Comparison of Heuristics for the Colorful Traveling Salesman Problem

J. Silberholz*, A. Raiconi†, R. Cerulli†, M. Gentili‡, B. Golden§, S. Chen¶

Abstract

In the Colorful Traveling Salesman Problem (CTSP), given a graph G with a (not necessarily distinct) label (color) assigned to each edge, a Hamiltonian tour with the minimum number of different labels is sought. The problem is a variant of the well-known Hamiltonian Cycle problem and has potential applications in telecommunication networks, optical networks, and multimodal transportation networks, in which one aims to ensure connectivity or other properties by means of a limited number of connection types. We propose two new heuristics based on the deconstruction of a Hamiltonian tour into subpaths and their reconstruction into a new tour, as well as an adaptation of an existing approach. Extensive experimentation shows the effectiveness of the proposed approaches.

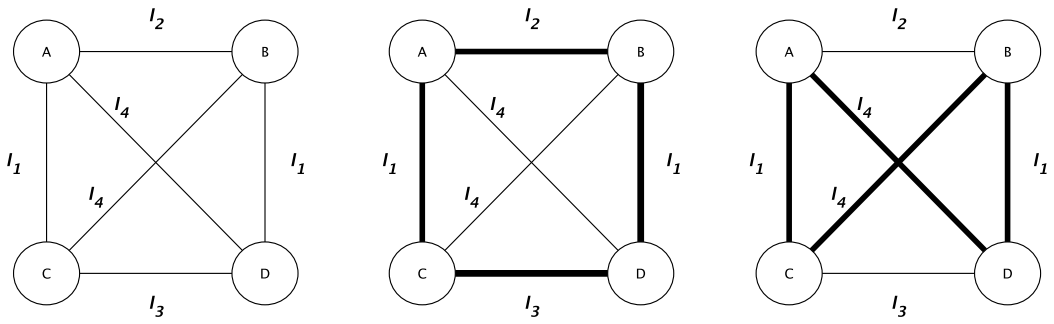
Keywords: Genetic Algorithms, Hamiltonian Tour, Metaheuristics

1 Introduction

Recently, a variant of the Traveling Salesman Problem called the Colorful Traveling Salesman Problem (CTSP) has been studied. For this problem, given a graph G with a label (or color) assigned to each edge, a Hamiltonian tour of G with the minimum number of different labels is sought. A CTSP instance with possible solutions is shown in Figure 1 (edges that are part of the solution in Subfigures 1(b) and 1(c) are highlighted in bold).

Looking for a Hamiltonian tour on a general graph is an $\mathcal{NP}-complete$ problem [13], so looking for a

*Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA, 02139. josilber@mit.edu
†Dipartimento di Matematica, Universitá di Salerno, P.te Don Melillo, 84084 Fisciano (SA), Italia. {araiconi@unisa.it, raffaele@unisa.it}
‡Dipartimento di Informatica, Universitá di Salerno, P.te Don Melillo, 84084 Fisciano (SA), Italia. mgentili@unisa.it
§R.H. Smith School of Business, University of Maryland, College Park, MD 20742. bgolden@rhsmith.umd.edu
¶College of Business, Murray State University, Murray, KY, 42071. si.chen@murraystate.edu



(a) (b) (c)

Figure 1: (a) Example graph G with four nodes, six edges and four labels ($\{l_1, l_2, l_3, l_4\}$). (b) Feasible CTSP solution using three different labels ($\{l_1, l_2, l_3\}$). (c) Optimal CTSP solution using two labels ($\{l_1, l_4\}$).

Hamiltonian tour with the minimum number of labels on general graphs is a difficult problem, too. The CTSP was proven to be $\mathcal{NP}-$hard in [6, 25].

Several related problems with labeled graphs have been studied in recent years. One such problem is the Minimum Label Spanning Tree (MLST) Problem, in which a spanning tree with the minimum number of labels is sought over a labeled graph. The problem was introduced in [2, 9]. Heuristics for the problem have been discussed in [7, 18, 21, 22, 23, 24]. Other problems involving classic combinatorial optimization problems defined on labeled graphs include the Minimum Label Steiner Tree Problem discussed in [8], the Minimum Label Generalized Forest Problem discussed in [4], the Minimum Label Path Problem studied in [4, 15] and the Labeled Maximum Matching Problem presented in [5].

As mentioned in [25], the CTSP has applications in transportation modeling. Consider an instance of the Traveling Salesman Problem in which there are several transportation providers, each of which charges the same fixed rate for any services. This can be modeled as the CTSP by assigning each provider a unique label and assigning a provider's label to each edge of the graph it services.

A further application may be found in planning a convoy trip. Consider a convoy that must service a set of locations (the nodes on the graph) and return home; assign a label to edges between nodes representing the major threat to the convoy along the path between the two locations. A planner would seek to minimize the number of distinct threats that the convoy needs to be hardened against, which corresponds to solving the CTSP.

A final application for the CTSP can be found in solving a variation of the classical machine scheduling problem with sequential tasks. In the classical version of this problem, an arc weight is assigned to arc

(i, j) to signify the amount of work needed to transition the machine from performing task i to performing task j. The TSP solution yields the least amount of work needed to perform all the tasks. Instead, now consider the machine scheduling problem where the label on arc (i, j) indicates the specialization needed for a worker to transition the machine from performing task i to performing task j. The CTSP finds the minimum number of specialized workers needed to perform all the tasks.

Additional research has been completed on the Labeled Max Traveling Salesman Problem, which is a variant of the CTSP in which the objective is to maximize the number of different labels used. The existence of perfect solutions to this problem (ones using n labels for an n-node Hamiltonian tour) on certain graphs was proven in [1]. Further, a 1.5-approximation algorithm for the problem is provided in [12]. Algorithms for the Labeled Minimum Path Problem, a variant of the CTSP in which the objective is to find a path between two nodes instead of a Hamiltonian tour, are presented in [3, 14].

A branch-and-cut algorithm for the CTSP and two variants was proposed in [16]. Meanwhile, a number of heuristics have been proposed for the CTSP. In [6], a heuristic called ColorHAM was developed, along with a tabu search heuristic based on that procedure. In [25], a heuristic called the MPEA was developed and incorporated into a genetic algorithm. Each of these two heuristics is benchmarked on specific test cases. In [12], a greedy algorithm called Greedy Tour is proposed for the problem, but no computational results are provided. Here we extend the CTSP heuristic proposed in [25], present two new heuristics for the problem, and show one of them to be the most effective heuristic to date.

Our paper is organized as follows: In Section 2, we introduce some notation and definitions that will be used throughout the paper. In Section 3, we discuss modifications to generalize a heuristic for the CTSP originally proposed in [25]. In Section 4, we propose two new heuristics based on the deconstruction of a Hamiltonian tour into subpaths and the reconstruction of these subpaths into a new Hamiltonian tour. Finally, we present computational results, data analysis, and our conclusions in Sections 5 and 6.

2 Notation and definitions

Given an undirected graph $G = (V, E, L)$, where V is the set of nodes, E is the set of edges, and L is the set of labels in the graph, let $c_e \in L$ be the label associated with edge $e \in E$. Let $n = |V|$ and $m = |L|$.

Given a path P of G, let $V(P) \subseteq V$ be the set of its nodes, $E(P) \subseteq E$ be the set of its edges and $V_e(P) \subseteq V(P)$ be the set of its endpoints. We define $C(P) = \{c_e \mid e \in E(P)\}$ as the set of labels assigned to the edges of the path. Given a subset of labels $L' \subseteq L$, let $E(L') \subseteq E$ be the edges with labels

belonging to L' (that is, $E(L') = \{e \in E \mid c_e \in L'\}$). For example, given the graph in Figure 1(a), let $P = (a, d, b, c)$, then $E(P) = \{(a, d), (d, b), (b, c)\}$, $V(P) = \{a, d, b, c\}$, $V_e(P) = \{a, c\}$ and $C(P) = \{l_4, l_1\}$. Moreover, for $L' = \{l_2, l_3\}$, it follows that $E(L') = \{(a, b), (c, d)\}$.

The CTSP searches for a Hamiltonian tour H of G such that the cardinality of its set of labels $|C(H)|$ is minimized.

3 Heuristic Extending MLST Solutions

An interesting method for approaching the CTSP, which uses solutions for the Minimum Label Spanning Tree (MLST) Problem as a starting point, was proposed in [25]. The approach uses a heuristic procedure named the Maximum Path Extension Algorithm (MPEA) to extend the base label set derived from the MLST solution to obtain a feasible CTSP solution. As shown by the authors, this method produces better results than an approach that does not consider the solution of the MLST problem. In Section 3.1 we describe this approach as presented in [25]. However, MPEA is limited in that it only considers complete graphs. In the applications of the CTSP, graphs with density less than one (i.e., that are not complete) may occur, demonstrating the need to extend the scope of the algorithm. We made several modifications to the MPEA to ensure that all densities are accepted, as described in Section 3.2. We called this algorithm MPEA-mod. Moreover, we also changed the initialization method with respect to the one presented in [25], and developed an iterated version of MPEA-mod. The resulting algorithm is presented in Section 3.3 and tested in Section 5.

3.1 Maximum Path Extension Algorithm (MPEA)

The MPEA heuristic for complete graphs presented in [25] operates by maintaining a path P and a set C of acceptable labels. The set C is initialized using the genetic algorithm for MLST presented in [22], and the path P is initialized with a random edge with a label belonging to C. In Section 4.4, we present a genetic algorithm for the CTSP based on the same structure of the genetic algorithm for the MLST used to initialize the set C.

