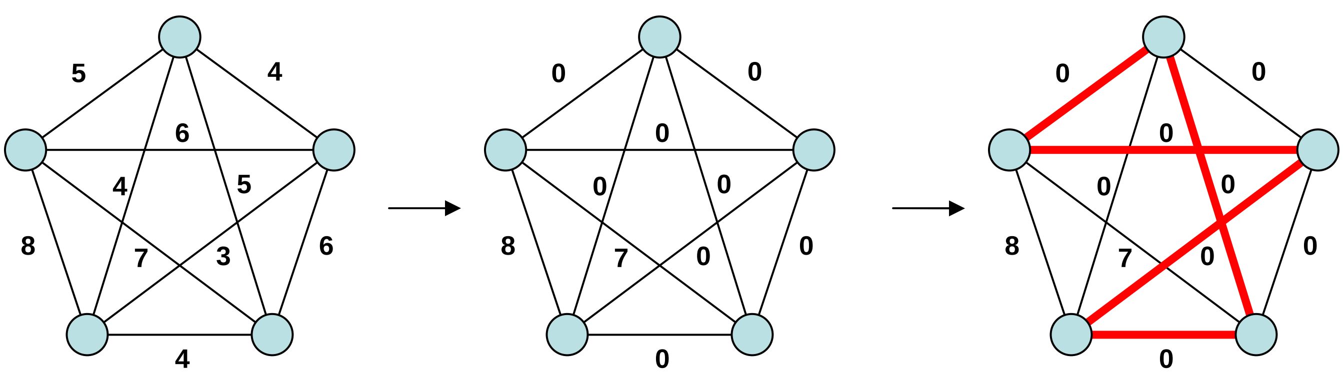
This algorithm clearly runs in $O(n^2)$ time. Because of the random nature of this algorithm, we could possibly improve the upper bound result we get from it by running the algorithm more than once.

3.3.3 LK Tour Heuristic

The two previous upper bound heuristics make efforts to build reasonable tours that can help define a good upper bound on the bottleneck value. The advantage of both methods is that they run reasonably quickly, even for large graphs. But since an upper bound can be found from any Hamiltonian cycle, it is reasonable to assume that the Hamiltonian cycle of a TSP solution to a graph will be a reasonably good upper-bound.

Finding the TSP solution to a graph is expensive, but we can make a very good guess with the Lin-Kernighan (LK) heuristic. As we'll see in the next section, the LK heuristic is used in our scheme for finding the BTSP solution to a graph. If a single call with the LK heuristic produces a significantly better upper-bound than either than nearest neighbour heuristic or the node-insertion heuristic, then it is certainly worth our while to spend the time.

For the cost matrix we pass the LK heuristic, we can utilize a lower bound we have already computed to help find a good TSP tour. If the resulting TSP tour length equals zero then the upper and lower bounds are equal. Otherwise, we choose the largest edge in the tour the LK heuristic found. Figure 3.1 illustrates the idea.



The original graph. BBSSP heuristic gives a lower bound equal to 6.

New graph where costs less than a lower bound of 6 set to a new cost of 0.

TSP Tour of Length 0 Result: Lower bound equals upper bound.

Figure 3.1: An illustration of the LK Tour heuristic for finding an upper bound.

Algorithm TSP-Tour(*V*, *C, lb*):

Inputs: A set of vertices *V* a cost matrix *C*, and a lower bound *lb*.
Outputs: An upper bound on the bottleneck value for BTSP

```
let D be a new cost matrix of the same dimensions as C
for all u ∈ V
        for all v ∈ V \ {u}
                if C[u,v] ≤ lb then
                        D[u,v] ← 0
                else
                        D[u,v] ← C[u,v]
                end if
        end for
end for
(tour, length) ← LK-Heuristic(V, D)
if length = 0 then
        return lb
else
        max ← +∞
        for all {u,v} ∈ tour
                if C[u,v] ≤ max then
                        max ← C[u,v]
                end if
        end for
        return max
end if
```

3.4 Finding Hamiltonian cycles using the LK heuristic

Before we introduce either a method for finding BTSP solutions, we start by explaining how to use a TSP heuristic to make a good guess at a Hamiltonian cycle. Finding Hamiltonian cycles in a sparse graph is an NP-Hard problem and there are too many Hamiltonian cycles in a complete graph to consider (as explained in section 2). We can, however, make a good guess at whether a Hamiltonian cycle exists in a complete graph by using the Lin-Kernighan (LK) heuristic.

Suppose we wish to know whether a Hamiltonian cycle exists in a graph using only edge costs less than or equal to a value *b* in a complete graph. Given a set of vertices *V*, a cost matrix *C*, and a value *b* we construct a new cost matrix *D* as follows:

$$D[u,v] = \begin{cases} 0 & \text{if } C[u,v] \le b \\ 1 & \text{otherwise} \end{cases} \text{ for } \forall u,v \in V$$

We then run the LK-heuristic using this new cost matrix. Because we are now attempting to solve TSP, the LK-heuristic will try and minimize the total length of the tour it finds. For this reason, the LK-heuristic will try and use as many edges of cost 0 as it can. If we find a tour of length 0 then we have found definite proof of a Hamiltonian cycle using only edge weights up to the value *b*. This idea is much like the one illustrated by figure 3.1.

If we don't find a tour of length 0 then we guess that such a Hamiltonian cycle does not exists, but because we are using a TSP heuristic we cannot say conclude with any certainty that one does not exist. Therefore, we might want to make more than one attempt at finding a TSP tour of length 0. Of course, we could use an exact TSP algorithm, but we want to avoid making such expensive calls.

We refer to the above cost matrix we constructed as the Zero/One cost matrix, but here are other cost matrix formulations that we can utilize to find Hamiltonian cycles. We are interested in studying them because they might provide better solutions or run quicker for the LK heuristic. Table 3.1 lists five different cost matrix formulations, but certainly is not an exhaustive list.

If we don't find a Hamiltonian cycle on the first attempt with any of the given cost matrix formulations then we can try making additional attempts using the Zero/Random cost matrix formulation. The LK heuristic can, for lack of a better term, get stuck trying to find an optimal tour. This added element of randomness might allow it to find a better tour. This appears to be a new idea; one that Dr. Punnen has termed "shaking the cost matrix". The PR department is currently hard at work finding a more catchy term.

Name:	Zero/One Cost Matrix
Formulation:	$D[u,v] = \left\{ \begin{matrix} 0 & \text{if } C[u,v] \le b \\ 1 & \text{otherwise} \end{matrix} \right\} \text{for } \forall u,v \in V$
Notes:	A Hamiltonian cycle exists if TSP tour has length equal to 0.
Name:	Zero/Random Cost Matrix
Formulation:	$D[u,v] = \left\{ \begin{matrix} 0 & \text{if } C[u,v] \le b \\ \text{random } \mathrm{Z}^{+} & \text{otherwise} \end{matrix} \right\} \text{for } \forall u,v \in V$
Notes:	A Hamiltonian cycle exists if TSP tour has length equal to 0. The random number can any non-zero integer.
Name:	Zero/Normal Cost Matrix
Formulation:	$D[u,v] = \left\{ \begin{matrix} 0 & \text{if } C[u,v] \le b \\ C[u,v] & \text{otherwise} \end{matrix} \right\} \text{for } \forall u,v \in V$
Notes:	A Hamiltonian cycle exists if TSP tour has length equal to 0.
Name:	Normal/Infinity Cost Matrix
Formulation:	$D[u,v] = \left\{ \begin{matrix} C[u,v] & \text{if } C[u,v] \le b \\ +\infty & \text{otherwise} \end{matrix} \right\} \text{for } \forall u,v \in V$
Notes:	The positive infinity value can be any relatively large number. The new A Hamiltonian cycle exists if TSP tour has length less than positive infinity.
Name:	Ordered Position Cost Matrix
Formulation:	$D[u,v] = \left\{ \begin{matrix} \text{(position in ordered cost array)} & \text{if } C[u,v] \le b \\ +\infty & \text{otherwise} \end{matrix} \right\} \text{for } \forall u,v \in V$
Notes:	Before we use this cost matrix we need to order all the unique costs in the graph. If the cost $C[u,v] = b$ is found in position i in the ordered array, then $D[u,v] = i$. The positive infinity value can be any relatively large number. A Hamiltonian cycle exists if TSP tour has length less than positive infinity.

Table 3.1: Five different cost matrix formulations for finding Hamiltonian cycles

3.5 BTSP Binary Search Heuristic

We will order the edge weights and locate the upper and lower bounds on the bottleneck value. Attempts will then be made to find a Hamiltonian cycle using the median edge weight. If one can be found, we can lower the upper bound to the median. If one cannot be found, we can raise the lower bound to the median (plus one step). We repeat this procedure until we converge to the bottleneck value.

For the sake of clarity, we define two helper functions. The implementation of these two functions is omitted as they are standard sort and binary search methods.

Algorithm Order-Edge-Weights(*V*, *C*):
Inputs: A set of vertices *V* and a cost matrix *C*.
Outputs: An array of unique edge weights ordered from lowest to highest.

Algorithm Binary-Search-Array(*Array*, *Value*):
Inputs: An ordered *Array*, and a *Value* to search for.
Outputs: The position in *Array* where *Value* is stored.

We now define our basic algorithm for finding the BTSP solution:

Algorithm BTSP-Binary-Search(*V*, *C*, *lb*, *ub*):
Inputs: A set of vertices *V*, a cost matrix *C*, a lower bound *lb*, an upper bound *ub*.
Outputs: A BTSP tour and the bottleneck value of the graph

```
E ← OrderEdgeWeights(V, C)
low ← Binary-Search-Array(E, lb)
high ← Binary-Search-Array(E, ub)
do while low ≠ high
    median ← ((high − low) ÷ 2) + low
    medCost ← E[median]
    D ← Build-Cost-Matrix(V, C, medCost)
    (tour, length) ← LK-Herustic(V, D)
    if length = 0 then
        high ← median
        bestTour ← tour
    else
        low ← median + 1
    end if
end do
return (bestTour, W[low])
```

In the pseudo-code outlined above, only one attempt is made at finding a Hamiltonian cycle using whatever cost matrix formulation desired. This would be fine if the LK heuristic was an exact TSP solver, but in reality we will want to make additional attempts with a Zero/Random cost matrix if we cannot find a Hamiltonian cycle on the initial attempt, what we have termed "shaking the cost matrix".

In analyzing the time complexity of the algorithm, we note that the time spent searching for the bottleneck value will dominate, so the complexity of this method is $O(\log n \cdot LK)$.

After this algorithm completes, we can confirm the result using an exact TSP solver. By performing a linear search from the found bottleneck value to the lower bound value, we can confirm that Hamiltonian cycles do or do not exist for smaller bottleneck values.

