TECHNISCHE UNIVERSITÄT WIEN
Institut für Computergraphik und Algorithmen

Combining Variable Neighborhood Search with Integer Linear Programming for the Generalized Minimum Spanning Tree Problem

Bin Hu and Markus Leitner and Günther R. Raidl

Forschungsbericht / Technical Report

TR–186–1–06–01

10. October 2006

Favoritenstraße 9-11 / E186, A-1040 Wien, Austria
Tel. +43 (1) 58801-18601, Fax +43 (1) 58801-18699
www.cg.tuwien.ac.at

Combining Variable Neighborhood Search with Integer Linear Programming for the Generalized Minimum Spanning Tree Problem

Bin Hu, Markus Leitner, Günther R. Raidl*

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Favoritenstraße 9-11 / 1861
1040 Vienna, Austria
Phone: +431 58801 18611
Fax: +431 58801 18699
{hu|raidl}@ads.tuwien.ac.at

Abstract

We consider the generalized version of the classical Minimum Spanning Tree problem where the nodes of a graph are partitioned into clusters and exactly one node from each cluster must be connected. We present a Variable Neighborhood Search (VNS) approach which uses three different neighborhood types. Two of them work in complementary ways in order to maximize search effectivity. Both are large in the sense that they contain exponentially many candidate solutions, but efficient polynomial-time algorithms are used to identify best neighbors. For the third neighborhood type we apply Mixed Integer Programming to optimize local parts within candidate solution trees. Tests on Euclidean and random instances with up to 1280 nodes indicate especially on instances with many nodes per cluster significant advantages over previously published metaheuristic approaches.

Keywords: Generalized Minimum Spanning Tree, Variable Neighborhood Search, Dynamic Programming, Integer Linear Programming

*This work is supported by the RTN ADONET under grant 504438.

The Generalized Minimum Spanning Tree (GMST) problem is an extension of the classical Minimum Spanning Tree (MST) problem on a graph and is defined as follows. We consider an undirected weighted complete graph $G = \langle V, E, c \rangle$ with node set V, edge set E, and edge cost function $c : E \to \mathbb{R}^+$. Node set V is partitioned into r pairwise disjoint clusters $V_1, V_2, \ldots, V_r$, $\bigcup_{i=1,\ldots,r} V_i = V$, $V_i \cap V_j = \emptyset \ \forall i, j = 1, \ldots, r, \ i \neq j$.

A spanning tree of a graph is a cycle-free subgraph connecting all nodes. A solution to the GMST problem defined on G is a graph $S = \langle P, T \rangle$ with $P = \{p_1, p_2, \ldots, p_r\} \subseteq V$ containing exactly one node from each cluster, i.e. $p_i \in V_i$ for all $i = 1, \ldots, r$, and $T \subseteq P \times P \subseteq E$ being a tree spanning the nodes in P, see Figure 1. The costs of such a tree are its total edge costs, i.e. $C(T) = \sum_{(u,v) \in T} c(u, v)$, and the objective is to identify a solution with minimum costs. We only consider undirected graphs, thus (u, v) is essentially $\{u, v\}$. For better readability, we use (u, v) throughout the article.

In case each cluster contains only one node, i.e. $|V_i| = 1$ for all $i = 1, \ldots, r$, the problem reduces to the simple MST problem, which can be efficiently solved in polynomial time. In general, however, the GMST problem is strongly NP-hard (Myung, Lee, and Tcha, 1995).

There are several real world applications of the GMST problem, e.g. in the design of backbones in large communication networks. Devices belonging to the same existing local area network correspond to nodes within the same cluster, and the backbone is required to connect one device per local network. For a more detailed overview on the GMST problem, see Myung, Lee, and Tcha (1995); Feremans (2001); and Pop (2002).

A variant of the GMST problem is the less restrictive At-Least GMST (LGMST) problem where more than one node is allowed to be connected from each cluster (Ihler, Reich, and Widmayer, 1999; Dror, Haouari, and Chaouachi, 2000). The GMST problem, as well as the LGMST problem, can further be considered as special cases of the Group Steiner Problem (GSP) introduced by Reich and Widmayer (1989). In this more general problem, clusters are replaced by groups of nodes, which are not required to be disjoint, nor do they have to cover all nodes. The objective is to find a subgraph which spans at least one node of each group.

For solving GSP, Duin, Volgenant, and Voß (2004) described a transformation to the classical Steiner tree problem on graphs. We remark that the GMST problem can also be transformed into a (not further constrained) GSP, and therefore, we can solve the GMST problem in principle by means of algorithms for the Steiner tree problem.

In this paper, we present a general Variable Neighborhood Search (VNS) approach for solving the GMST problem. VNS is a metaheuristic which exploits the idea of local search in changing neighborhoods in order to head for a global optimum (Hansen and Mladenovic, 1999, 2003). As local improvement within VNS, we use Variable Neighborhood Descent (VND) utilizing three different types of exponentially large neighborhoods. Two of them are based on complementary representations of candidate solutions. For the third neighborhood we make use of Mixed Integer Programming (MIP).