Using a set of rules (rules 1–4 described below), nodes are iteratively added to P using only edges with labels from C, if possible. More specifically, each extension step attempts, in sequence, the following operations and performs one of them if it is possible to do so using only edges with acceptable labels.

1. Add the edge (e, c) from an endpoint e of P to a node $c \in V \setminus V(P)$.

2. Insert a new node c between two adjacent nodes of the path a and b, substituting (a, b) with (a, c) and (c, b).

3. Perform a rotational transformation of P and add edge (b, c) from the new endpoint b to a node $c \in V \setminus V(P)$. A rotational transformation is a move that has the effect of changing one of the endpoints of P without changing its nodes or adding unwanted labels to C. A rotational transformation T of path P, as defined in [19], can be generated by selecting adjacent nodes a and b from P and considering endpoint e of P such that b is between a and e in P. If an edge exists between a and e and that edge is labeled with an element of C, then the edge between a and b is removed and the edge between a and e is added to complete the rotational transformation.

4. If an edge (e_1, e_2) exists between the endpoints of P and it is labeled with an element of C, then add that edge and remove some edge (a, b), making a and b the new endpoints of P. Add the edge between one of the new endpoints and a node $c \in V \setminus V(P)$, such that this new edge is labeled with an element of C.

If none of these attempts is successful, a new node is appended to one end of P such that the label added has the highest frequency in the graph. Then, C is updated to include this new label.

Finally, after a Hamiltonian path is formed, the edge between the endpoints of P is inserted and the final Hamiltonian tour is returned.

3.2 MPEA-mod

When densities less than 1 are considered, several of the steps described in 3.1 become more difficult to perform. As the number of nodes in P approaches n, there often exist no nodes that can be appended to P using an edge. Hence, to reduce the likelihood that the MPEA fails after it cannot insert any node with an edge labeled with an element of C, we have added more ways for the MPEA to insert this next node.

First, every time that an extension from the endpoints is attempted, the new method considers both endpoints of P as candidates, with the label with the highest frequency added if a new label has to be introduced.

Moreover, rotational transformations of P are considered in a wider number of cases if an extension from P is not possible. If none of the extension steps is successful, we consider inserting a new node

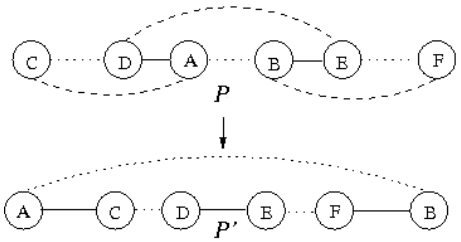


Figure 2: *Example of direct endpoint matching.*

between two existing nodes in P or any of its rotational transformations, with the insertion that would add the fewest new labels to C being the one accepted.

Another step of the MPEA that becomes more complicated when graphs with densities less than 1 are considered is the completion of the Hamiltonian tour. When a Hamiltonian path P has been discovered, it is often the case that no edge connects the endpoints of P on a sparse graph. In this case, we first consider again any rotational transformation of P, accepting the one that introduces the fewest new labels while completing a Hamiltonian tour. If this operation is unsuccessful, we then use a process we call *direct endpoint matching*. In this process, we search for consecutive internal nodes of P that can be inserted between the endpoints of P, resulting in a Hamiltonian path with a valid edge between its new endpoints. More formally (refer to Figure 2), if c and f are the endpoints of P and edges (d, a) and (b, e) are selected from P, with a and b between d and e in P, such that edges exist between nodes c and a, b and f, and d and e, then edges (d, a) and (b, e) are removed and edges (c, a), (b, f), and (d, e) are added, resulting in a Hamiltonian tour. Among all the Hamiltonian tours that could be generated in this way, we select the one with the fewest labels.

A pseudocode for MPEA-mod, which combines the initial MPEA described in Section 3.1 with the extensions described in Section 3.2 is provided in Algorithm 6 in the appendix. The runtime of MPEA-mod is $O(n^3)$, with the most intensive process coming in the part that extends the current path into a Hamiltonian path. Several of the steps to extend the path operate in $O(n^2)$ time, and these steps must be repeated $O(n)$ times to complete a Hamiltonian path. The code to complete the Hamiltonian tour once a Hamiltonian path has been obtained operates in $O(n^2)$ time, with the direct endpoint matching dominating the computation time. The overall runtime of direct endpoint matching is $O(n^2)$ because every pair of edges in P is considered for the matching and there are $O(n)$ edges in the Hamiltonian path.

3.3 Iterated MPEA-mod (IMPEA)

The computational runtime of the genetic algorithm for MLST used for the initialization of MPEA increases as the density of a dataset decreases, making it less suitable for MPEA-mod. Therefore, we chose to use the much quicker maximum node covering algorithm (MVCA) presented in [9] instead, and decided to run the whole procedure multiple times using a multistart approach, yielding a final algorithm that we call the iterated MPEA-mod (IMPEA).

MVCA is a greedy algorithm for solving the MLST problem. The algorithm maintains a set of labels C (initially C is empty) and iteratively selects and adds label l to C such that the subgraph induced by $C \bigcup \{l\}$ on graph G has a minimal number of components. In the version of the MVCA we used in this paper, ties are broken randomly, yielding variable results over multiple executions of the MVCA algorithm.

The IMPEA algorithm for the CTSP is very simple — the MVCA is run p times, with the resulting label set converted into a Hamiltonian tour each time by the MPEA. The Hamiltonian tour with the fewest number of labels after these p iterations is returned as the final solution. Pseudocode is provided in Algorithm 1.

The selection of parameter p for computational testing is described in Section 5.1.

Algorithm 1 Iterated MPEA-mod (IMPEA)

```
1: function IMPEA(p)
2: // Parameter p: number of iterations
3: best ← null
4: for iter = 1 to p do
5:    L ← mvca(), as presented in [9]
6:    H ← mpea-mod(L), as described in Section 3.2
7:    if H is a Hamiltonian tour and (best = null or |C(H)| < |C(best)|) then
8:       best ← H
9:    end if
10: end for
11: if best = null then
12:    Return in failure
13: else
14:    Return best, the final Hamiltonian tour
15: end if
16: end function
```

4 Repair-Based Heuristics

While existing heuristic approaches to the CTSP, such as those presented in [6, 25], have relied on completely generating Hamiltonian tours every time a new label set is considered, the heuristics described in this section use repair methods and consider new label sets in a different manner. That is, these heuristics try to reduce the number of labels used by a given Hamiltonian tour by removing all the edges corresponding to some of its labels (*deconstruction* phase), and reconnecting the resulting disconnected subpaths using only the remaining labels (*repair* phase).

First, we introduce the procedures that both these algorithms use for their deconstruction and repair phases (Sections 4.1 and 4.2, respectively). As will be shown, while the deconstruction phase is simple and straightforward, the repair phase is more complex and makes use of a number of different moves. Then, we present two repair-based heuristics, namely a greedy heuristic (described in Section 4.3) and a genetic algorithm (described in Section 4.4).

4.1 Hamiltonian Tour Deconstruction

The Hamiltonian tour deconstruction method takes as arguments a Hamiltonian tour H and a set of acceptable labels C'. All edges $e \in E(H)$ with a label $c_e \notin C'$ are removed, and a resulting set of subpaths P is generated.

For instance, if a Hamiltonian tour (a, b, c, d, e, f, a) were deconstructed with acceptable label set C' such that $c_{(a,b)}, c_{(c,d)}, c_{(d,e)}, c_{(f,a)} \in C'$ but $c_{(b,c)}, c_{(e,f)} \notin C'$, then subpaths (f, a, b) and (c, d, e) would be generated.

4.2 Hamiltonian Tour Repair

After Hamiltonian tour deconstruction, the set of disconnected subpaths P must be reconnected with edges from the new label set C' to create a new Hamiltonian tour that only uses labels from C'.

This is similar to the problem addressed in a number of works on the Hamiltonian cycle problem, such as [11] and later [17]. Each of these works assumes a set of subpaths (called segments in those works) and uses an exact algorithm to determine whether a Hamiltonian tour exists using those segments. However, these approaches maintain all the edges in each of the input segments. In practice, nearly all sets P generated using the method described in Section 4.1 cannot be reconnected using all the edges contained in each element of P, so the pruning techniques described in the literature for this problem are not useful

for our approach. This motivated the creation of new procedures that modify the stored subpaths in an attempt to connect them into a Hamiltonian tour.

We first provide a series of moves used to combine subpaths and to convert the final Hamiltonian path into a Hamiltonian tour. More specifically, we do the following:

1. In Sections 4.2.1 and 4.2.2, we present Direct Subpath Connection and Subpath Insertion, two moves that reduce the number of disconnected subpaths by joining two of them;

2. In Sections 4.2.3 and 4.2.4, we present Circular and Non-Circular Splicing, used to transform subpaths and increase their likelihood of being combined together later in the procedure;

3. In Sections 4.2.5, 4.2.6, and 4.2.7, we present Direct, Extended and Random Endpoint Matching, which are used to obtain a Hamiltonian tour once all the subpaths have been combined into a Hamiltonian path.

Finally, we present the full repair algorithm that makes use of these moves in Section 4.2.8.