4 Implementation and Testing Details

All code was written in C using the GNU GCC compiler on Red Hat Linux. The algorithms detailed in the previous section were implemented as outlined according to the pseudo-code descriptions given.

The one exception is the Node Insertion algorithm. In an effort to generate good results quickly, 10 trials were attempted, the best of which was chosen to be an upper bound. At each step, the best result was recorded. At every stage in a trial a check was made to see if the current result was worse than the best result found so far. If the answer to that question was true, then the current attempt was abandoned. This was a practical consideration, as if the current tour being built is no better than the best tour found then there is no advantage to completing the tour.

The implementation of the branch-and-cut TSP algorithm and the Lin-Kernighan heuristic in the Concorde TSP solver [2l] was used. Concorde is a well known solver for symmetric TSP. The solver is free for academic use and the full source code is available in ANSI C. Furthermore, Concorde was used to solve the largest known TSP solution at the time of writing, a 24,978 vertex problem [1]. The QSpot linear programming solver [3], written by the same authors of Concorde, was used to confirm results. QSopt is a free linear programmer that interfaces naturally with Concorde.

Our test problems mostly came from Reinelt's TSPLIB problem collection [19]. We limited testing to problems of 10,000 vertices or less. Of the remaining TSPLIB problems, we were unable to test the linhp318 problem because Concorde does not support problems with fixed edges. We also neglected to perform testing on vm1084 and vm1748 due to an oversight.

We also tested the standard random problems from the instance generation codes provided by Johnson and McGeoch [10]. These codes, used in the 8th DIMACS Implementation Challenge, allowed generation of random TSP instances that followed three different plans: uniform point, clustered points, and random distance matrices. We modified the random distance matrix generator to give us some specific random problems. These changes are explained in the next section.

Testing was carried out on UNB's 164-processor Sun V60 clustered computer, Chorus. Chorus consists of 60 slave nodes consisting of dual 2.8GHz Intel Xeon processors with 2 to 3 GB of RAM. Detailed information about the cluster can be found on UNB's ACRL site [21].

5 Experimental Results

We were interested to see how well our lower and upper bounds performed, which cost matrix formulation gave the best results, and how, if at all, shaking improved our ability to find Hamiltonian cycles. Finally, we attempted to solve as many problems from TSPLIB as we could.

5.1 Lower and Upper Bound Heuristic Analysis

5.1.1 Running Times and Accuracy on TSPLIB Problems

We examined both the accuracy and the run time of our bounds. 10 trials were carried out on each problem from our TSPLIB problem set and averaged the results to create a single result for each graph. Sample results can be found in Appendix A. Figure 5.1 summaries the run times of the lower bound heuristics, while figure 5.2 summaries the run times of the upper bound heuristics. The run times for these bounds are as expected. Unsurprisingly, the BBSSP lower bound heuristic and LK upper bound heuristic are the most expensive heuristics. The odd pattern the LK upper bound heuristic makes can be attributed to the fact that its run time is not solely dependent on the number of vertices but also the structure of the graph. This effect is noted by Helsgaun [9].

For analyzing the accuracy of each heuristic the percent error of a value of a given heuristic result was taken against the optimal solution for that particular graph. The results were plotted against the number of vertices. Figures 5.3 and 5.4 summarize the results for the lower and upper bound heuristics respectfully. Please note that the problem "brg180" was removed from the upper bound plots because of an outlier.

It appears that the BBSSP and LK tour heuristics provide extremely good bounds on the BTSP solution. In fact, for every problem attempted, with the exception of ts225, we found a lower bound equal to an upper bound, effectively finding the BTSP solution in the matter of minutes.

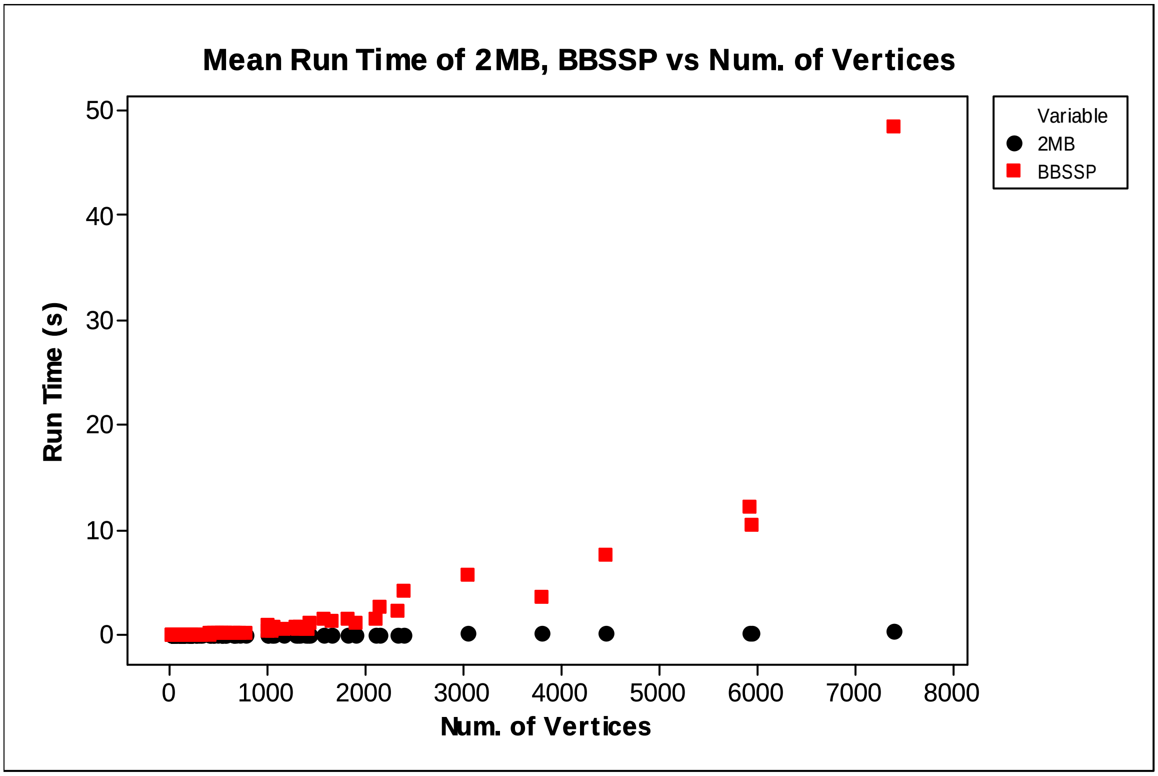


Figure 5.1: Run times of lower bound heuristics

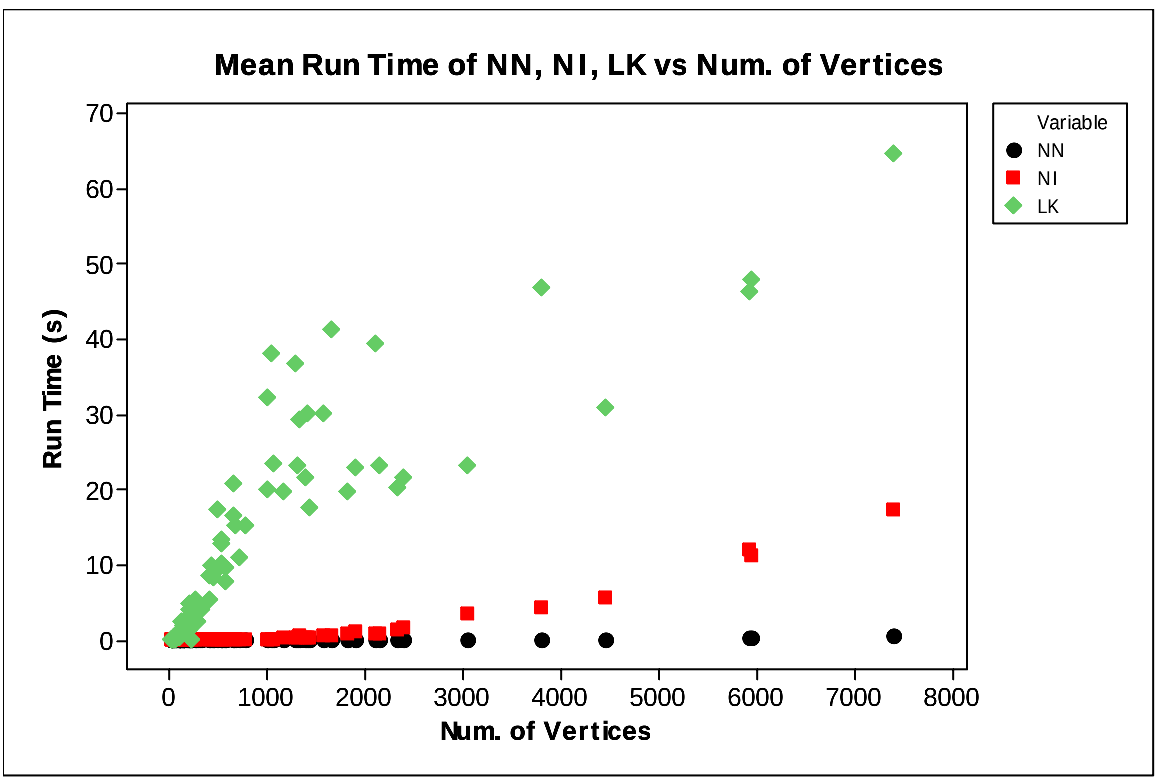


Figure 5.2: Run times of upper bound heuristics

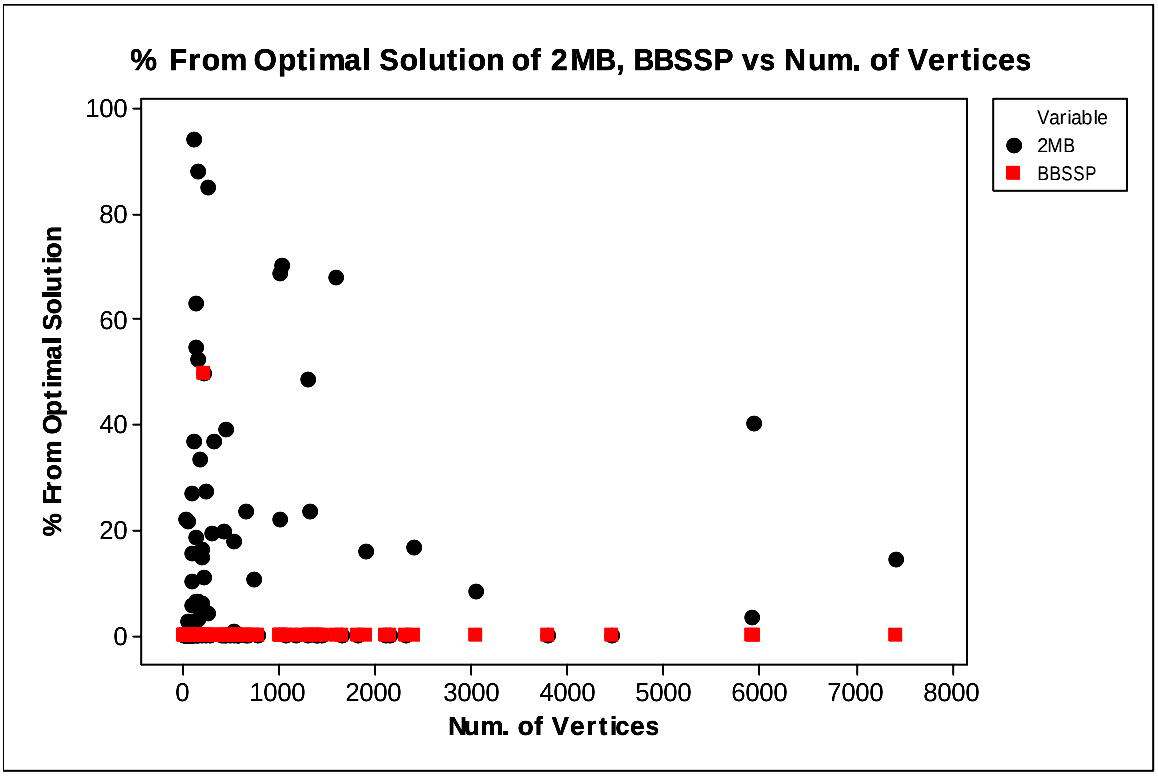


Figure 5.3a: Accuracy of lower bound heuristics (all problems)