The remainder of this article is organized as follows. In Section 1, we give an overview on research done on the GMST problem so far. In Section 2, we describe the components of our VNS approach in detail. The instance types we used for testing are explained in Section 3. We show experimental results including a comparison to previous approaches in Section 4 and conclude in Section 5.

1 Previous Work

The GMST problem was introduced by Myung, Lee, and Tcha (1995). They proved that this problem is NP-hard and provided four different Integer Linear Programming (ILP) formulations. Feremans, Labbe, and Laporte (2002) added another four formulations and performed an in-depth investigation on all eight ILPs. Pop (2002) introduced the "Local-Global" MIP formulation. It proved in particular to be more efficient in practice, especially in combination with a relaxation technique called "Rooting Procedure". Instances with up to 240 nodes divided into 30 clusters or 160 nodes divided into 40 clusters could be solved to optimality. Furthermore, Pop utilized the underlying idea of his MIP formulation in a Simulated Annealing approach in order to heuristically solve larger instances. His work also formed a basis for the design of one of the neighborhoods we present in this paper. A more complex Branch-and-Cut algorithm which features new sophisticated cuts and detailed separation procedures has been recently presented by Feremans, Labbe, and Laporte (2004). Nevertheless, large instances can still not be solved to optimality in reasonable time.

Regarding approximation algorithms, Myung, Lee, and Tcha (1995) have shown the inapproximability of the GMST problem in the sense that no approximation algorithm with constant quality guarantee can exist unless P = NP. However, there are better results for some special cases of the problem. Pop, Still, and Kern (2005) described an approximation algorithm for the case when the cluster size is constant. Moreover, Feremans and Grigoriev (2004) provided a Polynomial Time Ap-

proximation Scheme (PTAS) for the special case of the GMST problem with so-called grid-clustering.

To approach more general and larger GMST instances, various metaheuristics have been suggested. Ghosh (2003) implemented and compared a Tabu Search with recency based memory (TS), a Tabu Search with recency and frequency based memory (TS2), a Variable Neighborhood Descent Search, a Reduced VNS, a VNS with steepest descent and a Variable Neighborhood Decomposition Search (VNDS). For all the VNS approaches, he used 1-swap and 2-swap neighborhoods, which are based on the exchange of the spanned nodes within clusters. Comparing these approaches on instances ranging from 100 to 400 nodes partitioned into up to 100 clusters, Ghosh concluded that TS2 and VNDS perform best on average. Golden, Raghavan, and Stanojevic (2005) presented lower and upper bounding procedures. By considering the graph $G' = \langle V', E' \rangle$ in with nodes v'_i of V' correspond to clusters V_i of G and edges exist for any pair of nodes, a lower bound arises when one determines an MST on G' with respect to edge costs $c'(v'_i, v'_j)$ defined as $\min\{c(a,b) \mid (a,b) \in E \wedge a \in V_i \wedge b \in V_j\}$. Furthermore, the authors introduced construction heuristics by adapting Kruskal's (Kruskal, 1956), Prim's (Prim, 1957), and Sollin's algorithm for the classical MST problem, and described a Genetic Algorithm (GA).

A preliminary version of the VNS approach presented in the next sections has been described in Hu, Leitner, and Raidl (2005). The current article extends this previous work by exploiting an additional neighborhood type that is based on the idea of solving small parts of an instance via MIP to optimality.

2 Variable Neighborhood Search for the GMST Problem

In this section, we describe the new VNS approach in detail. First, we consider two constructive heuristics to produce initial solutions. In Section 2.2, we describe the neighborhoods and the search techniques applied to them. Finally, Section 2.3.1 describes the shaking procedure, and Section 2.3.2 explains a memory function to substantially reduce the number of evaluations for the same solutions.

2.1 Initialization

To compute an initial feasible solution for the GMST problem, either a specialized heuristic or an adaption of a standard algorithm for the classical MST problem can be used. Golden, Raghavan,

and Stanojevic (2005) give a comparison between three simple and three improved adaptions of Kruskal's, Prim's, and Sollin's MST algorithms for the GMST problem. While all three improved adaptions produce comparable results, the variant based on Sollin's algorithm in general has the highest computational effort. We therefore adopt the improved version based on Kruskal's MST heuristic and compare it to the rather simple minimum distance heuristic which was also used by Ghosh (2003) to generate initial solutions.

2.1.1 Minimum Distance Heuristic

The Minimum Distance Heuristic (MDH) for computing a feasible initial solution for the GMST problem is shown in Algorithm 1. For each cluster, the node with the lowest sum of edge costs to all nodes in other clusters is determined, and a MST is calculated on these nodes. Using Krukal's algorithm for computing the MST, the complexity of MDH is $O(|V|^2+r^2 \log r)$ where r is the number of clusters.