4.2.1 Direct Subpath Connection

The most intuitive move to connect subpaths is direct subpath connection. Direct subpath connection connects two subpaths if an edge with a label from the set of acceptable labels C' exists between one endpoint from each of the subpaths. If so, then the edge is added. Clearly, since two subpaths are combined into one, direct subpath connection decreases the total number of subpaths $|P|$ by 1. This method is illustrated in Figure 3 with subpaths P_a and P_b.

However, direct subpath connection as described above is limited in the sense that only the endpoints are considered in the connections. Some subpaths can be directly adjoined by more than just their endpoints. If a subpath is circular, meaning an edge exists between its endpoints and the label on that edge is an element of C', then any rotation of that subpath may be considered for direct subpath connection. Hence, if a subpath $(1, 4, 7, 3)$ is determined to be circular, then that subpath along with $(4, 7, 3, 1)$, $(7, 3, 1, 4)$, and $(3, 1, 4, 7)$ are considered for direct subpath connection. In the extreme case when both P_a and P_b are circular, if the edge between any node in P_a and any node in P_b is labeled with a label in C', direct subpath connection can be applied to those two subpaths.

Last, rotational transformations as described in Section 3.2 are used to strengthen the direct subpath connection step. When searching for direct subpath connections, we also connect any paths that can be linked after a rotational transformation to one or both of the paths.

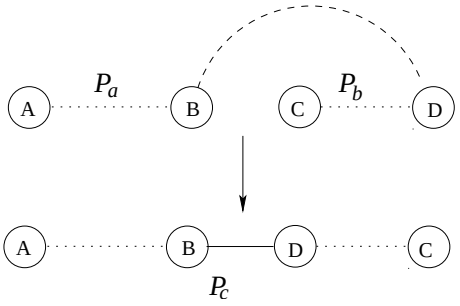


Figure 3: *Example of direct subpath connection.*

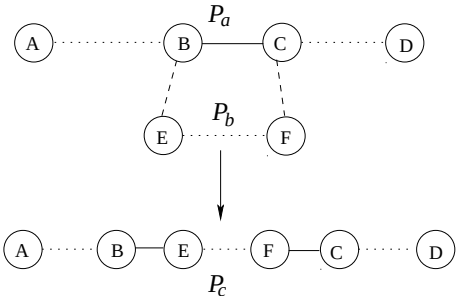


Figure 4: *Example of subpath insertion.*

The overall runtime of the direct subpath connection procedure is $O(n^2)$, because each node in the graph may need to be compared to $O(n)$ other nodes in the graph for possible connection.

4.2.2 Subpath Insertion

Direct subpath connection is not the only way to decrease the number of subpaths in P. The next type of transformation, subpath insertion, involves inserting one subpath into the interior of another subpath. To be specific, if $\exists\ P_a, P_b \in P, a \neq b$, with nodes e and f as the endpoints of P_b, and $\exists\ (b, c) \in E(P_a)$ such that edges (b, e) and (f, c) exist and are labeled with labels in C', then remove (b, c) from P_a and add all the edges in P_b in addition to edges (b, e) and (f, c), effectively combining P_a and P_b. As a result, the total number of subpaths $|P|$ is decreased by 1.

Subpath insertion can also be strengthened by considering circular subpaths. If a subpath is circular, then every rotation of that subpath is considered for insertion into another subpath. An example of subpath insertion is shown in Figure 4.

The overall runtime of subpath insertion is $O(n^2)$, as there are $O(n)$ edges among all the subpaths and each edge may be considered for $O(n)$ different insertions in a P with long circular subpaths.

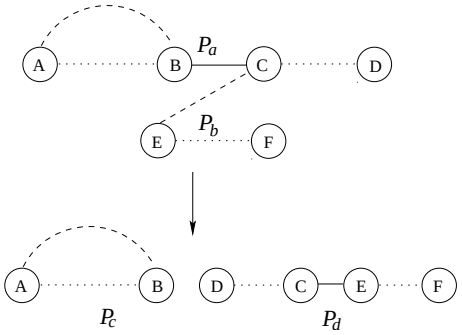


Figure 5: *Example of circular splicing.*

4.2.3 Circular Splicing

While direct subpath connection and subpath insertion are the only two methods used in the Hamiltonian tour repair method that decrease the number of subpaths, other methods are used to increase the algorithm's ability to combine subpaths. The first is called circular splicing. Since both direct subpath connection and subpath insertion are strengthened by considering circular subpaths, circular splicing creates circular subpaths that have a higher likelihood of being combined with other subpaths in future transformations. A circular splice splits an existing subpath P_a into new subpaths P_c and P_d by removing a single edge. P_c must be a circular subpath with more than 2 nodes, and P_d must be able to be combined with another existing subpath via direct subpath connection, as described in Section 4.2.1. As one subpath is split into two and two subpaths are combined into one in this method, there is no net change in the total number of subpaths, $|P|$.

Since longer circular subpaths have a larger likelihood of being combined with other subpaths, the longest splice possible for a given subpath is the one selected. Note that no subpaths of length less than 5 are considered for circular splicing to ensure that the circular subpath generated has length greater than 2. While a circular subpath of length 3 could be extracted from a P_a of length 4, the direct subpath connection of the last node would never be successful because circular splicing is only used after direct subpath connection has failed for all the subpaths in P. An example of circular splicing is shown in Figure 5.

The overall runtime of circular splicing is $O(n^2)$, as there are $O(n)$ edges that could be split in the splice and the direct subpath connection computation could need to consider connection to up to $O(n)$ other nodes.

4.2.4 Non-Circular Splicing

While it is preferable for a circular splice to be carried out when no combinations of subpaths are possible, situations are often encountered in CTSP problem instances for which there are no circular splices possible. In these situations, non-circular splices are carried out instead. A non-circular splice is identical to a circular splice except it does not guarantee a circular subpath to be added to the list of subpaths.

Just like in circular splicing, there is no net change in the total number of subpaths $|P|$. Though no circular subpaths are added to the list of subpaths, non-circular splicing still strengthens attempts to connect the subpaths by introducing new endpoints of paths that may allow future direct subpath connection or subpath insertion.

The runtime of non-circular splicing is $O(n^2)$.

4.2.5 Direct Endpoint Matching

If the subpaths have been combined into a single path, P_f, then all that remains before a Hamiltonian tour can be returned is ensuring an edge with a label in C' exists between the endpoints of the path, completing the tour. While it is best if there already exists an edge between the endpoints of the subpath labeled with an element of C' after combination, this is unusual. In the case in which the extreme nodes cannot be connected, direct endpoint matching is carried out as described in Section 3.2. Results are only accepted if the edges added are elements of C'.

4.2.6 Extended Endpoint Matching

As can be expected from the fact that there are three label constraints for a direct endpoint match to be successful, it is uncommon for a direct endpoint match to be successful. Hence, it is necessary to use extended endpoint matching. Extended endpoint matching considers every possible set of consecutive interior nodes from P_f that can be inserted between the endpoints of P_f using edges labeled with elements in C'. For each of the resulting Hamiltonian paths, direct endpoint matching is performed, and the resulting Hamiltonian tour is returned if this move is successful. Pseudocode for extended endpoint matching is provided in Algorithm 2.

There are $O(n^2)$ total possible paths generated in the first part of extended endpoint matching, and checking each for a direct endpoint match means the extended endpoint matching procedure as a whole operates in $O(n^4)$ runtime.

Algorithm 2 Extended endpoint matching procedure

function extendedEndpointMatching(P_f, C')
// Hamiltonian path P_f; let $V_e(P_f) \equiv \{c, f\}$
// Set C': acceptable labels
for all $(d, a), (b, e) \in E(P_f)$: $a \neq b$, $(c, a), (b, f) \in E$, $c_{(c,a)}, c_{(b,f)} \in C'$ and d, e do not lie between a and b in P_f **do**
  $P'_f \leftarrow P_f$
  $E(P'_f) \leftarrow E(P'_f) \setminus \{(d, a), (b, e)\} \cup \{(c, a), (b, f)\}$
  if direct endpoint matching as described in Sec 4.2.5 can be performed on P'_f successfully, yielding P''_f **then**
    Return P''_f
  end if
end for
Return in failure
end function

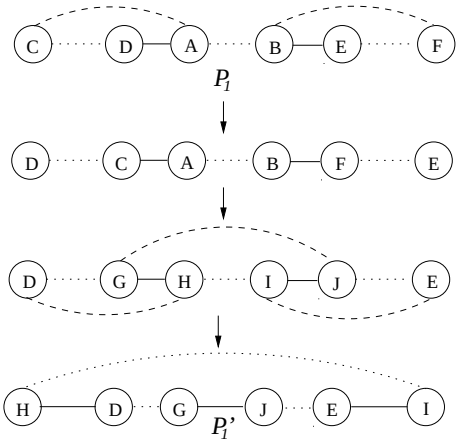


Figure 6: *Example of extended endpoint matching.*

Though extended endpoint matching clearly requires more runtime than direct endpoint matching, it is more successful for generating Hamiltonian tours from n-node paths. An illustration of extended endpoint matching is provided in Figure 6.

4.2.7 Random Endpoint Matching

While extended endpoint matching and direct endpoint matching often generate Hamiltonian tours from n-node paths, these procedures can still fail, introducing the need for random endpoint matching. Random endpoint matching is analogous to extended endpoint matching without the final direct endpoint matchings. Instead, it randomly selects from the valid nodes for connection to the endpoints, returning the transformed result. Pseudocode for random endpoint matching is provided in Algorithm 3.