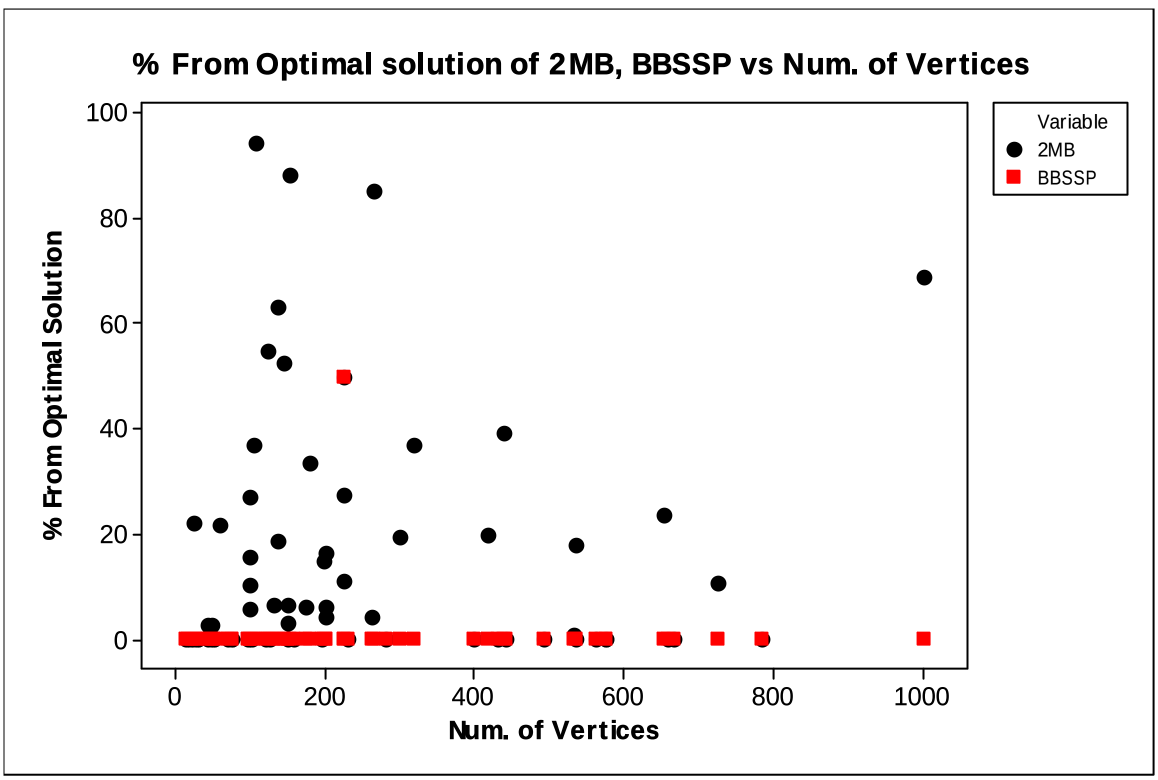


Figure 5.3b: Accuracy of upper bound heuristics (problems of 1000 vertices or less)

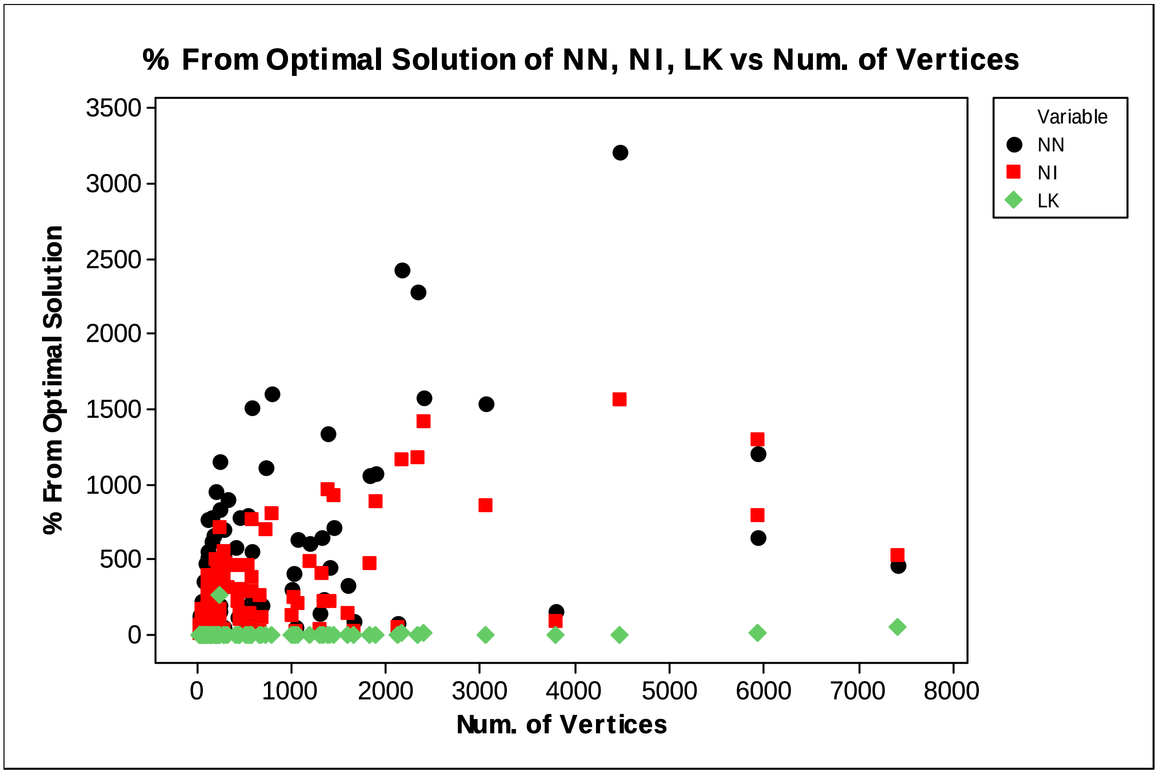


Figure 5.4a: Accuracy of upper bound heuristics (all problems)

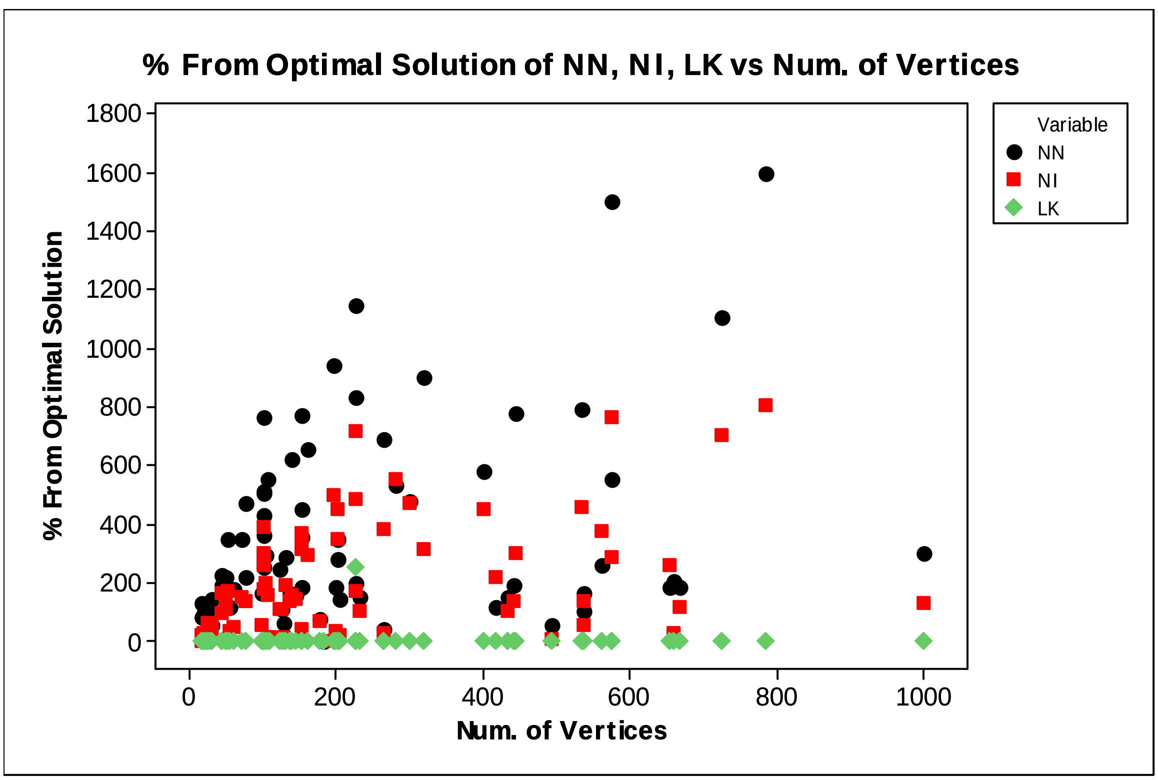


Figure 5.4b: Accuracy of upper bound heuristics (problems of 1000 vertices or less)

5.1.2 Analysis on Randomly Generated Instances

With the majority of problems of TSPLIB effectively solved with little effort we turned toward a random instance generator to hopefully give us more difficult problems. The instance generation codes we used, as already mentioned, came from the ones used in the 8th DIMACS Implementation Challenge. There is a standard set of random problems that a number of different TSP heuristics and exact solvers were asked to solve. We attempted all the random problems of 10,000 vertices or less. The results are given in Appendix C. We were once again able to easily solve every problem but one to optimality using nothing more than the BBSSP heuristic result combined with the LK tour heuristic. The one lone problem might be optimal, but no effort was made to run an exact solver on it due to its large size.