Algorithm 1: Minimum distance heuristic

for $i = 1, \ldots, r$ **do**
  choose $p_i \in V_i$ with minimal $\sum_{v \in V \setminus V_i} c(p_i, v)$ as the node to be spanned
determine MST T on the subgraph induced by node set $P = \{p_1, \ldots, p_r\}$
return solution $S = \langle P, T \rangle$

2.1.2 Improved Adaption of Kruskal's MST Heuristic

Creating a feasible solution for the GMST by an adaption of Kruskal's algorithm for the classical MST problem is straightforward (Golden, Raghavan, and Stanojevic, 2005). The basic idea is to consider edges in increasing cost-order. An edge is added to the solution iff it does not introduce a cycle and does not connect a second node of any cluster. Obviously, this adaption does not change the time complexity of Kruskal's original algorithm, which is $O(|V| + |E| \log |E|)$.

By fixing an initial node to be in the resulting generalized spanning tree, different solutions can be obtained. The Improved Adaption of Kruskal's MST Heuristic (IKH), as it is called by Golden, Raghavan, and Stanojevic (2005), is shown in Algorithm 2 and follows this idea by running the simple version $|V|$ times, once for each node to be initially fixed. Due to the fact that sorting of edges needs to be done only once, the computational complexity is $O(|V|^2 + |E| \log |E|)$.

Algorithm 2: Improved Kruskal heuristic

for $v \in |V|$ **do**
  fix v to be in the generalized spanning tree
  compute generalized spanning tree with the adaption of Kruskal's MST algorithm
return solution with minimal costs

2.2 Neighborhoods

Our VNS algorithm applies three types of neighborhoods. The first two are based on local search concepts from Ghosh (2003) and Pop (2002). Ghosh represents solutions by the spanned nodes and defines neighborhood structures on them. Optimal edges are derived for a given selection of nodes by determining a classical MST. On the other hand, Pop approaches the GMST problem from an alternative side by representing a solution via its "global connections" – the pairs of clusters which are directly connected. The complete solution is obtained by a decoding function which identifies the best suited nodes and associated edges for the given global connections. The neighborhood of a solution contains all solutions obtained by replacing a global connection by another feasible one. In our third neighborhood type we select reasonably small subgraphs of the whole instance and solve them independently to optimality via the MIP formulation from Pop (2002). These solved parts are then reconnected in a best possible way. The following sections describe the three neighborhood types and the ways we search them in detail.

2.2.1 Node Exchange Neighborhood

In this neighborhood, which was originally proposed by Ghosh (2003), a solution is represented by the set of spanned nodes $P = \{p_1, \ldots, p_r\}$ where p_i is the node to be connected from each cluster V_i, $i = 1, \ldots, r$. Knowing these nodes, there are r^{r-2} possible spanning trees, but one with smallest costs can be efficiently derived by computing a classical MST on the subgraph of G induced by the chosen nodes.

The Node Exchange Neighborhood (NEN) of a solution P consists of all node vectors (and corresponding spanning trees) in which for precisely one cluster V_i the node p_i is replaced by a different node p_i' of the same cluster. This neighborhood therefore consists of $\sum_{i=1}^{r}(|V_i| - 1) = O(|V|)$ different node vectors representing in total $O(|V| \cdot r^{r-2})$ trees. Since a single MST can be computed in $O(r^2)$ time, e.g. by Prim's algorithm, a straight-forward generation and evaluation of the whole neighborhood in order to find the best neighboring solution can be accomplished in $O(|V| \cdot r^2)$ time.

Using an incremental evaluation scheme, we can reduce the computational effort significantly. The goal is to derive from a current minimum-cost tree S represented by P a new minimum-cost tree S' when node p_i is replaced by some node p'_i. Removing p_i and all its incident edges from the initial tree S results in a graph consisting of $k \geq 1$ connected components $T_1, \ldots, T_k$, where usually $k \ll r$.

The new minimum-cost tree S' will definitely not contain new edges within each component $T_1, \ldots, T_k$, because they are connected in the cheapest way as they where optimal in S. New edges are only necessary between nodes of different components and/or p'_i. Furthermore, only the shortest edges connecting any pair of components must be considered. So, the edges of S' must be a subset of

- the edges of S after removing p_i and its incident edges,
- all edges (p'_i, p_j) with $j = 1, \ldots, r \wedge j \neq i$, and
- the shortest edges between any pair of the components $T_1, \ldots, T_k$.

To compute S', we therefore have to calculate the MST of a graph with $(r - k - 1) + (r - 1) + (k^2 - k)/2 = O(r + k^2)$ edges only. Unfortunately, this optimization does not change the worst case time complexity, because identifying the shortest edges between any pair of components may require $O(r^2)$ operations. However, in most practical cases it is substantially faster to compute these shortest edges and to apply Kruskal's MST algorithm on the resulting thin graph. Especially when replacing a leaf node of the initial tree S, we only get a single component plus the new node and the incremental evaluation's benefits are largest.