After random endpoint matching, P_f is replaced with $P_f^{'}$. Even though no Hamiltonian tour is

Algorithm 3 Random endpoint matching procedure

function randomEndpointMatching(P_f, C')
// Hamiltonian path P_f; let $V_e(P_f) \equiv \{c, f\}$
// Set C': acceptable labels
$A \leftarrow \{a \in V : (a, c) \in E \text{ and } c_{(a,c)} \in C'\}$
$B \leftarrow \{b \in V : (b, f) \in E \text{ and } c_{(b,f)} \in C'\}$
Select random $a \in A$ and random $b \in B$
Select $d, e \in V$: $(d, a), (b, e) \in E(P_f)$ and d, e do not lie between a and b in P_f
$P'_f \leftarrow P_f$
$E(P'_f) \leftarrow E(P'_f) \setminus \{(d, a), (b, e)\} \cup \{(c, a), (b, f)\}$
Return P'_f
end function

returned by random endpoint matching, the endpoints of the path stored are changed, meaning this procedure allows for future Hamiltonian tours to be generated.

Since there are $O(n^2)$ pairs of nodes that can connect to the endpoints, random endpoint matching operates in $O(n^2)$ runtime.

4.2.8 Full Repair Algorithm

The repair procedure combines the moves described in the previous sections to heuristically attempt to combine the initial subpaths into a Hamiltonian tour. Pseudocode for the repair procedure is provided in Algorithm 7 in the appendix. The repair heuristic has one parameter, *maxsplice*, controlling both the number of splices and the number of random endpoint matchings. As described in Section 4.2.3, Section 4.2.4, and Section 4.2.7, these procedures only serve to introduce new endpoints for subpaths, helping the heuristic in the future to generate a Hamiltonian tour. Hence, if the parameter value is set to a high value, the heuristic will be more successful in generating Hamiltonian tours, but it will require longer runtimes. The procedure iteratively tries to perform a direct subpath connection, a subpath insertion (trying to favor the creation of a circular path first), or a splicing move (favoring circular ones first). This loop iterates until a Hamiltonian path P_f is obtained or the *maxsplice* limit is reached for the number of splices (in the latter case, the procedure returns in failure). Finally, the procedure tries to transform P_f into a Hamiltonian tour by first checking the trivial case in which its endpoints are connected by an edge with an acceptable label. If this fails, the procedure iteratively performs direct endpoint matching, extended endpoint matching, or random endpoint matching for the next iteration. The procedure returns in failure if a tour cannot be obtained and the *maxsplice* limit is reached for the number of random endpoint matchings.

Since each step in the loop that connects subpaths operates in $O(n^2)$ time and there must be $O(n)$

subpath connections to yield the Hamiltonian path, the subpath connection part of the procedure runs in $O(n^3)$ time. The extended endpoint matching procedure dominates the overall runtime of the whole procedure, which is therefore $O(n^4)$.

4.3 Iterative Deconstruction and Repair Heuristic

The Iterative Deconstruction and Repair (IDR) heuristic is a multistart approach based on the Hamiltonian tour deconstruction method described in Section 4.1 and the Hamiltonian tour repair method described in Section 4.2. The heuristic begins by generating an initial solution by completing the repair heuristic described in Section 4.2 on a list of subpaths for which each element is a single node and the list is randomly ordered. For this repair operation, each label is allowed and the *maxsplice* parameter (see Section 4.2) limits the number of options considered. This method of generation is necessary to allow the generation of feasible solutions for graphs with density less than 1. Since the Hamiltonian tour repair method is not always successful in connecting the disconnected subpaths into a Hamiltonian tour, the algorithm only returns in failure if the repair method failed *maxcreate* times.

After an initial feasible solution is generated, labels are iteratively selected and removed using the deconstruction and repair heuristics with the parameter value set at *maxsplice*. Each label to be removed is selected from a probability distribution, where the probability of selecting a label l for removal is $\frac{\frac{1}{freq_l}}{\sum_{m \in C} \frac{1}{freq_m}}$, where $freq_l$ is the number of times the label l is used in the current Hamiltonian tour, and C is the set of all labels used in the current tour. The procedure terminates when no additional labels can be removed while maintaining a Hamiltonian tour.

The entire procedure of creating a random solution and then iteratively improving it by removing labels is repeated *numrepeat* times, with the best result from those iterations being taken as the final solution for the IDR. The procedure is outlined in Algorithm 4.

The runtime complexity of the IDR is $O(n^4 m)$, because at most m labels can be removed from the current solution and the cost to remove a label is dominated by the repair function, which operates in $O(n^4)$ runtime.

Parameter values used in the testing of this model are provided in Section 5.1.

4.4 Deconstruction and Repair Genetic Algorithm

The Deconstruction and Repair Genetic Algorithm (DRGA) is based in structure and functionality on the genetic algorithm detailed in [22], which is governed by a single parameter p. However, several key

Algorithm 4 Iterative Deconstruction and Repair Heuristic

```
1: function idr(numrepeat, maxcreate)
2: best ← null
3: for iter = 1 to numrepeat do
4:    Attempt to build a Hamiltonian tour H for a max. of maxcreate times, as described in Sec. 4.3
5:    if H is a valid Hamiltonian tour then
6:       flag ← true
7:       while flag = true do
8:          Select l ∈ C(H) with probability 1/freq_l / Σ_{m∈C(H)} 1/freq_m
9:          H' ← deconstruct and repair H removing l, using the procedures described in Sec. 4.1-4.2
10:         if H' is a valid Hamiltonian tour then
11:            H ← H'
12:         else
13:            flag ← false
14:         end if
15:      end while
16:   end if
17:   if best = null or |C(H)| < |C(best)| then
18:      best ← H
19:   end if
20: end for
21: Return best, the best Hamiltonian tour found
22: end function
```

modifications have been made to this effective metaheuristic. The overall structure of the algorithm is reported in Algorithm 5. As can be seen, each generation is composed of p chromosomes, and each chromosome is labeled, resulting in chromosomes $C_0, C_1, \ldots, C_{p-1}$. Then, generations 1 through $p-1$ are then carried out. In generation m, each chromosome C_a is crossed over with chromosome $C_{a+m \mod p}$. If this crossover is successful, the result is mutated and replaces C_a if it has higher fitness (fewer distinct labels in the Hamiltonian tour). Ties are broken randomly. After $p-1$ generations, the most fit chromosome ever encountered is returned as the solution of the DRGA.

Chromosomes are the base unit of genetic algorithms. In the DRGA, each chromosome always represent a feasible solution, that is, the members of the starting population as well as the new individuals resulting from crossover and mutation contain legal Hamiltonian tours along with their associated label sets. The fitness of a chromosome is simply the number of distinct labels it contains — the fewer the number of labels, the higher the fitness.

To create an initial population of chromosomes, each of the p chromosomes is generated randomly using the initialization method described in Section 4.3.

The most crucial element of any genetic algorithm is the crossover operator, in which two parent chromosomes are combined into a child chromosome. The crossover phase between chromosomes C_a and

Algorithm 5 Deconstruction and Repair Genetic Algorithm

```
1: function drga(p)
2: // Parameter p: population size
3: best ← null
4: for all a ∈ 0, ..., p − 1 do
5:    Generate chromosome C_a using the initialization method described in Sec. 4.3
6:    if best = null or C_a has higher fitness than best then
7:       best ← C_a
8:    end if
9: end for
10: for all m ∈ 1, ..., p − 1 do
11:    // m-th generation
12:    for all a ∈ 0, ..., p − 1 do
13:       C'_a ← crossover between C_a and C_{a+m mod p}
14:       C''_a ← mutation of C'_a
15:       if C''_a has higher fitness than C_a then
16:          C_a ← C''_a
17:          if C_a has higher fitness than best then
18:             best ← C_a
19:          end if
20:       end if
21:    end for
22: end for
23: Return best, the chromosome with highest fitness found
24: end function
```

$C_{a+m \mod p}$ starts by selecting for inclusion labels from C_a with a one-half chance. Then, a Hamiltonian tour deconstruction of this parent is carried on, that is the labels not selected for inclusion are removed from C_a and the resulting disconnected subpaths are used to initialize C'_a. Next, we regain feasibility through the repair mechanism described in Section 4.2.8. If the repair procedure cannot obtain a Hamiltonian tour using the current label set of C'_a, labels randomly selected from the list of labels in at least one parent that are not currently in the new chromosome are added one at a time to the set of accepted labels and the repair procedure is repeated until a Hamiltonian tour is again created.

Another important operator for any genetic algorithm is the mutation operator, as this operator allows the amount of diversity in the population of chromosomes to be increased, potentially improving the final result of the genetic algorithm. In the DRGA, the mutation operator which transforms chromosome C'_a into C''_a starts by selecting randomly a label l which is not used by C'_a. Then, l is added to the list of acceptable labels for the chromosome and the IDR heuristic is performed (that is, while attempting the removal of labels from the Hamiltonian tour, edges with l could be accepted by the repair phase and introduced in the solution). Just as in the IDR heuristic, removal ends when no more labels can be removed.

The DRGA operates in $O(p^2mn^4)$ runtime. There are $O(p)$ generations and $O(p)$ chromosomes to be operated on in each generation. These operations consist of a crossover and mutation, each of which operates in $O(mn^4)$ runtime.

Discussion of the parameter selection for the DRGA can be found in Section 5.1.

5 Experimental Results

In this section, we collect output from the heuristics on a number of problem instances to gauge the effectiveness of each technique we described. First, we perform computations on complete problem instances with up to 200 nodes and compare heuristic solutions to the exact solution value, published in [16]. Then, to gain insight into runtime growth and comparative heuristic solutions as problems get larger and sparser, we test just the heuristics on larger problem instances containing up to 1000 nodes and 10000 labels.