We then made an attempt to try and construct problems we hoped would have a weak lower bound. We theorized that perhaps random problems with a large range of costs might produce a weak lower bound with the BBSSP heuristic. To this end, we modified the random distance matrix generation code to use a modulo function to restrict values to a given range. We then generated a number of problems with 100, 500, 1000, 2500, 5000, and 10,000 vertices, restricting the range of costs based upon the size of the problem according to the following equations:

$$n \quad n^2 \quad 2n^2 \quad n\sqrt{n} \quad n/2 \quad n/10 \quad n\log n \quad n(\log n)^2$$

With five different seeds this gave us a total of 240 unique problems. Of these 240 problems, only one problem (with 100 vertices, range of n^2) seems to have a bottleneck solution that is not equal to the lower bound computed by the BBSSP heuristic. This one lone problem converged to the upper bound calculated by the LK tour heuristic. This solution was not confirmed with an exact solver, so it is possible a smaller bottleneck value exists. However, our solver works quite well for small instances (discussed in the next section), so it is likely this solution is optimal. Overall, it seemed that the range of costs did not affect the quality of the BBSSP heuristic.

While this result was exciting, we still wanted to find problems where the upper and lower bounds we were calculating we not tight. The solution was to construct a cost matrix we were guaranteed not to calculate tight bounds on. We once again modified the random distance matrix generation code to produce problems with cost matrices of the following form:

$$C = \begin{array}{c} \begin{array}{cc} \overbrace{\qquad A \qquad}^{\gamma \text{ columns}} & \overbrace{\quad B \quad}^{s \text{ columns}} \end{array} \\ \left[\begin{array}{c|c} \qquad A \qquad & \quad B \quad \\ \hline B^T & D \end{array}\right] \end{array} \begin{array}{l} \left.\vphantom{A}\right\} \gamma \text{ rows} \\ \left.\vphantom{B^T}\right\} s \text{ rows} \end{array}$$

A is a symmetric $\gamma \times \gamma$ matrix with entries in the range $[\alpha+1, \beta]$. B is a $\gamma \times s$ matrix with entries in the range $[0, \alpha]$. D is a symmetric $s \times s$ matrix with entries in the range $[\alpha+1, \beta]$. Furthermore, $\gamma \geq 2$, $s \geq 2$, and $\gamma + s = n$.

We generated problem instances from sizes of 100 to 2500 for various values of α, β, and γ and tried solving them with our binary search algorithm. Here are the averaged results for n = 100, α = 1000, β = 10000, and various values of γ over 5 trials:

γ	**Lower Bound**	**Upper Bound**	**Unconfirmed Solution**	**% Solution From Lower Bound**	**% Solution From Upper Bound**
10	459	1427	1357	195.64%	4.90%
20	377	1299	1285	240.85%	1.08%
30	209	1217	1211	479.43%	0.46%
40	205	1064	1064	419.02%	0.00%
50	150	150	150	0.00%	0.00%
60	173	1095	1095	532.95%	0.00%
70	160	1223	1199	649.38%	1.96%
80	234	1270	1247	432.91%	1.82%
90	466	1463	1354	190.56%	7.44%

Table 5.1: Results with specially constructed matrix

The other trials performed similarly. This small problem size is a nice to look at because we can be fairly confident in the solution found, even without running an exact TSP solver to confirm it. Without reading too much from this one sample, we notice that the solution found is certainly not equal to the lower bound as expected, except when $\gamma = s$. It seems that when the dimensions of A and B are equal that the lower bound computed by the BBSSP heuristic is the solution.

Problems of this nature are going to make our solver work harder because it will not be able to compute a tight upper and lower bound. Furthermore, without running an exact solver on these problems to confirm any solution it makes any analysis of our heuristics pointless. In order to gain any knowledge of how well our algorithms tackle these sorts of problems time is needed to run an exact solver to find an optimal solution. This is a very expensive operation, so no effort was made to confirm any of the solutions we found. We therefore leave the analysis of this particular cost matrix for another thesis.

4.2 Cost Matrix Formulation Results

We previously introduced five different ways of formulating a cost matrix to find Hamiltonian cycles in a graph. If we were using an exact TSP algorithm then all five should give us the same answer. However, the quality of solutions the LK heuristic gives us will be reliant on how we formulate the problem. Therefore, we were interested to see which formulation gave us the best results, as well as look at their respective running times.

As a result of the extremely strong lower and upper bounds we were computing on TSPLIB problems, we used the weaker 2-Max Bound and Node Insertion heuristics to ensure that our binary search method for arriving at a bottleneck solution was needed. For each problem in our TSPLIB problem set we performed 10 trials with each of the five cost matrix formulations described in table 3.1. The solutions and running times for all 10 trials were averaged. Since we had an optimum solution for all the problems in our TSPLIB problem set, we calculated the percent error from the calculated solution against the optimum solution.

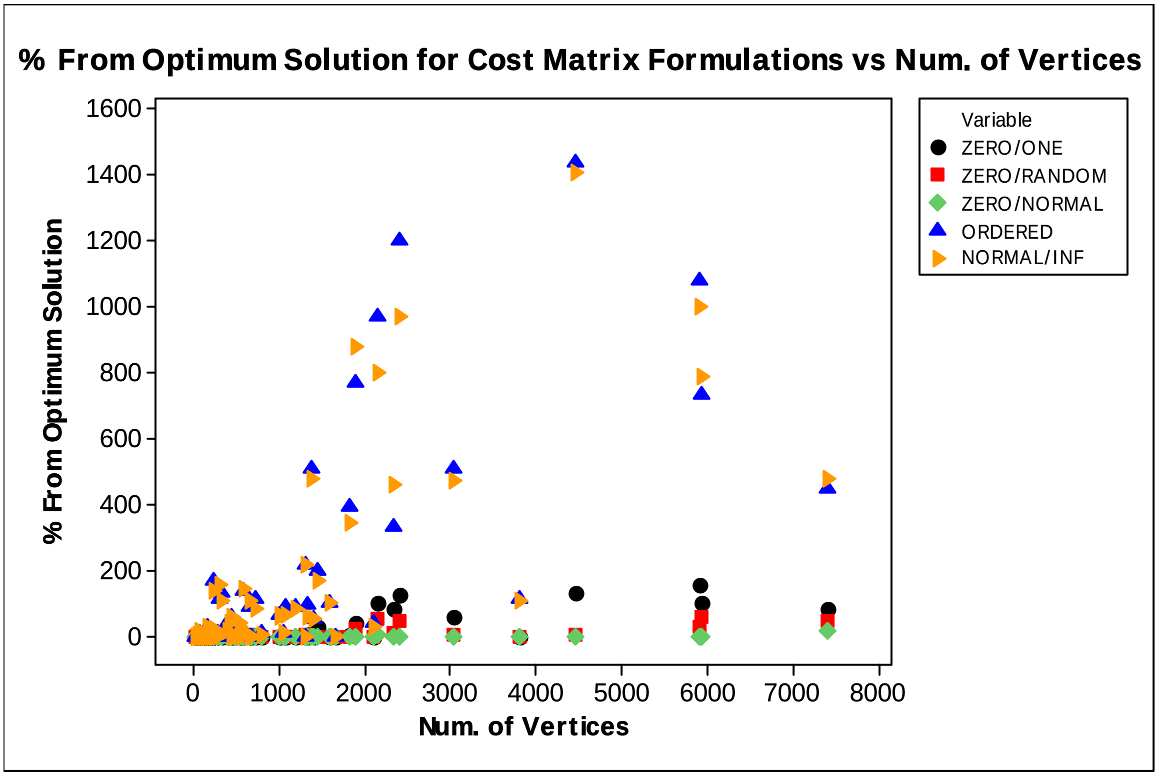


Figure 5.5a: Accuracy of cost matrix formulations (all problems)

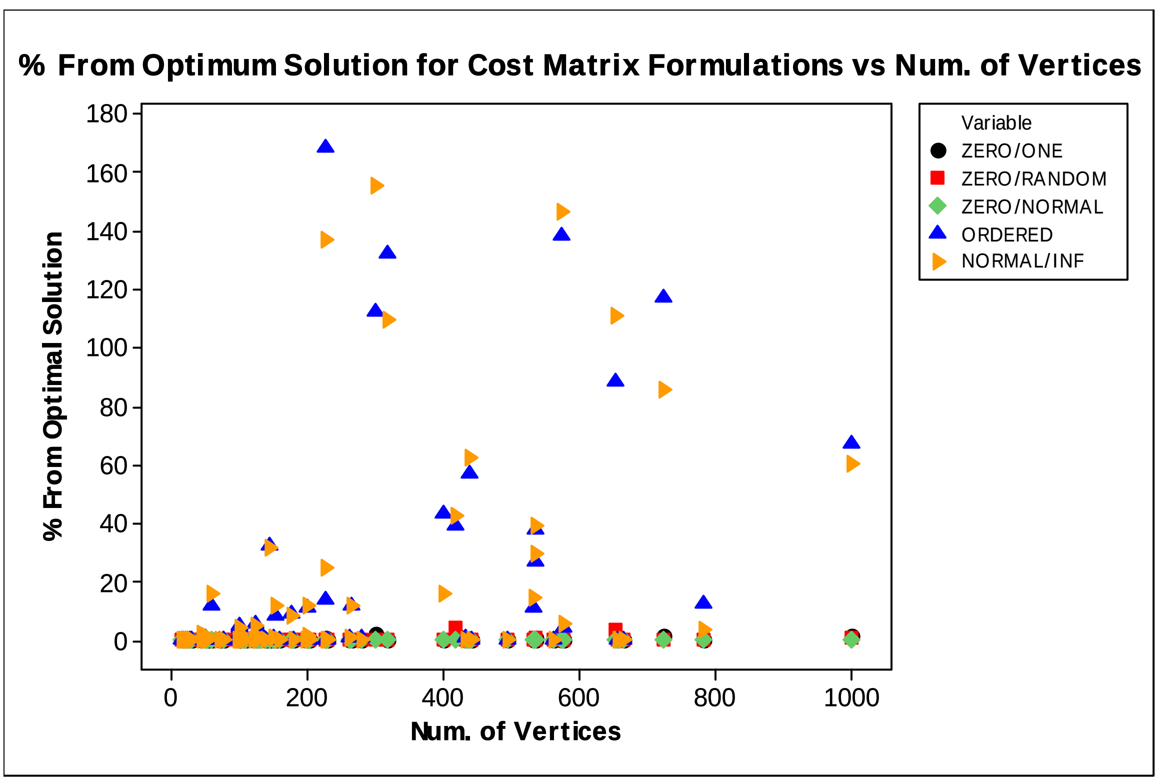


Figure 5.5b: Accuracy of cost matrix formulations (problems of 1000 vertices or less)