Exchanging More Than One Node

The above neighborhood can be easily generalized by simultaneously replacing $t \geq 2$ nodes. The computational complexity of a complete evaluation raises to $O(|V|^t \cdot r^2)$. While an incremental computation is still possible in a similar way as described above, the complete evaluation of the neighborhood becomes nevertheless impracticable for larger instances even when $t = 2$. We therefore apply a Restricted Two Nodes Exchange Neighborhood (RNEN2) in which only pairs of clusters that are adjacent in the current solution S are simultaneously considered. Supposing the clusters are of similar size, the time complexity for a complete evaluation is then only $O(|V| \cdot r^2)$.

Nevertheless, RNEN2 is in practice still a relatively expensive neighborhood. Since its complete evaluation consumes too much time in case of large instances, we abort its exploration after a

certain time limit returning the best neighbor identified so far.

2.2.2 Global Edge Exchange Neighborhood

For a given selection of nodes, optimal edges can be determined by an MST algorithm. Pop (2002) has shown that this process can also be reversed: starting from a spanning tree, i.e. a given selection of connections of a so-called "global graph", one can determine optimal vertices (one for each cluster) by another efficient algorithm.

The global graph $G^{\mathrm{g}} = \langle V^{\mathrm{g}}, E^{\mathrm{g}} \rangle$ consists of nodes corresponding to clusters in G, i.e. $V^{\mathrm{g}} = \{V_1, V_2, \ldots, V_r\}$, and edge set $E^{\mathrm{g}} = V^{\mathrm{g}} \times V^{\mathrm{g}}$, see Figure 2.

We now consider a spanning tree $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$ with $T^{\mathrm{g}} \subseteq E^{\mathrm{g}}$ on this global graph. This tree represents the set of all feasible generalized spanning trees on G which contain for each edge $(V_a, V_b) \in T^{\mathrm{g}}$ a corresponding edge $(u, v) \in E$ with $u \in V_a \wedge v \in V_b \wedge a \neq b$. Such a set of trees on G that a particular global spanning tree represents is in general exponentially large with respect to the number of nodes. However, we can use dynamic programming to efficiently determine a minimum cost solution from this set. We start by rooting the global spanning tree at an arbitrary cluster $V_{\mathrm{root}} \in V^{\mathrm{g}}$ and directing all edges towards the leafs. Then, we traverse this tree in a recursive depth-first way calculating for each cluster $V_k \in V^{\mathrm{g}}$ and each node $v \in V_k$ the minimum costs for the subtree rooted in V_k when v is the node to be connected from V_k. These minimum costs of a subtree are determined by the following recursion:

$$C(T^{\mathrm{g}}, V_k, v) = \begin{cases} 0 \text{ if } V_k \text{ is a leaf of the global spanning tree} \\ \sum_{V_l \in Succ(V_k)} \min_{u \in V_l} \{c(v,u) + C(T^{\mathrm{g}}, V_l, u)\} \text{ else,} \end{cases}$$

where $Succ(V_k)$ denotes the set of all successors of V_k in T^{g}. After having determined the minimum costs for the whole tree, the nodes to be used can be easily derived in a top-down fashion by fixing for each cluster $V_k \in V^{\mathrm{g}}$ the node $p_k \in V_k$ yielding minimum costs. This dynamic programming algorithm requires in the worst case $O(|V|^2)$ time and is illustrated in Figure 3.

As Global Edge Exchange Neighborhood (GEEN) for a given global tree T^{g}, we consider any feasible spanning tree differing from T^{g} by precisely one edge. There are $O(r)$ edges which can be removed and $O(r^2)$ feasible ways of reconnecting the resulting two components. If we determine the best neighbor by evaluating all possibilities and naively perform the whole dynamic programming for each global candidate tree, the total time complexity is $O(|V|^2 \cdot r^3)$.

Incremental Dynamic Programming: For a more efficient evaluation of all neighbors, we perform the whole dynamic programming only once at the beginning, keep all costs $C(T^{\mathrm{g}}, V_k, v),\ \forall k = 1, \ldots, r,\ v \in V_k$, and incrementally update our data for each considered move. According to the recursive definition of the dynamic programming approach, we only need to recalculate the values of a cluster V_i if it gets a new child, loses a child, or the costs of a successor change.

Moving to a solution in this neighborhood means to exchange a single global connection (V_a, V_b) by a different connection (V_c, V_d) so that the resulting graph remains a valid tree, see Figure 4. By removing (V_a, V_b), the subtree rooted at V_b is disconnected, hence V_a loses a child and V_a, as well as all its predecessors, must be updated. Before we add (V_c, V_d), we first need to consider the isolated subtree. If $V_d \neq V_b$, we have to re-root the subtree at cluster V_d. Thereby, the old root V_b loses a child. All other clusters which get new children or lose children are on the path from V_b up to V_d, and they must be reevaluated. Otherwise, if $V_d = V_b$, nothing changes within the subtree. When adding the connection (V_c, V_d), V_c gets a new successor and therefore must be updated together with all its predecessors on the path up to the root. In conclusion, whenever we replace a global connection (V_a, V_b) by (V_c, V_d), it is enough to update the costs of V_a, V_b, and all their predecessors on the ways up to the root of the new global tree.