5.1 Parameters

For IMPEA, there is only one parameter, p, which is the number of iterations of MVCA followed by MPEA-mod. To have a fair comparison between the procedures, we decided to iterate the MVCA and MPEA-mod until they had consumed as much computation time as a single execution of the DRGA.

For the IDR, there are three parameters: the number of attempts at initial population generation *maxcreate*, the maximum number of splices for the repair procedure *maxsplice*, and the number of times the entire procedure is repeated *numrepeat*. Because the entire procedure fails if an initial feasible solution cannot be created, we set *maxcreate* to the high value of 1000. The *maxsplice* value was set to 100, a value that provides a good tradeoff between runtime and solution quality as shown in the computational results. Again, we decided to repeat the procedure until its computation time exceeds that of the DRGA.

For the DRGA, there are four parameters. Because the DRGA returns in failure if even one of its initial chromosomes fails in creation, we set the number of attempts at initial population generation *maxcreate* to the high value of 1000. Because the repair procedure is used more times in the crossover than it is used in the mutation operation, we set the number of splices allowed in the crossover *maxsplicecrossover* to 25, while we set the limit on splices in mutation *maxsplicemutate* to the higher value of 100. Finally, we set the value of p, the parameter that governs the GA intensity, to 20, a value that performed well in

computational testing and that was similar to the value used in [22].

5.2 Testing Instances

First, we compare the heuristics with the optimal solutions to the pseudorandom problem instances described at the beginning of Section 4 in [16]. As described in that paper, an edge $(i, j), i, j \in \{1, 2, \ldots, n\}$ is given label $\lfloor m(ijA - \lfloor ijA \rfloor) \rfloor \in \{0, 1, \ldots, p-1\}$, where $A = (\sqrt{5} - 1)/2$. Each problem instance is a complete graph, and each combination of $n = 50, 100, 150, 200$ and $m = 50, 100, 150, 200$ is tested, resulting in a total of 16 problem instances.

Moreover, we evaluate the heuristics on the TSPLib-based instances presented in [10]. These instances, which are generated deterministically from the TSPLib dataset found in [20], decide which edges to include using either a pseudorandom or a length-based distribution and decide what labels to assign edges using either a pseudorandom or clustered distribution. Instances with clustered distributions are particularly difficult because labels are localized within the graph and the frequency of each label in the graph is within 1 of the frequency of any other label. To generate these instances, we converted four TSPLib instances, $eil101$, $a280$, $att532$, and $dsj1000$, into labeled instances. Using the method described in [10], we assigned $m = \lfloor \frac{n^2}{100} \rfloor$ labels to each dataset, resulting in datasets with $n = 101$ and $m = 102$, $n = 280$ and $m = 784$, $n = 532$ and $m = 2830$, and $n = 1000$ and $m = 10000$. For each of these converted instances, we generated instances with edge density $d = 0.2, 0.5, 0.8$, and 1. For each combination of n, m, and d, we generated an instance with the length-based frequency distribution and the clustered label distribution, an instance with the length-based frequency distribution and the pseudorandom label distribution, an instance with the pseudorandom frequency distribution and a clustered label distribution, and an instance with the pseudorandom frequency distribution and a pseudorandom label distribution. Therefore, we generated a total of 64 instances based on the TSPLib dataset.[1]

5.3 Testing Environment and Procedures

Computation experiments were performed on a Dell Precision T7600 with 128 GB RAM and two Intel Xeon E5-2687W Processors, each with 8 cores and a clock speed of 3.1 GHz. Because all heuristics compared were implemented on a single thread, none took advantage of the number of processor cores available. All code was compiled in C++. Because all heuristics considered are nondeterministic, we ran

[1] These instances can be found online at *www.rhsmith.umd.edu/faculty/bgolden/* and *josilber.scripts.mit.edu*

	50 nodes		100 nodes		150 nodes		200 nodes	
	Gap	Iter.	Gap	Iter.	Gap	Iter.	Gap	Iter.
IMPEA	29.38%	1590.1	39.46%	623.3	69.73%	459.1	57.00%	369.4
IDR	15.06%	359.6	4.61%	405.1	6.09%	515.1	6.25%	645.7
DRGA	4.78%	1.0	0.33%	1.0	1.69%	1.0	6.13%	1.0
# Nodes	50		100		150		200	
# Labels	50, 100, 150, 200		50, 100, 150, 200		50, 100, 150, 200		50, 100, 150, 200	
Density	1		1		1		1	
Avg. # Sec.	0.40		0.69		1.16		1.73	
# Instances	4		4		4		4	

Table 1: *Gap from optimal and number of iterations on pseudorandom problem instances (average over 40 runs).*

each heuristic 40 times for each instance with a different random seed, averaging the solution qualities and the runtimes in analysis.

Runtimes for the branch-and-cut algorithm described in [16] are for a computer with an Intel Core 2 Duo E6550 2.33 Ghz CPU.

Detailed results for each instance are provided in the appendix. A summary of the results is provided in Section 5.4.

5.4 Performance Analysis

First, we compare the performance of the algorithms on the pseudorandom problem instances from [16], using the branch-and-cut solutions presented in that paper to calculate error values for the heuristics. Table 1 presents both this gap from optimal, "Gap," and an average number of iterations for each heuristic, "Iter." Each heuristic was run until the DRGA completed one iteration; the average amount of time for each size of problem instance is labeled "Avg. # Sec." In this table, IMPEA refers to iterated MPEA-mod, as described in Section 3.3, IDR refers to the Iterative Deconstruction and Repair heuristic described in Section 4.3, and DRGA refers to the Deconstruction and Repair Genetic Algorithm described in Section 4.4. Full results are presented in Table 3, in the appendix.

In the computational comparisons on these small, complete graphs, we found the DRGA outperformed all the other heuristics considered in terms of solution quality. It averaged an error of just 3.23% over the instances, as opposed to an average error of 8.00% for the IDR and 48.89% for the IMPEA. Because we tested each heuristic 40 times for each problem instance, we could use hypothesis testing with the Welch t-test to compare solution qualities. With significance at $\alpha = 0.05$, the DRGA significantly outperformed the IDR on 7/16 instances and the IMPEA on 15/16 instances, and always returned the best solution.

	eil101		a280		att532		dsj1000	
	Gap	Iter.	Gap	Iter.	Gap	Iter.	Gap	Iter.
IMPEA	47.80%	1488.6	66.50%	475.7	59.92%	832.5	45.65%	97.3
IDR	15.11%	379.6	19.48%	536.7	31.11%	446.6	23.67%	681.0
DRGA	5.85%	1.0	6.81%	1.0	5.05%	1.0	3.80%	1.0
# Nodes	101		280		532		1000	
# Labels	102		784		2830		10000	
Density	0.2, 0.5, 0.8, 1.0		0.2, 0.5, 0.8, 1.0		0.2, 0.5, 0.8, 1.0		0.2, 0.5, 0.8, 1.0	
Type	LC, LR, RC, RR		LC, LR, RC, RR		LC, LR, RC, RR		LC, LR, RC, RR	
Avg. # Sec.	1.86		17.08		308.65		597.13	
# Instances	16		16		16		16	

Table 2: *Gap from best heuristic solution and number of iterations on TSPLib-based problem instances (average over 40 runs).*

Of the remaining two heuristics considered, the IDR performed the best in terms of solution quality, obtaining significantly better solutions than the IMPEA on 15/16 instances.

In terms of runtime, all the algorithms performed well on these relatively small problem instances — the DRGA averaged no more more than 3 seconds of runtime for a single problem instance, and single iterations of the IMPEA and IDR completed in small fractions of a second. In comparison, the branch-and-cut algorithm from [16] took more than one minute on 11/16 instances and more than one hour on 3/16 instances. Even though that algorithm was running on different hardware than what the heuristics were tested on (see Section 5.3 for details), it's clear the heuristics provide significant runtime savings, even on these relatively small problem instances.

Next, we compared the heuristics to each other on larger problem instances to investigate runtime growth and comparative solution quality as problem instances become large. A summary of the results are presented in Table 2, and the full results appear in Table 4 in the appendix. In these tables, all gaps are from the best solution encountered among all 40 executions of each heuristic, because the optimal solution is not known for these problem instances. Instance type abbreviations LC, LR, RC, and RR are described in the appendix.

Over the TSPLib-based problem instances, which are generally larger in both size and in the number of labels used in the final solution, the DRGA performed the best in terms of solution quality. It averaged 5.38% deviation from the best heuristic solution for a given problem instance, which was obtained by selecting the best solution from the 40 computed by each algorithm for each problem instance. Meanwhile, the IDR averaged 22.34% deviation and the IMPEA averaged 54.97% deviation. The DRGA had a statistically significantly better solution quality than the IDR on 61/64 instances and the IMPEA on 62/64 instances. It was only outperformed by another algorithm on two large, sparse instances, for which

the IMPEA returned slightly higher quality solutions.

The solution time for the DRGA increased sharply with the size of the problem instance, from seconds on the smallest instances to nearly half an hour on a few large problem instances. The runtime was also affected by the type of problem instance considered — instances with a clustered label distribution took on average 323.3 seconds, while instances with pseudorandom label distributions took only 139.1 seconds on average. Iterations of the IMPEA slowed in comparison to the DRGA runtime as instance size increased; on small instances more than 1000 IMPEA iterations took the same time as one DRGA run, while for larger instances this was less than 100 on average. Meanwhile, iterations of the IDR slightly sped up in comparison to the DRGA as problem instances increased in size.