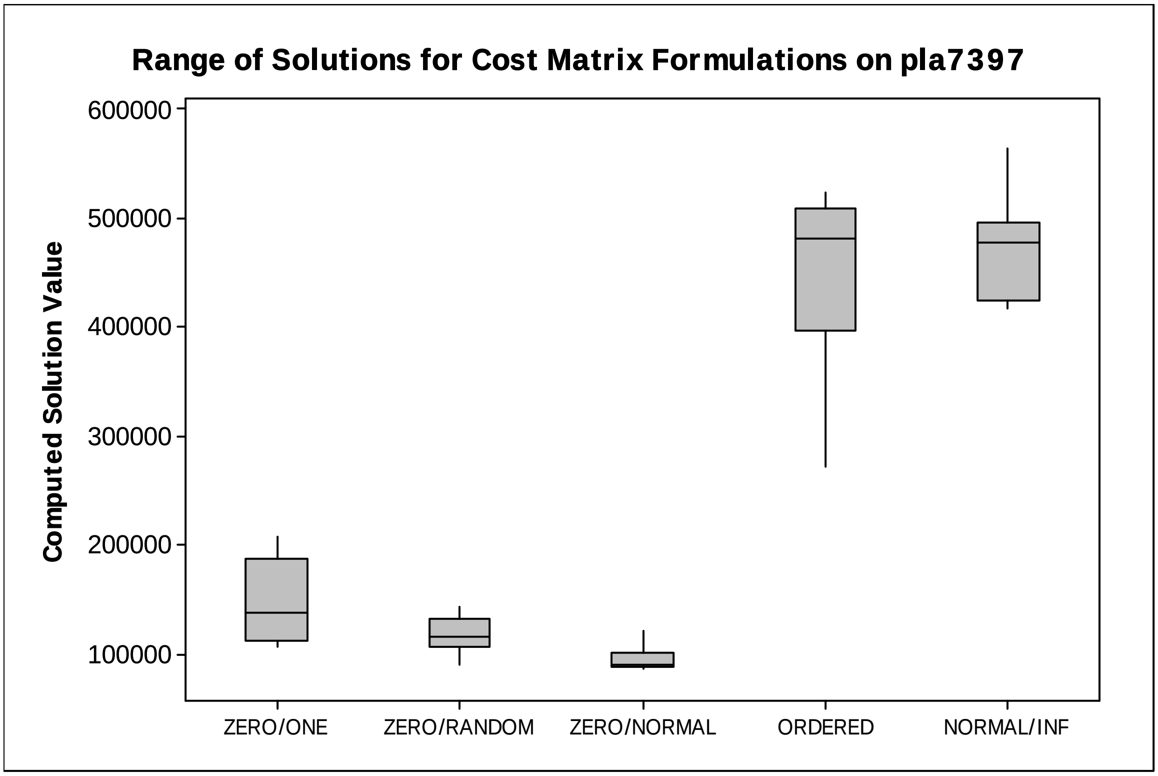


Figure 5.6a: Box plot of solutions for all five cost matrix formulations on pla7397

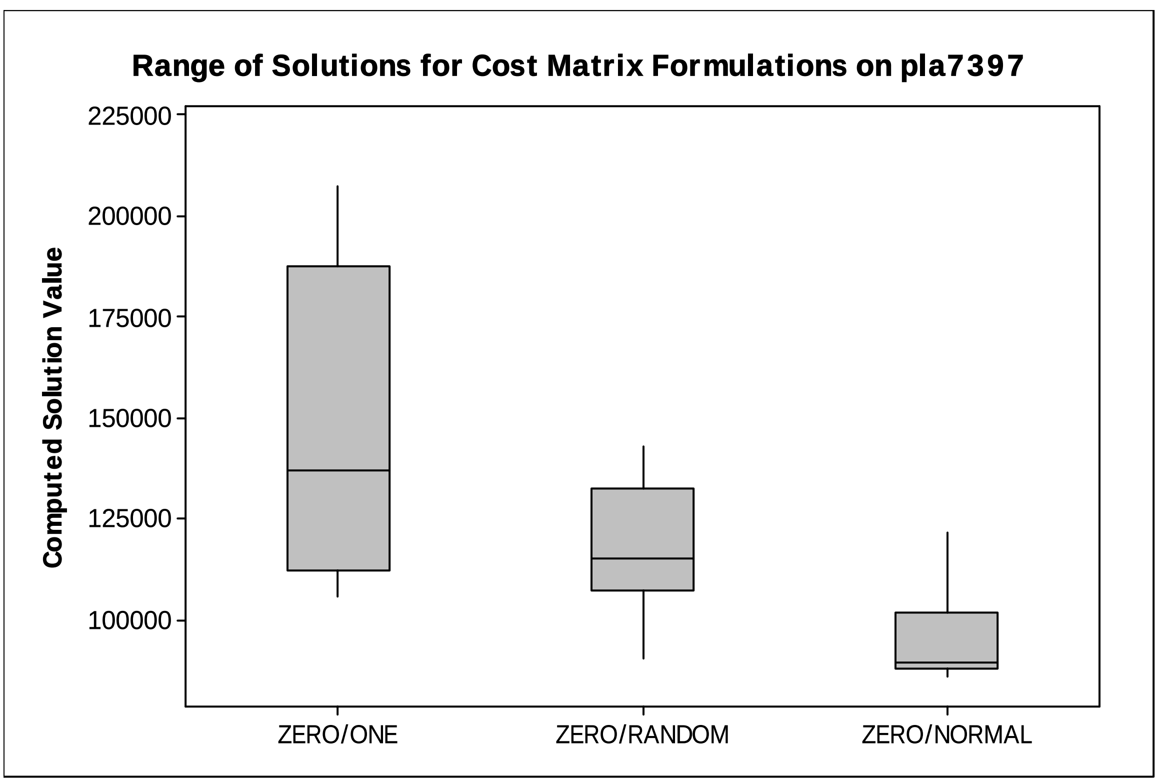


Figure 5.6b: Box plot of solutions for three best cost matrix formulations on pla7397

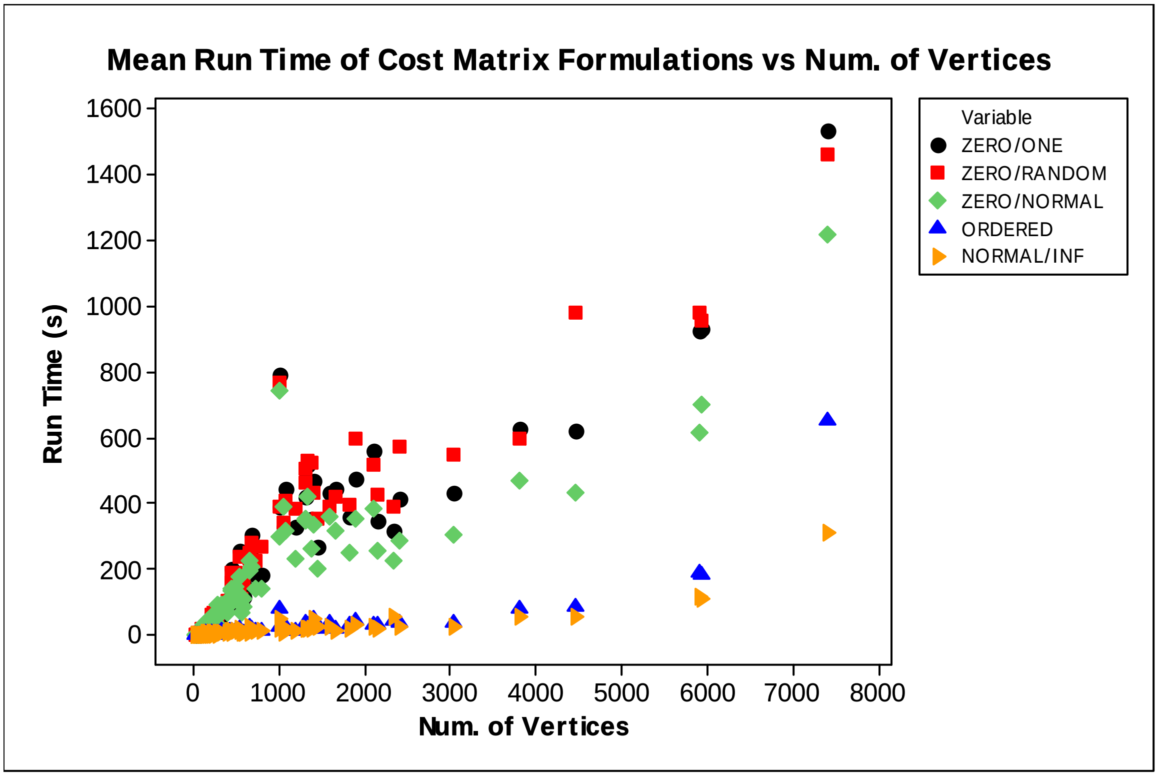


Figure 5.7: Mean run times of cost matrix formulations (all problems)

Figure 5.5 illustrates the accuracy of each of the five cost matrix formulations. Figure 5.6 shows box plots of the solutions from each trial for each formulation calculated for the pla7397 problem, which was the largest problem in our problem set. It's quite clear that the Zero/Normal cost matrix formulation is giving the best results. For problems less than 1000 vertices the Zero/One and Zero/Random formulations appear to be giving pretty good results as well. Overall, the Normal/Infinity and Ordered Position formulations appear to be giving very poor results.

Figure 5.7 shows that the running time of the Zero/Normal formulation is quite good as well when compared to the Zero/One and Zero/Random formulations. It seems the other two formulations run much more quickly, but to the consequence of poor answers. This more or less agrees with what Helsgaun observed [9]: problems that obey the triangle inequality are harder to solve (in the sense that they require more calculations) but their results are more accurate.

4.3 "Cost Shaking" Results

In addition to evaluating the performance of each of the five cost matrix formulations, we wanted to know how well this idea of "shaking" the cost matrix could improve our solutions. We made a single attempt to find a Hamiltonian cycle using the Zero/One formulation. If a Hamiltonian cycle could not be found, we made a number of additional attempts with the Zero/Random formulation. Three experiments consisting of no shake attempts, five shake attempts, and ten shake attempts were performed on problems from our TSPLIB problem set. Additionally, an experiment was performed where we made five repeated attempts with the Zero/One formulation.