If the tree is not degenerated, its height is $O(\log r)$, and we only need to update $O(\log r)$ clusters of G^{g}. Suppose each of them contains no more than $d_{\max}$ nodes and has at most $s_{\max}$ successors, the time complexity of updating the costs of a single cluster V_i is $O(d_{\max}^2 \cdot s_{\max})$, and the whole process needs time that is bounded by $O(d_{\max}^2 \cdot s_{\max} \cdot \log r)$. The incremental evaluation is therefore much faster than the complete evaluation with its time complexity of $O(|V|^2)$ as long as the trees are not degenerated. An additional improvement is to further avoid unnecessary update calculations by checking if an update actually changes costs of a cluster. If this is not the case, we may skip the update of the cluster's predecessors as long as they are not affected in some other way.

To examine the whole neighborhood of a current solution by using the improved method described above, it is a good idea to choose a processing order that further supports incremental evaluation. Algorithm 3 shows how this is done in detail.

Removing an edge (V_i, V_j) splits our rooted tree into two components: K_1^{g} containing V_i and K_2^{g} containing V_j. The algorithm iterates through all clusters $V_k \in K_1^{\mathrm{g}}$ and makes them root. Each of these clusters is iteratively connected to every cluster of K_2^{g} in the inner loop. The advantage of this calculation order is that none of the clusters in K_1^{g} except its root V_k has to be updated more than

Algorithm 3: Global Edge Exchange Neighborhood (solution S)

```
forall global edges (V_i, V_j) ∈ T^g do
    remove (V_i, V_j)
    M_1 = list of clusters in component K_1^g containing V_i (traversed in preorder)
    M_2 = list of clusters in component K_2^g containing V_j (traversed in preorder)
    forall V_k ∈ M_1 do
        root K_1^g at V_k
        forall V_l ∈ M_2 do
            root K_2^g at V_l
            add (V_k, V_l)
            use incremental dynamic programming to determine the complete solution
                and the objective value
            if current solution better than best then
                save current solution as best
            remove (V_k, V_l)
restore and return best solution
```

once, because global edges are only added between the roots of K_1^{g} and K_2^{g}. Processing clusters in preorder has another additional benefit: Typically, most of the time very few clusters have to be updated when re-rooting either K_1^{g} or K_2^{g}.

2.2.3 Global Subtree Optimization Neighborhood

This neighborhood follows the idea of selecting subproblems of reasonable size, solving them to provable optimality via MIP and merging the results to an overall solution as well as possible. We consider the current GMST $S = \langle P, T \rangle$ with its corresponding global spanning tree $S^{\mathrm{g}} = \langle V^{\mathrm{g}}, T^{\mathrm{g}} \rangle$ defined on the global graph G^{g} as described in Section 2.2.2, i.e. for each edge $(u, v) \in T$ with $u \in V_i \wedge v \in V_j$, there exists a global edge $(V_i, V_j) \in T^{\mathrm{g}}$. After rooting S^{g} at a randomly chosen cluster V_{root}, we perform a depth-first search to determine all subtrees $Q_1, \ldots, Q_k$ containing at least $N_{\min}$ and no more than $N_{\max}$ clusters. Figure 5 shows an example for this selection mechanism with $N_{\min} = 3$ and $N_{\max} = 4$ yielding subtrees $Q_1, \ldots, Q_4$ rooted at $V_1, \ldots, V_4$.

Moving to a solution in the Global Subtree Optimization Neighborhood (GSON) means to optimize one subtree Q_i as an independent GMST problem on the restricted graph induced by the clusters and nodes of Q_i. After solving this subproblem via MIP, we reconnect the new subtree to the remainder of the current overall tree in the best possible way. This can be achieved by inspecting all global edges connecting both components, which is similar as in GEEN. Algorithm 4 summarizes the evaluation of this neighborhood in pseudo-code.

Algorithm 4: Global Subtree Optimization Neighborhood (solution S)

```
V_1, ..., V_k = roots of the subtrees Q_1, ..., Q_k containing at least N_min and
    no more than N_max clusters
forall i = 1, ..., k do
    remove the edge (parent of V_i, V_i) // separate subtree Q_i from S
    optimize Q_i via MIP
    reconnect Q_i to S in a best possible way // as GEEN reconnection mechanism
    if current solution better than best then
        save current solution as best
    restore initial solution
restore and return best solution
```

Whether or not to also consider contained subtrees as Q_2 in addition to Q_1 in Figure 5 was a difficult question while designing GSON. In general, if Q_i contains Q_j, it is not guaranteed that optimizing and reconnecting Q_i would always yield a better result than optimizing and reconnecting only the smaller subtree Q_j. This is possible in particular if the connection between Q_i's root cluster V_i and its predecessor is cheap, but Q_j fits better at a different location. So we decided to include contained subtrees. If $N_{\min}$ and $N_{\max}$ are close, the additional computational effort caused by contained subtrees is relatively low.