On several of the small, sparse problem instances, the IMPEA failed to find a feasible solution, even though one existed. The IDR and the DRGA never failed to find a feasible solution on any of the problem instances.

6 Conclusions

The object of study in this paper is the *Colorful Traveling Salesman Problem*, a variant of the well-known Traveling Salesman Problem defined on labeled graphs, with applications in telecommunication and transportation networks.

We designed different heuristic algorithms to solve the problem. We developed two heuristics, the Iterative Deconstruction and Repair Heuristic (IDR) and the Deconstruction and Repair Genetic Algorithm (DRGA), which are based on the idea of removing some of the edges from a feasible solution and reconstructing a Hamiltonian tour from the disconnected subpaths obtained. We also modified an existing heuristic procedure called the Maximum Path Extension Algorithm (MPEA) and used this heuristic along with solutions to the Minimum Label Spanning Tree Problem to approach the CTSP.

All of these algorithms have been tested on different problem instances, and computational results as well as data analysis have been provided. The IDR and the DRGA proved to return the fewest labels of the heuristics tested, with DRGA performing the best. The IDR appears to scale well for larger instances, making it a good candidate for instances for which a quick solution is most important.

Lastly, comparison of the results of the heuristics presented with published optimal solutions for pseudorandom graphs containing up to 200 nodes and labels provided good evidence that the DRGA produces solutions that are close to the optimal solution. The gap between the DRGA's results and the

Instance		IMPEA				IDR				DRGA				B+C	
n	m	Sol.	% Gap	Var.	Iter.	Sol.	% Gap	Var.	Iter.	Sol.	% Gap	Var.	Sec.	Sol.	Sec.
50	50	**4.05**	1.25	0.05	1560.8	**4.00**	0.00	0.00	376.4	**4.00**	0.00	0.00	0.23	4	19
50	100	7.00	40.00	0.00	1595.1	5.63	12.60	0.23	359.7	**5.05**	1.00	0.05	0.35	5	4
50	150	7.98	33.00	0.02	1546.2	7.23	20.50	0.22	342.8	**6.23**	3.83	0.17	0.45	6	3
50	200	10.03	43.29	0.12	1658.4	8.90	27.14	0.19	359.4	**8.00**	14.29	0.05	0.56	7	295
100	50	3.88	29.33	0.11	517.7	**3.00**	0.00	0.00	403.1	**3.00**	0.00	0.00	0.34	3	26
100	100	6.00	50.00	0.00	658.6	**4.00**	0.00	0.00	434.1	**4.00**	0.00	0.00	0.63	4	290
100	150	7.15	43.00	0.13	662.9	5.23	4.60	0.17	375.6	**5.00**	0.00	0.00	0.82	5	2036
100	200	8.13	35.50	0.11	654.1	6.83	13.83	0.14	407.7	**6.08**	1.33	0.07	0.95	6	12583
150	50	4.00	100.00	0.00	421.0	**2.00**	0.00	0.00	553.0	**2.00**	0.00	0.00	0.63	2	46
150	100	5.00	66.67	0.00	527.4	**3.00**	0.00	0.00	493.5	**3.00**	0.00	0.00	1.09	3	91
150	150	6.25	56.25	0.19	451.3	4.23	5.75	0.17	525.9	**4.03**	0.75	0.02	1.33	4	524
150	200	7.80	56.00	0.16	436.7	5.93	18.60	0.07	487.9	**5.30**	6.00	0.21	1.58	5	8872
200	50	3.00	50.00	0.00	323.2	**2.00**	0.00	0.00	657.3	**2.00**	0.00	0.00	0.81	2	152
200	100	4.53	51.00	0.25	355.6	**3.00**	0.00	0.00	677.3	**3.00**	0.00	0.00	1.54	3	289
200	150	6.00	50.00	0.00	424.9	**4.00**	0.00	0.00	632.8	**4.00**	0.00	0.00	2.11	4	4105
200	200	7.08	77.00	0.17	373.9	**5.00**	25.00	0.00	615.3	**4.98**	24.50	0.02	2.45	4	2630

Table 3: Detailed heuristic results on pseudorandom problem instances.

optimal solution was, on average, 3.23%.

7 Acknowledgements

We thank Yupei Xiong for giving us a copy of his MPEA heuristic code. We also thank Gilbert Laporte and Nicolas Jozefowiez for helping us generate the pseudorandom problem instances.

Appendix

In this section, in Algorithm 6 and Algorithm 7, we provide detailed pseudocodes for the MPEA-mod and Hamiltonian Repair procedures, described in Sections 3.2 and 4.2.8. Moreover, this section also contains Table 3 and Table 4, which respectively contain the detailed computational results for each heuristic considered for every pseudorandom and TSPLib-based problem instance considered. In the

Algorithm 6 MPEA-mod

function mpea-mod(C_{init})
// Set C_{init}: initial allowed labels
Randomly select $a, b \in V$, $b \neq a$, such that $(a, b) \in E$ and $c_{(a,b)} \in C_{init}$. If this is impossible, randomly select $a, b \in V$, $b \neq a$, such that $(a, b) \in E$. If this is impossible, return in failure.
Path $P \leftarrow (a, b)$
$C \leftarrow C_{init} \cup \{c_{(a,b)}\}$
while P is not a Hamiltonian path **do**
  $C \leftarrow C \cup C(P)$
  $T \leftarrow$ all rotational transformations of P (see Section 3.2)
  if $\exists\ (k, e) \in E(C)$: $k \in V \setminus V(P)$ and $e \in V_e(P)$ **then**
    $E(P) \leftarrow E(P) \cup \{(k, e)\}$
  else if $\exists\ (a, k), (k, b) \in E(C)$: $k \in V \setminus V(P)$ and $(a, b) \in E(P)$ **then**
    $E(P) \leftarrow E(P) \setminus \{(a, b)\} \cup \{(a, k), (k, b)\}$
  else if $\exists\ Q \in T$, $(k, e) \in E(C)$: $k \in V \setminus V(P)$ and $e \in V_e(Q)$ **then**
    $E(P) \leftarrow E(Q) \cup \{(k, e)\}$
  else if $\exists\ (e_1, e_2), (a, k) \in E(C), (a, b) \in E(P)$: $\{e_1, e_2\} \equiv V_e(P)$ and $k \in V \setminus V(P)$ **then**
    $E(P) \leftarrow E(P) \setminus \{(a, b)\} \cup \{(a, k), (e_1, e_2)\}$
  else if $\exists\ (k, e) \in E$: $k \in V \setminus V(P)$ and $e \in V_e(P)$ **then**
    $K \leftarrow \{(k, e) \in V \setminus V(P) \times V_e(P) : (k, e) \in E\}$
    $k_{max} \leftarrow$ random from $\arg\max_{k \in K}(|\{e \in E : c_e = c_k\}|)$
    $E(P) \leftarrow E(P) \cup \{k_{max}\}$
  else
    $K \leftarrow \bigcup_{Q \in T}\{(k, e) \in V \setminus V(Q) \times V_e(Q) : (k, e) \in E\}$
    if $|K| > 0$ **then**
      $k_{max} \leftarrow$ random from $\arg\max_{k \in K}(|\{(e \in E : c_e = c_k\}|)$
      $Q \leftarrow$ element of T associated to k_{max}
      $E(P) \leftarrow E(Q) \cup \{k_{max}\}$
    else
      $K \leftarrow \{((a, b), k) \in E(P) \times V \setminus V(P) : (a, k), (k, b) \in E\}$
      if $|K| > 0$ **then**
        $((a_{min}, b_{min}), k_{min}) \leftarrow$ random from $\arg\min_{((a,b),k) \in K}(|C \cup \{c_{(a,k)}, c_{(k,b)}\}|)$
        $E(P) \leftarrow E(P) \setminus \{(a_{min}, b_{min})\} \cup \{(a_{min}, k_{min}), (k_{min}, b_{min})\}$
      else
        Return in failure
      end if
    end if
  end if
end while
if $\exists\ (e_1, e_2) \in E$: $\{e_1, e_2\} \equiv V_e(P)$ **then**
  $E(P) \leftarrow E(P) \cup \{(e_1, e_2)\}$
else
  $T \leftarrow$ all rotational transformations of P (see Section 3.2)
  $K \leftarrow \{Q \in T : (e_1, e_2) \in E\}$, where $\{e_1, e_2\} \equiv V_e(Q)$
  if $|K| > 0$ **then**
    $Q_{min} \leftarrow$ random from $\arg\min_{Q \in K}(|C \cup \{c_{(e_1,e_2)}\}|)$, where $\{e_1, e_2\} \equiv V_e(Q)$
    $E(P) \leftarrow E(Q_{min}) \cup \{(e_1, e_2)\}$
  else
    $K \leftarrow$ all Hamiltonian tours that can be generated by direct endpoint matching
    if $|K| > 0$ **then**
      $K_{min} \leftarrow$ random from $\arg\min_{Q \in K} |C(Q)|$
      $E(P) \leftarrow E(K_{min})$
    else
      Return in failure
    end if
  end if
end if
Return P, the final Hamiltonian tour
end function