The percent error difference from the optimal solution versus the number of vertices in the graph is summarized in figure 5.8. Figure 5.11 summarizes the range of values for the pr2392 problem from TSPLIB. This result is typical for the larger problems. As we increase the number of "shake" attempts the range of our solutions tightens. In addition, making repeated attempts with a Zero/Random cost matrix formulation appears to produce better results than additional attempts with the same Zero/One formulation. As we speculated earlier, the LK heuristic can sometimes get "stuck" at a certain place when trying to find a good TSP tour. Introducing some randomness into the graph can sometimes help the LK heuristic find a better tour.

The remaining figures summarize the mean run time for each LK heuristic call (figure 5.9), and the mean total run time for each binary search trial (figure 5.10). The shape of the points in figure 5.9 unsurprisingly resembles the shape of the LK heuristic of figure 5.2. Figure 5.10 shows that for problems of less than 1000 vertices that the difference in run times will not be that significantly larger as we attempt more and more shakes. The reason for this probably because the LK heuristic can easily find Hamiltonian cycles for problems less than 1000 vertices, so all four experiments will probably make a correct guess the first attempt. The difference comes when the LK heuristic is making attempts at finding a Hamiltonian cycle that is not there, because it will still make the five or ten attempts.

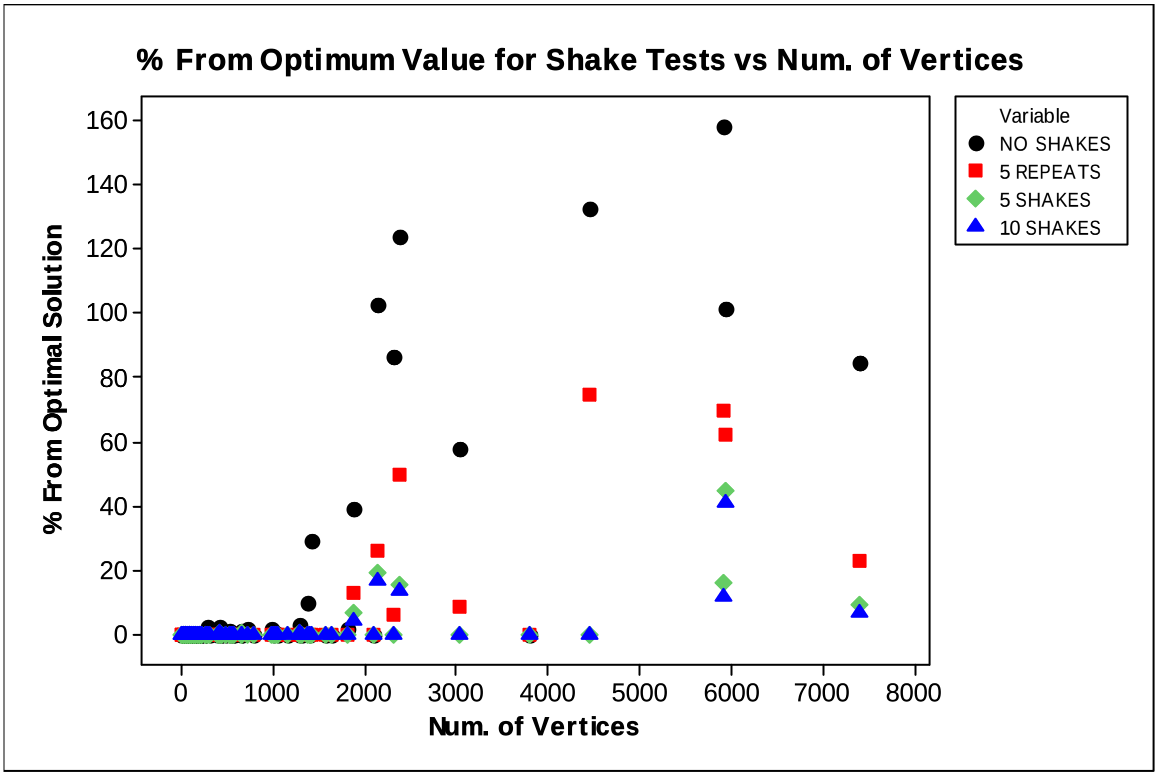


Figure 5.8a: Accuracy from performing shaking (all problems)

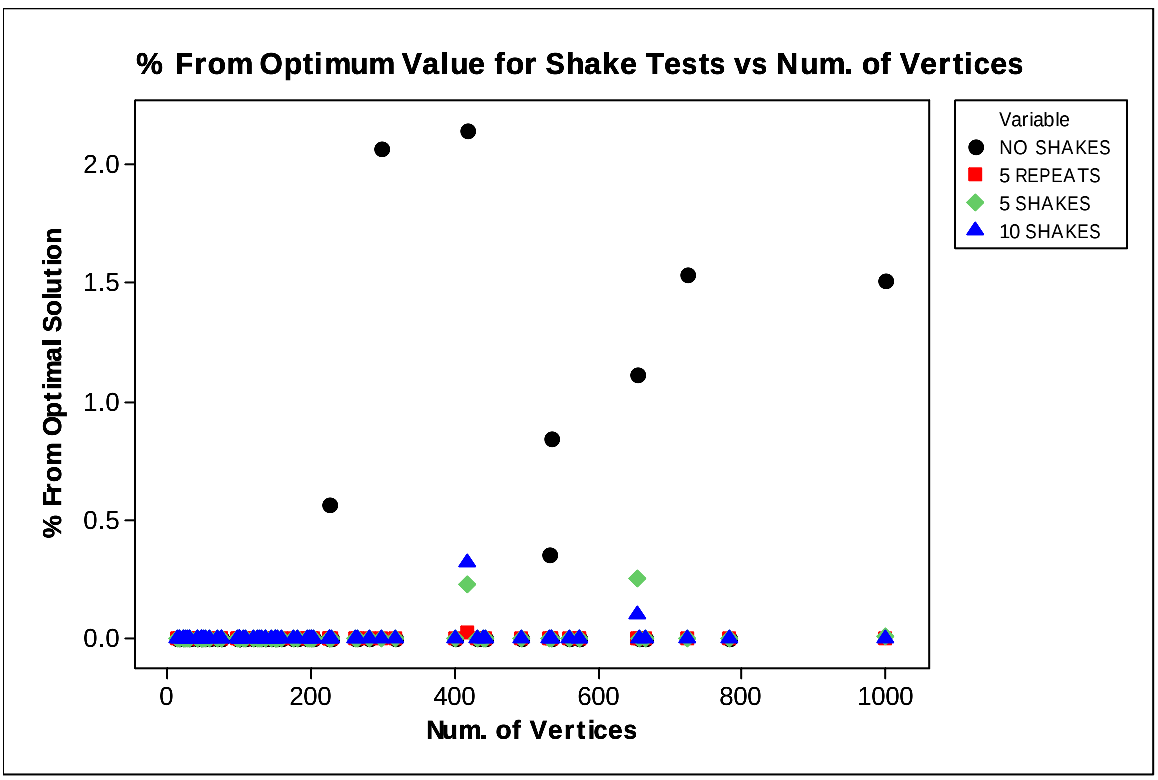


Figure 5.8b: Accuracy from performing shaking (problems of 1000 vertices or less)