The computational complexity of GSON is hard to determine due to the optimization procedure via MIP. If we do not allow overlapping subtrees, the number of subtrees to be considered is bounded below by 0 and above by $\lfloor \frac{r}{N_{\min}} \rfloor$. In our case, we allow contained subtrees, and the number of subtrees to be optimized can be as large as $\lfloor \frac{r}{N_{\max}} \cdot (N_{\max} - N_{\min} + 1) \rfloor$. In our experiments, choosing $N_{\min} = 5$ and $N_{\max} = 6$ yielded the best results.

Local-Global MIP Formulation

In order to solve the subproblems on restricted sets of clusters to optimality, GSON utilizes Pop's local-global MIP formulation (Pop, 2002), which turned out to be more efficient than other formulations when using a general purpose MIP solver as CPLEX. This formulation is based on the fact that for each cluster $V_k,\ k = 1, \ldots, r$, there must be a directed global path from V_k to each other cluster $V_j,\ j \neq k$. For each k, these paths together form a directed tree rooted at V_k. We use the following binary variables.

$$y_{ij} = \begin{cases} 1 & \text{if cluster } V_i \text{ is connected to cluster } V_j \text{ in the global graph} \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j = 1, \ldots, r,\ i \neq j$$

$$\lambda_{kij} = \begin{cases} 1 & \text{if cluster } V_j \text{ is the parent of cluster } V_i \text{ when we root the tree at cluster } V_k \\ 0 & \text{otherwise} \end{cases} \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k$$

$$x_e = \begin{cases} 1 & \text{if edge } (u, v) \in E \text{ appears in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall e = (u, v) \in E$$

$$z_v = \begin{cases} 1 & \text{if node } v \text{ is connected in the solution} \\ 0 & \text{otherwise} \end{cases} \qquad \forall v \in V$$

Pop proved that if the binary incidence matrix y describes a spanning tree of the global graph, then the local solution is integral. Therefore it is sufficient to only force y_{ij} to be integral in the following local-global MIP formulation.

$$\text{(1)} \quad \text{minimize} \quad \sum_{e \in E} c_e x_e$$

$$\text{(2)} \quad \text{subject to} \quad \sum_{v \in V_k} z_v = 1 \qquad \forall k = 1, \ldots, r$$

$$\text{(3)} \quad \sum_{e \in E} x_e = r - 1$$

$$\text{(4)} \quad \sum_{e=(u,v) | u \in V_i, v \in V_j} x_e = y_{ij} \qquad \forall i, j = 1, \ldots, r,\ i \neq j$$

$$\text{(5)} \quad \sum_{e=(u,v) | u \in V_i} x_e \leq z_v \qquad \forall i = 1, \ldots, r,\ \forall v \in V \backslash V_i$$

$$\text{(6)} \quad y_{ij} = \lambda_{kij} + \lambda_{kji} \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k$$

$$\text{(7)} \quad \sum_{j \in \{1, \ldots, r\} \backslash \{i\}} \lambda_{kij} = 1 \qquad \forall i, k = 1, \ldots, r,\ i \neq k$$

$$\text{(8)} \quad \lambda_{kkj} = 0 \qquad \forall j, k = 1, \ldots, r,\ j \neq k$$

$$\text{(9)} \quad \lambda_{kij} \geq 0 \qquad \forall i, j, k = 1, \ldots, r,\ i \neq j,\ i \neq k$$

$$\text{(10)} \quad x_e,\ z_v \geq 0 \qquad \forall e \in E, \forall v \in V$$

$$\text{(11)} \quad y_{lr} \in \{0, 1\}$$

Constraints (2) guarantee that only one node is selected per cluster. Equality (3) forces the solution

to contain exactly $r-1$ edges, while constraints (4) allow them only between nodes of clusters which are connected in the global graph. Inequalities (5) ensure that edges only connect nodes v for which $z_v = 1$. For each $k = 1, \ldots, r$, constraints (6) and (8) force variables λ_{kij} to represent a directed spanning tree: Equalities (6) ensure the selection of a global edge (i, j) iff i is parent of j or j is parent of i in a spanning tree directed out of V_k. Constraints (7) guarantee that each cluster except root k has exactly one parent, while Equalities (8) makes sure that root k has no parents.

2.2.4 Alternative Neighborhoods

When designing GSON we considered several alternative large neighborhoods combining the concepts of the global graph with an exact MIP. One variation of GSON is to first solve all subtrees of limited size exactly and then iterate through a neighborhood structure in which we consider all possibilities of reconnecting these parts. As there are exponentially many such possibilities, the exhaustive exploration turned out to be too expensive in practice.

Another idea for enhancing GSON is to select the clusters inducing a subproblem to be solved exactly not just from the subtrees connected via a single edge to the remaining tree, but from any connected subcomponent of limited size. However, the number of such components is in general too large for a complete enumeration. A practical possibility is to consider the restricted set formed by choosing each cluster as root exactly once and adding $N_{\max} - 1$ further clusters identified via breadth first search. Thus one considers components of the current global tree where the clusters are close to each other. Unfortunately, experiments we performed indicated that the gain of this variant of GSON could not cover its high computational costs.