Algorithm 7 Full Hamiltonian tour repair procedure

function repair(P, C', $maxsplice$)
// Set of subpaths P: created using the procedure from Section 4.1
// Set C': acceptable labels
// Parameter $maxsplice$: maximum number of splices or random endpoint matches
$numsplice \leftarrow 0$
while $|P| > 1$ and $numsplice < maxsplice$ **do**
  if direct subpath connection as described in Sec. 4.2.1 can be performed on P, returning P' **then**
    $P \leftarrow P'$
  else if subpath insertion with at least 1 circular subpath as described in Sec. 4.2.2 can be performed on P, returning P' **then**
    $P \leftarrow P'$
  else if subpath insertion with no circular subpaths as described in Sec. 4.2.2 can be performed on P, returning P' **then**
    $P \leftarrow P'$
  else
    $numsplice \leftarrow numsplice + 1$
    if circular splicing as described in Sec. 4.2.3 can be performed on P, returning P' **then**
      $P \leftarrow P'$
    else if non-circular splicing as described in Sec. 4.2.4 can be performed on P, returning P' **then**
      $P \leftarrow P'$
    end if
  end if
end while
if $|P| > 1$ **then**
  Return in failure
else
  $P_f \leftarrow$ The path of length n contained in P
end if
if $(e_1, e_2) \in E$ and $c_{(e_1,e_2)} \in C'$, where $e_1, e_2 \in V_e(P_f)$ **then**
  $E(P_f) \leftarrow E(P_f) \cup \{(e_1, e_2)\}$
end if
$nummatches \leftarrow 0$
while $nummatches < maxsplice$ **do**
  if direct endpoint matching as described in Sec. 4.2.5 can be performed on P_f, returning P'_f **then**
    Return P'_f
  else if extended endpoint matching as described in Sec 4.2.6 can be performed on P_f, returning P'_f **then**
    Return P'_f
  else
    $nummatches \leftarrow nummatches + 1$
    $P_f \leftarrow$ the result of random endpoint matching as described in Sec 4.2.7
  end if
end while
Return in failure
end function

Instance				IMPEA				IDR				DRGA				Upper
n	m	d	Type	Sol.	% Gap	Var.	Iter.	Sol.	% Gap	Var.	Iter.	Sol.	% Gap	Var.	Sec.	Sol.
101	102	0.2	LC	28.88	31.27	0.48	1892.0	24.70	12.27	0.21	317.3	**22.83**	3.77	0.44	3.56	22
101	102	0.2	LR	17.22	23.00	0.60	2664.4	15.98	14.14	0.22	413.7	**14.43**	3.07	0.24	2.29	14
101	102	0.2	RC	27.26	36.30	0.65	1774.3	22.83	14.15	0.29	323.8	**21.08**	5.40	0.27	3.34	20
101	102	0.2	RR	17.28	32.92	0.20	2431.5	15.33	17.92	0.27	420.2	**13.78**	6.00	0.22	2.19	13
101	102	0.5	LC	17.53	59.36	0.45	1167.5	14.05	27.73	0.15	231.3	**12.35**	12.27	0.33	2.12	11
101	102	0.5	LR	9.40	34.29	0.24	1425.1	7.95	13.57	0.10	415.9	**7.20**	2.86	0.16	1.43	7
101	102	0.5	RC	17.33	57.55	0.37	1061.6	13.48	22.55	0.25	266.7	**12.05**	9.55	0.35	2.09	11
101	102	0.5	RR	9.10	30.00	0.09	1336.5	7.78	11.14	0.17	429.8	**7.05**	0.71	0.05	1.40	7
101	102	0.8	LC	15.03	67.00	0.27	1725.0	10.98	22.00	0.12	383.4	**9.93**	10.33	0.07	2.08	9
101	102	0.8	LR	6.05	21.00	0.05	1032.7	**5.05**	1.00	0.05	408.1	**5.00**	0.00	0.00	1.04	5
101	102	0.8	RC	14.85	85.63	0.13	1562.0	10.08	26.00	0.22	410.9	**8.85**	10.63	0.13	1.94	8
101	102	0.8	RR	7.00	40.00	0.00	985.9	5.08	1.60	0.07	418.5	**5.00**	0.00	0.00	1.07	5
101	102	1.0	LC	13.63	94.71	0.23	1508.1	8.80	25.71	0.21	408.5	**7.93**	13.29	0.07	1.67	7
101	102	1.0	LR	5.15	28.75	0.13	862.1	**4.10**	2.50	0.09	405.0	**4.03**	0.75	0.02	0.95	4
101	102	1.0	RC	13.55	93.57	0.25	1523.9	8.98	28.29	0.02	412.2	**8.05**	15.00	0.20	1.64	7
101	102	1.0	RR	5.18	29.50	0.14	864.9	**4.05**	1.25	0.05	408.1	**4.00**	0.00	0.00	0.96	4
280	784	0.2	LC	92.93	42.97	1.92	432.9	77.10	18.62	0.69	346.6	**67.48**	3.82	1.20	32.36	65
280	784	0.2	LR	39.25	35.34	0.59	751.5	34.63	19.41	0.28	563.8	**30.83**	6.31	0.49	17.45	29
280	784	0.2	RC	91.83	41.28	1.89	400.7	74.45	14.54	1.45	340.0	**66.83**	2.82	1.04	31.04	65
280	784	0.2	RR	33.35	28.27	0.93	738.4	31.13	19.73	0.31	572.9	**27.63**	6.27	0.53	16.40	26
280	784	0.5	LC	68.73	80.87	2.20	329.7	49.55	30.39	0.55	321.2	**41.60**	9.47	1.14	22.92	38
280	784	0.5	LR	21.90	56.43	0.64	489.9	16.45	17.50	0.25	569.9	**15.20**	8.57	0.26	10.52	14
280	784	0.5	RC	67.48	73.03	1.45	303.8	48.30	23.85	0.66	310.1	**40.55**	3.97	0.95	21.48	39
280	784	0.5	RR	17.45	34.23	0.25	496.3	14.70	13.08	0.26	599.2	**13.98**	7.54	0.17	10.04	13
280	784	0.8	LC	61.18	97.35	1.44	564.1	38.95	25.65	0.75	659.1	**33.00**	6.45	0.80	19.70	31
280	784	0.8	LR	15.30	70.00	0.31	426.8	10.60	17.78	0.24	584.3	**9.98**	10.89	0.12	10.00	9
280	784	0.8	RC	59.65	92.42	1.43	529.1	37.63	21.39	0.58	666.9	**32.15**	3.71	0.83	18.79	31
280	784	0.8	RR	12.65	40.56	0.23	397.3	9.95	10.56	0.05	600.8	**9.38**	4.22	0.23	9.88	9
280	784	1.0	LC	56.58	109.56	1.74	498.1	34.00	25.93	0.60	613.2	**28.70**	6.30	0.71	17.06	27
280	784	1.0	LR	12.60	80.00	0.24	366.9	8.15	16.43	0.13	604.2	**7.93**	13.29	0.07	9.23	7
280	784	1.0	RC	56.35	101.25	2.28	515.2	33.93	21.18	0.62	631.5	**28.90**	3.21	0.54	17.14	28
280	784	1.0	RR	12.63	80.43	0.23	369.9	8.10	15.71	0.09	603.8	**7.85**	12.14	0.13	9.23	7
532	2830	0.2	LC	223.85	40.79	5.98	279.4	189.70	19.31	1.76	447.8	**162.60**	2.26	3.29	249.66	159
532	2830	0.2	LR	96.45	39.78	2.95	334.4	83.90	21.59	1.04	520.8	**71.93**	4.25	1.67	91.07	69
532	2830	0.2	RC	223.58	49.05	4.64	262.9	189.45	26.30	3.55	456.9	**155.08**	3.39	6.37	233.34	150
532	2830	0.2	RR	95.30	27.07	2.01	314.0	90.03	20.04	0.92	685.6	**77.75**	3.67	1.69	96.72	75
532	2830	0.5	LC	178.55	66.87	5.15	265.7	147.65	37.99	4.73	267.3	**110.65**	3.41	4.08	199.17	107
532	2830	0.5	LR	59.83	57.45	1.94	228.9	44.58	17.32	0.49	514.1	**40.25**	5.92	0.84	61.37	38
532	2830	0.5	RC	185.80	73.64	6.96	294.2	159.48	49.05	5.00	146.4	**111.08**	3.81	2.47	243.50	107
532	2830	0.5	RR	56.73	41.83	1.40	196.0	45.40	13.50	0.49	705.4	**42.25**	5.63	1.04	62.72	40
532	2830	0.8	LC	163.20	77.39	6.11	732.4	139.93	52.10	3.67	218.7	**95.28**	3.57	2.50	289.75	92
532	2830	0.8	LR	43.08	59.56	1.42	171.4	31.28	15.85	0.25	580.5	**29.23**	8.26	0.67	57.96	27
532	2830	0.8	RC	166.03	78.53	4.57	1300.9	147.30	58.39	8.51	146.0	**96.25**	3.49	3.84	499.48	93
532	2830	0.8	RR	43.78	56.36	1.12	158.1	31.25	11.61	0.44	721.3	**30.00**	7.14	0.65	58.77	28
532	2830	1.0	LC	162.90	79.01	2.89	4798.6	148.35	63.02	9.68	148.6	**95.45**	4.89	5.10	1516.85	91
532	2830	1.0	LR	38.10	65.65	0.99	145.9	25.95	12.83	0.15	739.3	**24.90**	8.26	0.54	55.43	23
532	2830	1.0	RC	163.28	87.68	2.05	3692.1	148.50	70.69	9.15	113.7	**93.78**	7.79	5.02	1167.28	87
532	2830	1.0	RR	37.95	58.13	1.40	145.4	25.95	8.13	0.20	733.2	**25.23**	5.13	0.27	55.35	24
1000	10000	0.2	LC	390.78	48.02	14.02	166.3	332.43	25.92	7.34	791.9	**268.53**	1.72	4.25	1211.46	264
1000	10000	0.2	LR	**180.20**	3.56	5.86	107.1	208.08	19.59	3.82	490.1	181.60	4.37	5.24	678.03	174
1000	10000	0.2	RC	372.85	49.14	17.98	139.5	313.48	25.39	7.95	729.4	**257.55**	3.02	8.30	1107.32	250
1000	10000	0.2	RR	**177.15**	2.99	4.33	104.3	205.93	19.73	4.57	504.7	179.95	4.62	4.65	676.81	172
1000	10000	0.5	LC	247.05	71.56	13.70	121.2	202.43	40.58	4.89	769.6	**147.48**	2.42	3.60	608.17	144
1000	10000	0.5	LR	104.53	4.53	2.00	93.5	110.68	10.68	1.17	754.6	**102.43**	2.43	1.94	432.34	100
1000	10000	0.5	RC	239.00	75.74	20.30	112.6	188.78	38.81	4.17	787.9	**141.88**	4.32	3.06	583.58	136
1000	10000	0.5	RR	102.75	7.03	2.04	93.4	107.88	12.38	0.91	776.0	**100.23**	4.41	2.72	432.14	96
1000	10000	0.8	LC	198.63	94.74	21.63	97.1	142.98	40.18	3.72	664.4	**106.88**	4.78	2.91	613.71	102
1000	10000	0.8	LR	76.75	11.23	1.84	52.8	75.38	9.25	0.58	531.7	**71.38**	3.45	1.78	409.35	69
1000	10000	0.8	RC	196.98	95.03	27.67	101.8	137.20	35.84	4.81	717.8	**103.95**	2.92	1.75	598.38	101
1000	10000	0.8	RR	76.40	12.35	1.39	44.0	74.20	9.12	0.96	460.3	**70.43**	3.57	1.14	411.94	68
1000	10000	1.0	LC	180.93	110.38	42.02	102.5	117.00	36.05	2.60	742.3	**89.25**	3.78	3.04	511.69	86
1000	10000	1.0	LR	64.70	15.54	1.51	63.4	61.15	9.20	0.73	763.1	**58.98**	5.32	1.37	382.60	56
1000	10000	1.0	RC	183.48	113.35	27.00	102.3	117.28	36.37	2.95	742.0	**89.68**	4.28	2.57	512.31	86
1000	10000	1.0	RR	64.53	15.23	1.20	54.5	61.38	9.61	0.43	669.7	**58.98**	5.32	1.17	384.24	56