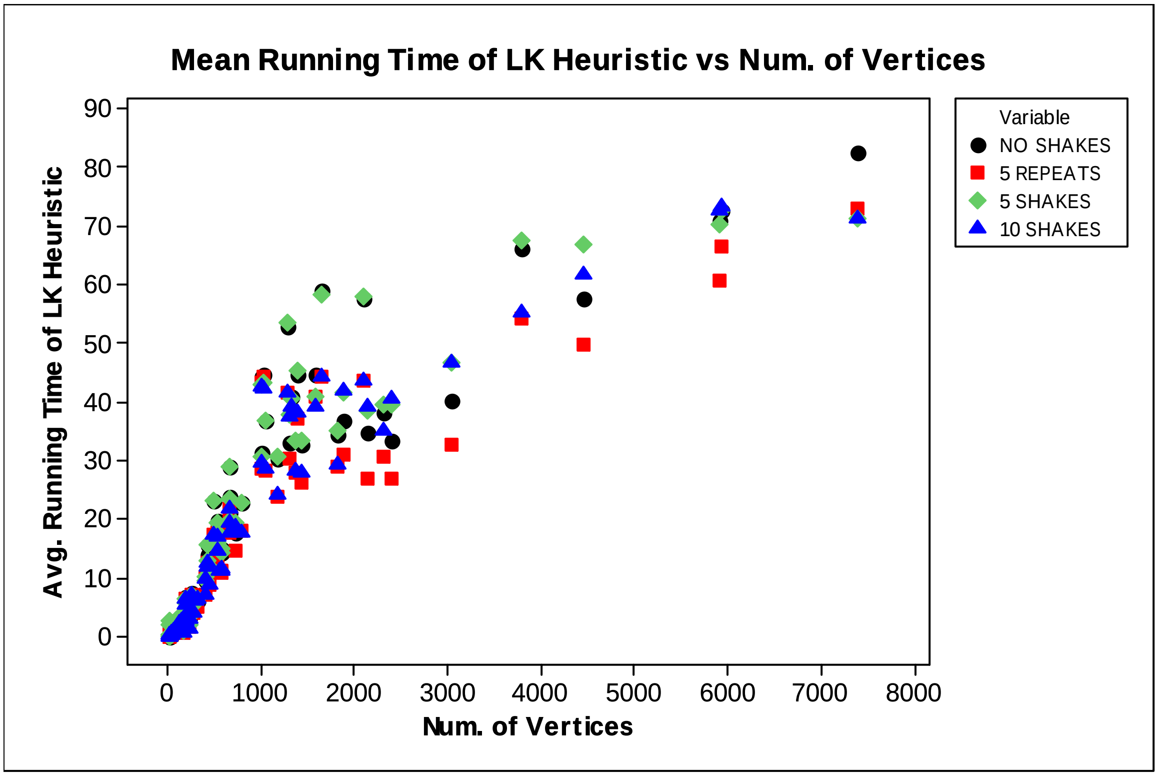


Figure 5.9: Mean running times of LK heuristic

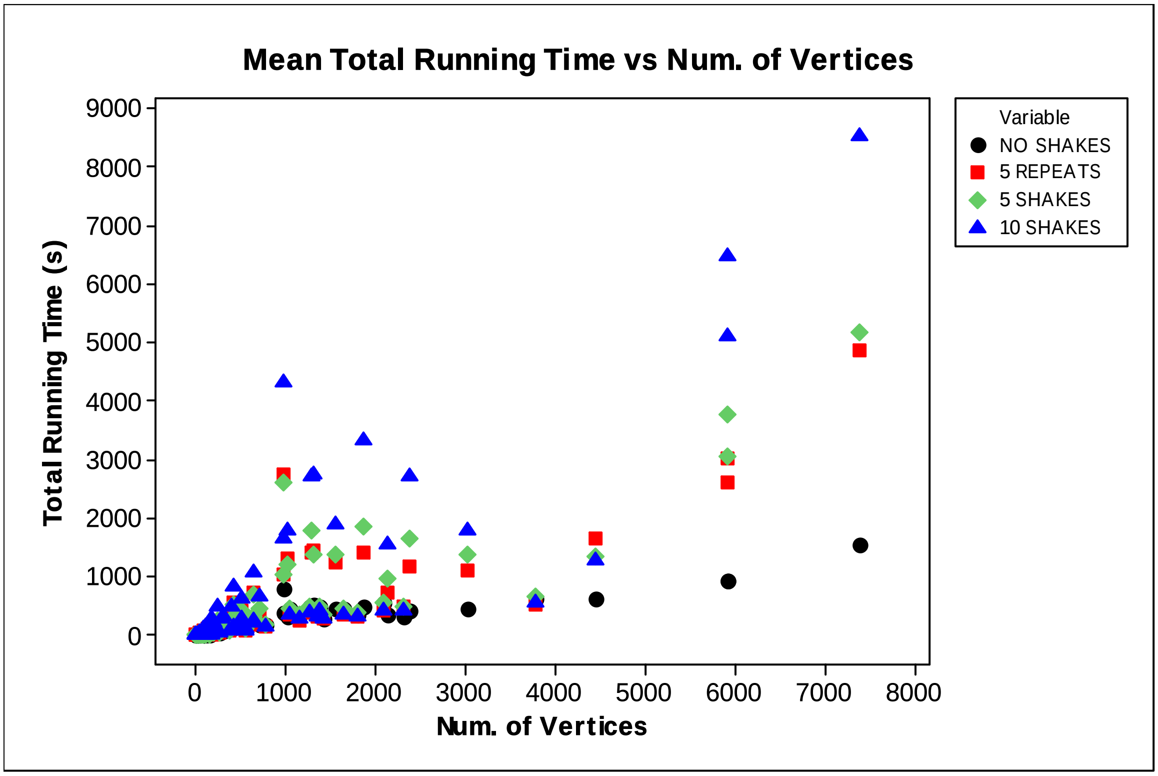


Figure 5.10: Mean total running time of BTSP binary search heuristic

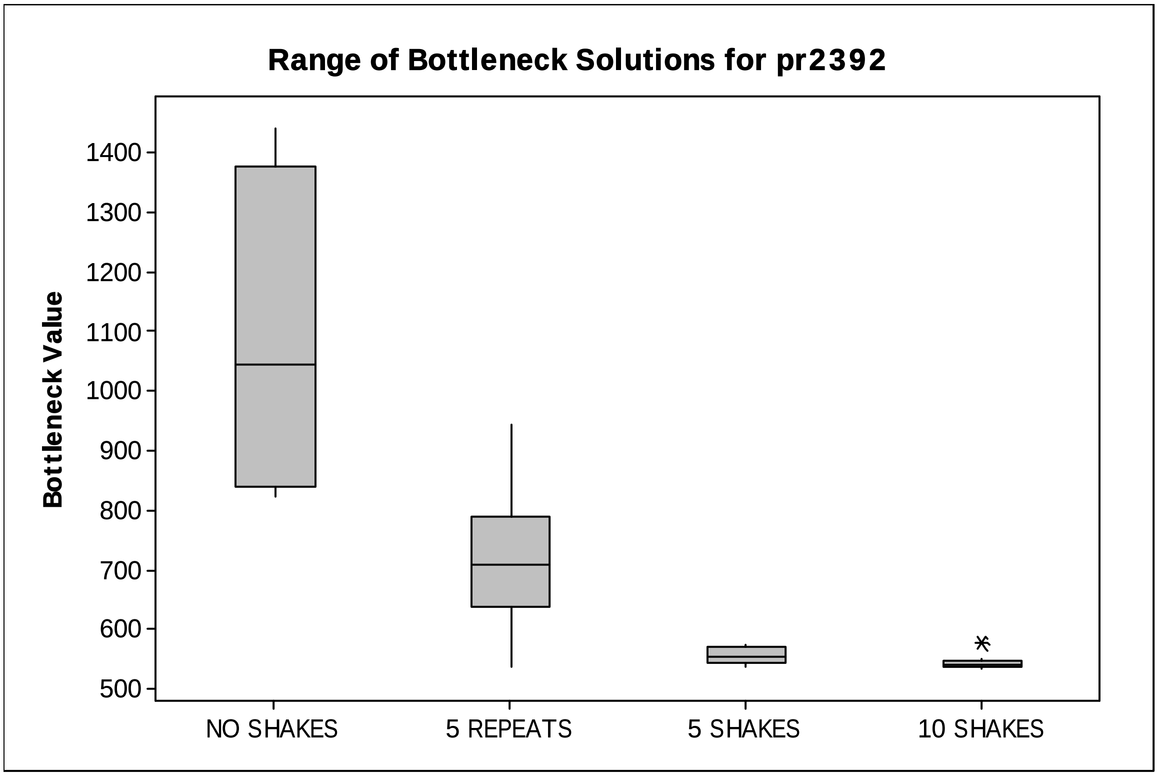


Figure 5.11: A box plot of the solutions found for pr2392 for each experiment

Based on this data it might be best to determine the number of "shake" attempts with the Zero/Random formulation based upon the size of the graph. As figure 5.8b shows, we tend to get just as good solutions for problems of less than 1000 vertices using five shake attempts as we do making ten shake attempts. For problems of less than 100 vertices it is probably sufficient enough to make a single call without any shaking. Implementing such a method could allow us to maintain the quality of the solutions while minimizing the total run time.

4.3 Results on TSPLIB Problems

Although our testing was limited to only problems less than 10,000 vertices, we attempted all the problems we could solve. It appears the BBSSP lower bound heuristic is the solution for almost every problem. This allowed us to find the solutions to the vast majority of the problems quite easily.

Five problems remain unsolved: linhp318, brd14051, pla33810, pla85900, rl11849. The first is because, as already noted, Concorde does not support TSPLIB problems with fixed length edges. It appears a memory limitation as a result of our implementation is preventing us from solving the other four large problems. Hopefully some minor coding changes will allow us to attempt these larger problems.

The BTSP solutions to the remaining problems can be found in Appendix B.

6 Conclusions and Further Study

The BBSSP lower bound and related LK tour upper bound appear to be excellent for solving BTSP on many problems. We have started study into problems where the BBSSP lower bound is weak. Another interesting study might be into the probability the BBSSP lower bound is the BTSP solution to a random graph.

Of the five cost matrix formulations we tested, the Zero/Normal one appears to work the best. However, there are other possible formulations we could test. It is possibly a different formulation could gives us even better results.

The idea of "shaking" the cost matrix to help the LK heuristic find Hamiltonian cycles appears to be a worthwhile idea. It certainly helps improve the results of our BTSP binary search heuristic. To strike a balance between accuracy and total run time it might be prudent to decide the number of shake attempts based upon the size of the graph.

Two other areas of further study include solving BTSP on directed graphs, and solving the related Maximum Scatter Traveling Salesman Problem. Most of the theory and techniques described in this paper can be applied to both problems.

Appendix A: Sample Results

Note: All times are in seconds.

Bounds Results for pla7379

Trial #	2MB	Time	BCSSP	Time	NN	Time	NI	Time	LK	Time
1	69772	0.38	81438	48.83	453983	0.59	476539	15.39	136015	66.64
2	69772	0.37	81438	48.68	453983	0.60	502499	15.09	81438	53.45
3	69772	0.37	81438	48.23	453983	0.59	481471	15.42	136015	66.60
4	69772	0.38	81438	48.36	453983	0.59	459952	15.17	81438	65.31
5	69772	0.38	81438	48.27	453983	0.60	572071	7.67	81438	64.70
6	69772	0.37	81438	48.30	453983	0.60	548436	23.05	136015	66.58
7	69772	0.37	81438	48.28	453983	0.60	516047	22.54	136015	66.62
8	69772	0.37	81438	48.31	453983	0.60	522750	22.95	136015	66.28
9	69772	0.37	81438	48.24	453983	0.59	504970	29.75	81438	64.19
10	69772	0.38	81438	48.36	453983	0.60	461946	7.58	136015	66.84
Mean	69772	0.37	81438	48.39	453983	0.59	504668	17.46	114184	64.72
StdDev	0	0.00	0	0.19	0	0.00	34747.7	6.72	26737.2	3.86

Cost Matrix Formulation result (Zero/Normal fomulation) for pla7397:

Trial	Optimal Solution	Lower Bound	Upper Bound	Solution	%-Opt	#LK Calls	Mean Lk Time	Total Time
1	81438	69772	474232	102173	25.46%	18	66.8	1217.6
2	81438	69772	564586	88814	9.06%	19	62.6	1212.6
3	81438	69772	524107	102058	25.32%	19	66.4	1285.1
4	81438	69772	555177	90359	10.95%	18	64.2	1187.2
5	81438	69772	509099	88752	8.98%	18	65.2	1204
6	81438	69772	548605	87914	7.95%	18	64.9	1191.9
7	81438	69772	507583	121665	49.40%	18	68.2	1250.9
8	81438	69772	492688	94493	16.03%	18	65.6	1210.7
9	81438	69772	481728	88384	8.53%	18	64.3	1188.2
10	81438	69772	549033	85960	5.55%	18	65.4	1208.3
Mean	81438	69772	520684	95057.2	16.72%	18.2	65.36	1215.65
StdDev	0	0	32450.1	10990.2	13.50%	0.42	1.55	30.59

Binary Search Heuristic result (5 "shakes" with 0/random cost matrix) for pr2392:

Trial	Optimal Solution	Lower Bound	Upper Bound	Solution	%-Opt	#LK Calls	Mean Lk Time	Total Time
1	*481*	*401*	*6374*	*553*	*14.97%*	*46*	*39.60*	*1822.90*
2	*481*	*401*	*5691*	*553*	*14.97%*	*36*	*38.60*	*1390.70*
3	*481*	*401*	*7401*	*576*	*19.75%*	*49*	*40.10*	*1964.30*
4	*481*	*401*	*6751*	*543*	*12.89%*	*35*	*38.90*	*1360.70*
5	*481*	*401*	*8114*	*570*	*18.50%*	*38*	*39.10*	*1487.00*
6	*481*	*401*	*7517*	*538*	*11.85%*	*50*	*39.50*	*1974.70*
7	*481*	*401*	*5764*	*552*	*14.76%*	*45*	*39.50*	*1776.50*
8	*481*	*401*	*8055*	*560*	*16.42%*	*46*	*40.60*	*1866.60*
9	*481*	*401*	*7341*	*546*	*13.51%*	*37*	*38.60*	*1427.00*
10	*481*	*401*	*6337*	*571*	*18.71%*	*39*	*39.40*	*1537.20*
Mean	*481*	*401*	*6934.5*	*556.2*	*15.63%*	*42.1*	*39.39*	*1660.76*
StdDev	*0*	*0*	*881.294*	*12.75234*	*2.65%*	*5.67*	*0.63*	*244.00*

Appendix B: BTSP Solutions to TSP-LIB Problems

All the solutions given here are optimal. All these problems, with the exception of ts225, were proven optimal by finding a tour whose largest cost was equal to the lower bound given by the biconnected bottleneck spanning subgraph problem heuristic. The ts225 solution was proven optimal using Concorde's exact solver.