2.3 Variable Neighborhood Search Framework

We use the general VNS scheme, as shown in Algorithm 5, with VND as local improvement (Hansen and Mladenovic, 1999, 2003). In VND, we alternate between NEN, GEEN, RNEN2, and GSON in this order, see Algorithm 6. This sequence has been determined according to the computational complexity of evaluating the neighborhoods.

Algorithm 5: VNS

```
create initial solution S
repeat
    k = 1
    repeat
        S' = S
        Shake(S', k)
        VND(S')
        if S' better than S then
            S = S'
            k = 1
        else
            k = k + 1
    until k == k_max
until a termination criterion is met
return S
```

Algorithm 6: VND (solution $S = \langle P, T \rangle$)

```
l = 1
repeat
    switch l do
        case 1: // NEN
            S' = Node Exchange Neighborhood (S)
        case 2: // GEEN
            S' = Global Edge Exchange Neighborhood (S) //see Algorithm 3
        case 3: // RNEN2
            S' = Restricted Two Nodes Exchange Neighborhood (S)
        case 4: // GSON
            S' = Global Subtree Optimization Neighborhood (S) //see Algorithm 4
    if solution improved then
        S = S'
        l = 1
    else
        l = l + 1
until l > 4
return S
```

2.3.1 Shaking

It turned out that using a shaking function which puts more emphasis on diversity yields good results for our approach, see Algorithm 7. This shaking process uses both, the NEN and the GEEN structures. For NEN, the number of random moves for shaking starts at three because we have a restricted 2-Opt NEN improvement already included in VND; thus, shaking in NEN with smaller values would mostly lead to the same local optimum as reached before. Shaking in GEEN starts

with two random moves for a similar reason. The number k of random moves increases in steps of two up to $\lfloor \frac{r}{2} \rfloor$.

Algorithm 7: Shake (solution $S = \langle P, T \rangle$, size k)

```
for i = 1, ..., k + 1 do
    randomly change the spanned node p_i of a random cluster V_i
determine the MST T and derive T^g
for i = 1, ..., k do
    remove a randomly chosen global edge e ∈ T^g yielding components K_1^g and K_2^g
    insert a randomly chosen global edge e' connecting K_1^g and K_2^g with e' ≠ e
    determine the spanned nodes p_1, ..., p_r by dynamic programming
return S
```

2.3.2 Memory Function

There is a common situation where VNS unnecessarily spends much time on iterating through all neighborhoods. A local optimum reached by VND is a dead end for all neighborhoods and VNS uses shaking to escape from it. Sometimes, applying VND on the new solution soon leads to the same local optimum. Nevertheless, VND iterates through all neighborhoods again, trying to improve the solution with no success.

We use a hash memory to avoid such situations. For each deterministic neighborhood structure N_i, we store a hash value h_{N_i} of the best solution obtained by it. Before VND tries to improve a solution within N_i, it compares the hash value of the current solution with the memorized hash value h_{N_i}. If they are equal, the evaluation of the neighborhood is skipped, as the current solution cannot be improved by searching through N_i. Since only one hash value per neighborhood structure is memorized at a time, it is not comparable with full-fledged Tabu Search. Nevertheless, this simple concept turned out to save much time in practice.

3 Test Instances

We tested our algorithm on instances used by Ghosh (2003), some further large instances of the same types but with more nodes per cluster created by ourself, and most of the large Euclidean TSPlib[1] instances with geographical clustering (Feremans, 2001).

[1] http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html

Ghosh (2003) created so called grouped Euclidean instances. In this type of instances, squares with side length *span* are associated to clusters and are regularly laid out on a grid of size $col \times row$ as shown in Figure 6. The nodes of each cluster are randomly distributed within the corresponding square. By changing the ratio between cluster separation *sep* and cluster span *span*, it is possible to generate instances with clusters that are overlapping or widely separated. The second type of benchmark instances are so called random Euclidean where nodes of the same cluster are not necessarily close to each other. Such instances are created by simply scattering nodes randomly within a square of size 1000×1000 and making the cluster assignment independently at random. Finally, Ghosh also generated non-Euclidean random instances by choosing all edge costs randomly from the integer interval $[0, 1000]$. All graphs are fully connected. His benchmark set contains instances with up to 1280 nodes, 818560 edges, and 64 clusters; details are listed in Table 1. For each type and size, we consider three different instances. Expanding this benchmark library, we analogously generated further large instances with 600 nodes and 20 clusters, yielding 30 nodes per cluster, using the same algorithms. The values in the columns denote names of the sets, numbers of nodes, numbers of edges, numbers of clusters, and numbers of nodes per cluster. In case of grouped Euclidean instances, numbers of columns and rows of the grid, as well as the cluster separation and cluster span values are additionally given.