Table 4: Detailed heuristic results on TSPLib-based problem instances.

column headings, n represents number of nodes $|V|$, m represents the number of labels $|L|$, d is the density of a problem instance, *Sol.* means the average number of labels returned over the 40 testing runs of the heuristic listed or the bound in the case of the upper bound, while $Var.$ is the sample variance of the solutions returned. % *Gap* represents the percentage deviation from the optimal solution for the pseudorandom problem instances and the best encountered heuristic solution for the TSPLib-based instances. *Sec.* represents the average runtime in seconds over the 40 testing runs of the Deconstruction and Repair Genetic Algorithm, while *Iter.* represents the average number of iterations of the other procedures during the execution of the genetic algorithm. For several runs for the iterated MPEA-mod[2], no valid solution was found; in these cases, the average solution and sample variance of solutions is only over the iterations for which a valid solution was returned.

$IMPEA$ represents iterated MPEA-mod, as described in Section 3.3, $DRGA$ is the Deconstruction and Repair Genetic Algorithm described in Section 4.4, IDR is the Iterative Deconstruction and Repair heuristic described in Section 4.3, *Upper* is the best heuristic solution, and $B + C$ is the branch-and-cut algorithm described in [16]. Note that the computations for this algorithm were not performed on the same computer as the computations for the heuristics, and therefore that the runtimes might now be directly comparable. See Section 5.3 for more details.

For TSPLib-based problem instances, the *Type* column heading refers to the generation method of the problem instance. LC refers to generation using a length-based frequency distribution and a clustered label distribution, LR refers to generation using a length-based frequency distribution and a pseudorandom label distribution, RC refers to generation using a pseudorandom frequency distribution and a clustered label distribution, and RR refers to generation using a pseudorandom frequency distribution and a pseudorandom label distribution. For more details on problem instance generation please see [10].

For each testing instance, a procedure's solution is bolded if that solution is not provably different from the best procedure's solutions for that instance, at the $\alpha = 0.05$ level. Hypothesis testing was performed with Welch's t-test with 39 degrees of freedom.

References

[1] M. Albert, A. Frieze, and B. Reed. Multicoloured hamilton cycles. *Electronic Journal of Combinatorics*, 2:R10, 1995.

[2]Failure to return valid solutions occurred only on the TSPLib-based instance from eil101 with density 0.2. For the problem of type LC, 8 runs failed to find a valid solution. For type LR, 3 runs failed. For type RC, 1 run failed. For type RR, 4 runs failed.

[2] H. Broersma and X. Li. Spanning trees with many or few colors in edge-colored graphs. *Discussiones Mathematicae Graph Theory*, 17(2):259–269, 1997.

[3] H. Broersma, X. Li, G. Woeginger, and S. Zhang. Paths and cycles in colored graphs. *Australasian Journal on Combinatorics*, 31:299–311, 2005.

[4] R.D. Carr, S. Doddi, G. Konjedov, and M. Marathe. On the red-blue set cover problem. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 345–353, 2000.

[5] F. Carrabs, R. Cerulli, and M. Gentili. The labeled maximum matching problem. *Computers and Operations Research*, 36(6):1859–1871, 2009.

[6] R. Cerulli, P. Dell'Olmo, M. Gentili, and A. Raiconi. Heuristic approaches for the minimum labelling hamiltonian cycle problem. *Electronic Notes in Discrete Mathematics*, 25(1):131–138, 2006.

[7] R. Cerulli, A. Fink, M. Gentili, and S. Voß. Metaheuristics comparison for the minimum labelling spanning tree problem. In B. Golden, S. Raghavan, and E. Wasil, editors, *The Next Wave in Computing, Optimization, and Decision Technologies*, pages 93–106. Springer, 2005.

[8] R. Cerulli, A. Fink, M. Gentili, and S. Voß. Extensions of the minimum labelling spanning tree problem. *Journal of Telecommunication and Information Technology*, 4:39–45, 2006.

[9] R.S. Chang and S.J. Leu. The minimum labeling spanning trees. *Information Processing Letters*, 63(5):277–282, 1997.

[10] Y. Chen, N. Cornick, A. Hall, R. Sahajpal, J. Silberholz, I. Yahav, and B. Golden. Comparison of heuristics for solving the GMLST problem. In S. Raghavan, B. Golden, and E. Wasil, editors, *Telecommunications Modeling, Policy, and Technology*, pages 191–217. Springer, 2008.

[11] N. Christofides. *Graph theory: An algorithmic approach.* Academic Press, New York, 1975.

[12] B. Couëtoux, L. Gourvès, J. Monnot, and O. Telelis. On labeled traveling salesman problems. In S-H Hong, H. Nagamochi, and T. Fukunaga, editors, *Algorithms and Computation*, pages 776–787. Springer, 2008.

[13] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[14] R. Hassin, J. Monnot, and D. Segev. Approximation algorithms and hardness results for labeled connectivity problems. *Journal of Combinatorial Optimization*, 14(4):437–453, 2007.

[15] R. Jacob, G. Koniedov, S. Krumke, M. Marathe, R. Ravi, and H. Wirth. The minimum label path problem. Unpublished manuscript, Los Alamos National Laboratory, 1999.

[16] N. Jozefowiez, G. Laporte, and F. Semet. A branch-and-cut algorithm for the minimum labeling hamiltonian cycle problem and two variants. *Computers and Operations Research*, 38(11):1534–1542, 2011.

[17] W. Kocay. An extension of the multi-path algorithm for finding hamilton cycles. *Discrete Mathematics*, 102:171–188, 1992.

[18] S.O. Krumke and H.C. Wirth. On the minimum label spanning tree problem. *Information Processing Letters*, 66(2):81–85, 1998.

[19] L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14(4):359–364, 1976.

[20] G. Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[21] Y. Wan, G. Chen, and Y. Xu. A note on the minimum label spanning tree. *Information Processing Letters*, 84(2):99–101, 2002.

[22] Y. Xiong, B. Golden, and E. Wasil. A one-parameter genetic algorithm for the minimum labeling spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 9(1):55–60, 2005.

[23] Y. Xiong, B. Golden, and E. Wasil. Worst-case behavior of the MVCA heuristic for the minimum labeling spanning tree problem. *Operations Research Letters*, 33(1):77–80, 2005.

[24] Y. Xiong, B. Golden, and E. Wasil. Improved heuristics for the minimum labelling spanning tree problem. *IEEE Transactions on Evolutionary Computation*, 10(6):700–703, 2006.

[25] Y. Xiong, B. Golden, and E. Wasil. The colorful traveling salesman problem. In E. Baker, A. Joseph, A. Mehrotra, and M. Trick, editors, *Extending the Horizons: Advances in Computing, Optimization, and Decision Technologies*, pages 115–123. Springer, 2007.