NAME	# CITIES	TYPE	SOLUTION
a280	280	EUC_2D	20
ali535	535	GEO	3889
att48	48	ATT	519
att532	532	ATT	229
bayg29	29	GEO	111
bays29	29	GEO	154
berlin52	52	EUC_2D	475
bier127	127	EUC_2D	7486
brazil58	58	MATRIX	2149
brg180	180	MATRIX	30
burma14	14	GEO	418
ch130	130	EUC_2D	142
ch150	150	EUC_2D	93
d198	198	EUC_2D	1380
d493	493	EUC_2D	2008
d657	657	EUC_2D	1368
d1291	1291	EUC_2D	1289
d1655	1655	EUC_2D	1476
d2103	2103	EUC_2D	1133
d15112	15112	EUC_2D	1370
d18512	18512	EUC_2D	476
dantzig42	42	MATRIX	35
dsj1000	1000	CEIL_2D	295939
eil51	51	EUC_2D	13
eil76	76	EUC_2D	16
eil101	101	EUC_2D	13
fl417	417	EUC_2D	472
fl1400	1400	EUC_2D	530
fl1577	1577	EUC_2D	431
fl3795	3795	EUC_2D	528
fnl4461	4461	EUC_2D	132
fri26	26	MATRIX	93
gil262	262	EUC_2D	23
gr17	17	GEO	282
gr21	21	GEO	355
gr24	24	GEO	108
gr48	48	GEO	227
gr96	96	GEO	2807
gr120	120	MATRIX	220

NAME	# CITIES	TYPE	SOLUTION
gr137	137	GEO	2132
gr202	202	GEO	2230
gr229	229	GEO	4027
gr431	431	GEO	4027
gr666	666	GEO	4264
hk48	48	MATRIX	534
kroA100	100	EUC_2D	475
kroB100	100	EUC_2D	530
kroC100	100	EUC_2D	498
kroD100	100	EUC_2D	491
kroE100	100	EUC_2D	490
kroA150	150	EUC_2D	392
kroB150	150	EUC_2D	436
kroA200	200	EUC_2D	408
kroB200	200	EUC_2D	344
lin105	105	EUC_2D	487
lin318	318	EUC_2D	487
nrw1379	1379	EUC_2D	105
p654	654	EUC_2D	1223
pa561	561	MATRIX	16
pcb442	442	EUC_2D	500
pcb1173	1173	EUC_2D	243
pcb3038	3038	EUC_2D	198
pla7397	7397	CEIL_2D	81438
pr76	76	EUC_2D	3946
pr107	107	EUC_2D	7050
pr124	124	EUC_2D	3302
pr136	136	EUC_2D	2976
pr144	144	EUC_2D	2570
pr152	152	EUC_2D	5553
pr226	226	EUC_2D	3250
pr264	264	EUC_2D	4701
pr299	299	EUC_2D	498
pr439	439	EUC_2D	2384
pr1002	1002	EUC_2D	2129
pr2392	2392	EUC_2D	481
rat99	99	EUC_2D	20
rat195	195	EUC_2D	21
rat575	575	EUC_2D	23
rat783	783	EUC_2D	26
rd100	100	EUC_2D	221
rd400	400	EUC_2D	104
rl1304	1304	EUC_2D	1535
rl1323	1323	EUC_2D	2489
rl1889	1889	EUC_2D	896
rl5915	5915	EUC_2D	602
rl5934	5934	EUC_2D	896
rl11849	11849	EUC_2D	842
si175	175	MATRIX	177

NAME	# CITIES	TYPE	SOLUTION
si535	535	MATRIX	227
si1032	1032	MATRIX	362
st70	70	MATRIX	24
swiss42	42	MATRIX	67
ts225	225	EUC_2D	1000
tsp225	225	EUC_2D	36
u159	159	EUC_2D	800
u574	574	EUC_2D	345
u724	724	EUC_2D	170
u1060	1060	EUC_2D	2378
u1432	1432	EUC_2D	300
u1817	1817	EUC_2D	234
u2152	2152	EUC_2D	105
u2319	2319	EUC_2D	224
ulysses16	16	GEO	1504
ulysses22	22	GEO	1504
usa13509	13509	EUC_2D	16754
vm1084	1084	EUC_2D	998
vm1784	1784	EUC_2D	1017

Unsolved Problems: linhp318, brd14051, pla33810, pla85900, rl11849

Appendix C: BTSP Solutions to Standard Random Problems

All the solutions given here are optimal, as proven by the existence of a tour (found by the LK tour upper bound heuristic) whose largest cost is equal to the lower bound (found by the BBSSP lower bound heuristic)

Key:
E = Uniform Point
C = Clustered point
M = Random distance matrix

Note: The solution to C10k.1 is just a lower and upper bound, not the optimal solution.

For more information about these problems, please visit Johnson and McGeoch's web site [10].

NAME	TYPE	SEED	NODES	SOLUTION
C1k.0	C	1000	1000	290552
C1k.1	C	10001	1000	335184
C1k.2	C	10002	1000	225295
C1k.3	C	10003	1000	416768
C1k.4	C	10004	1000	318930
C1k.5	C	10005	1000	260389
C1k.6	C	10006	1000	175740
C1k.7	C	10007	1000	301366
C1k.8	C	10008	1000	246519
C1k.9	C	10009	1000	208091
C3k.0	C	3162	3162	252245
C3k.1	C	31621	3162	167466
C3k.2	C	31622	3162	194007
C3k.3	C	31623	3162	180852
C3k.4	C	31624	3162	180583
C10k.0	C	10000	10000	161062
C10k.1	C	100001	10000	[94139, 106864]
C10k.2	C	100002	10000	121209
E1k.0	E	1000	1000	64739
E1k.1	E	10001	1000	67476
E1k.2	E	10002	1000	88522
E1k.3	E	10003	1000	59220
E1k.4	E	10004	1000	68259
E1k.5	E	10005	1000	61406
E1k.6	E	10006	1000	68777
E1k.7	E	10007	1000	70389
E1k.8	E	10008	1000	57597
E1k.9	E	10009	1000	68420
E3k.0	E	3162	3162	39854
E3k.1	E	31621	3162	37500
E3k.2	E	31622	3162	35145
E3k.3	E	31623	3162	44428
E3k.4	E	31624	3162	36621
E10k.0	E	10000	10000	20174
E10k.1	E	100001	10000	22883
E10k.2	E	100002	10000	20208
M1k.0	M	1000	1000	9328
M1k.1	M	10001	1000	8856
M1k.2	M	10002	1000	11282
M1k.3	M	10003	1000	11617
M3k.0	M	3162	3162	3289
M3k.1	M	31621	3162	3034
M10k.0	M	10000	10000	1189

References

[1] D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun. "Optimal Tour of Sweden". Last Updated June, 2004. URL: http://www.tsp.gatech.edu/sweden/index.html

[2] D. Applegate, R. Bixby, V. Chvátal, W. Cook, and M. Mevenkamp. Corcode TSP Solver. Last Updated Jan. 2005. URL: http://www.tsp.gatech.edu/concorde.html

[3] D. Applegate, W .Cook, S. Dash, and M. Mevenkamp. QSopt Linear Programming Solver. Last Updated March 2004. URL: http://www.isye.gatech.edu/~wcook/qsopt/

[4] G. Carpaneto, S. Martello, and P. Toth. An algorithm for the bottleneck traveling salesman problem. Oper. Res., 32: 380— 389, 1984.

[5] W. Cook. The First Big TSP. Last Updated Jan. 2005. URL: http://www.tsp.gatech.edu/history/pictorial/dfj.html

[6] W. Cook. World Traveling Salesman Problem. Last Updated Sept. 13, 2004. URL: http://www.tsp.gatech.edu/world/

[7] R. S. Garfinkel and K. C. Gilbert. The bottleneck traveling salesman problem: Algorithms and probabilistic analysis. J. Assoc. Comput. Mach., 25: 435448, 1978.

[8] P. C. Gilmore and R. E. Gomory. Sequencing a one state-variable machine: A solvable case of the traveling salesman problem. Oper. Res., 12: 655— 679, 1964.

[9] K. Helsgaun, "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic", DATALOGISKE SKRIFTER (Writings on Computer Science), No. 81, 1998, Roskilde University. URL: http://www.akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf

[10] D. S. Johnson and L. A. McGeoch. Benchmark Code and Instance Generation Codes. Last Updated May 2002. URL: http://www.research.att.com/~dsj/chtsp/download.html

[11] S. Kabadi and A. P. Punnen. The Bottleneck TSP. In *The Travelling Salesman Problem and Its Variants*. Secaucus, NJ, USA: Kluwer Academic Publishers, 2002. Ch. 15, p 697.

[12] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. Oper. Res. 21:972-989, 1973

[13] G. S. Manku. A linear time algorithm for the bottleneck biconnected spanning subgraph problem. Inform. Process. Lett., 59: 1— 7, 1996.

[14] Dave Mount, Articulation points and Biconnected Components, Oct. 6 1998. URL: http://www.cs.umd.edu/~samir/451/bc.ps

[15] D. Naddef. Polyhedral Theory and Branch-and-Cut Algorithms for the Symmetric TSP. In *The Travelling Salesman Problem and Its Variants*. Secaucus, NJ, USA: Kluwer Academic Publishers, 2002. Ch. 2, p 29.

[16] R. G. Parker and R. L. Rardin. Guaranteed performance heuristics for the bottleneck traveling salesperson problem. Oper. Res. Lett., 12: 269— 272, 1982.

[17] A. P. Punnen. Computational Complexity. In *The Travelling Salesman Problem and Its Variants*. Secaucus, NJ, USA: Kluwer Academic Publishers, 2002. Appendix B, p 754.

[18] A. P. Punnen and K. P. K. Nair. A fast and simple algorithm for the bottleneck biconnected spanning subgraph problem. Inform. Process. Lett., 50: 283— 286, 1994.

[19] G. Reinelt. TSPLIB. Last Updated Nov. 2004.
URL: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

[20] E. A. Timofeev. Minmax di-associated subgraphs and the bottleneck traveling salesman problem. Cybernetics, 4: 75— 79, 1979.

[21] UNB Advanced Computational Research Laboratory. Last Updated Nov. 2004.
URL: http://acrl.cs.unb.ca