Applying geographical clustering (Fischetti, González, and Toth, 1997) on TSPlib instances is done as follows. First, r center nodes are chosen to be located as far as possible from each other. This is achieved by selecting the first center randomly, the second center as the farthest node from the first center, the third center as the farthest node from the set of the first two centers, and so on. Then, clustering is done by assigning each of the remaining nodes to its nearest center node. We consider the largest of such TSPlib instances with up to 442 nodes, 97461 edges, and 89 clusters; details are listed in Table 2. The values in the columns denote names of the instances, numbers of nodes, numbers of edges, numbers of clusters, and the average, minimal, and maximal numbers of nodes per cluster.

4 Experimental Results

In the following, we first present a summary for an experimental comparison of the two constructive heuristics described in Section 2.1, which we consider for the creation of initial solutions for VNS. Computational results of our VNS approach on the different test data sets follow in Section 4.2.

Finally, Section 4.3 analyses the individual contributions of the different neighborhoods within VND. All experiments were performed on a Pentium 4, 2.8GHz PC with 2GB RAM, and we used CPLEX 9.03 to solve the MIP subproblems within GSON.

4.1 Comparison of Construction Heuristics

Table 3 summarizes the comparison of MDH and IKH on all considered input instances. It turned out that IKH performs consistently better than MDH on the TSPlib based, grouped Euclidean, and non-Euclidean instances. Only on random Euclidean instances, MDH could outperform IKH on 70% of the instances. Ratios $\overline{\text{IKH/MDH}}$ indicate the average factor between the objective values of solutions generated by IKH and MDH. Interestingly, the two heuristics never obtained the same solution or solutions of the same quality. As the required CPU-times of both heuristics are very small (less than 80ms for our largest instances with 1280 nodes), we decided to run both, MDH and IKH, and to choose the better result as initial solution for VNS.

4.2 Computational Results for VNS

We compare the results of our VNS to Tabu Search with recency and frequency based memory (TS2) (Ghosh, 2003), Variable Neighborhood Decomposition Search (VNDS) (Ghosh, 2003), the Simulated Annealing (SA) approach from Pop (2002), and, in case of TSPlib instances, also to the Genetic Algorithm (GA) from Golden, Raghavan, and Stanojevic (2005). While TS2 is deterministic, we provide average results over 30 runs for VNDS and VNS and over at least 10 runs for SA (due to its long running times). For TS2, VNDS, and our VNS, runs were terminated when a certain CPU-time limit had been reached. In contrast, SA was run with the same cooling schedule and termination criterion as specified by Pop (2002), which led to significantly longer running times compared to the other algorithms. The results for the GA are adopted from Golden, Raghavan, and Stanojevic (2005).

In Table 4 and 5 we show instance names, numbers of nodes, numbers of clusters, (average) numbers of nodes per cluster, and (average) objective values of the final solutions obtained by the different algorithms. Best values are printed bold. In case of SA and VNS, we also provide corresponding standard deviations of objective values. VNDS produces very stable results as the standard deviations are always zero, except for the second instance of set "Random Eucl 400" where it is 0.34.

For GA, we do not have any standard deviations as they are not listed by Golden, Raghavan, and Stanojevic (2005).

In Table 4 we compare our VNS to TS2, VNDS, and SA on grouped Euclidean instances, random Euclidean instances, and non-Euclidean instances. The time limit was set to 600s for TS2, VNDS, and VNS. In fact, none of the tested algorithms practically needs that much time on smaller instances to find the finally best solutions, but Ghosh (2003) used this time limit as termination criterion, so we decided to retain it. SA required 150s for small instances with 125 nodes and up to about 40000s for the largest instances with 1280 nodes.

When comparing our VNS with SA, we can observe that VNS consistently finds better solutions. Wilcoxon rank sum tests yield error probabilities of less than 1% for the assumption that the mean objective values from VNS are smaller. Also in comparison to VNDS, our VNS is the clear winner. There are only two instances where VNS and VNDS obtained exactly the same mean results and one instance (the second of set "Grouped Eucl 500") on which VNDS performed better. In all other cases, VNS' solutions are superior with high statistical significance (error levels less than 1%). Results of VNS and TS2 are ambiguous. While TS2 usually produces better results on instances with few nodes per cluster, VNS is typically superior when the number of nodes per cluster is higher. This can in particular be observed on instances with 30 nodes per cluster.

On grouped Euclidean instances, the objective values of the final solutions obtained by the considered algorithms, especially those by TS2 and VNS, are relatively close. We assume that these instances are easier to handle as the quality of the solutions are less affected by the differences of the approaches. On random Euclidean instances, especially when the number of nodes per cluster is higher, VNS produces substantially better results than TS2 and VNDS; e.g. for the third instance of set "Random Eucl 600", solutions obtained by VNS are on average 34.4% better than those of TS2. We also observe that SA, which is usually worst, is able to outperform TS2 and VNDS on some of these instances. We conclude that the neighborhood type GEEN, which is also the main component of SA, is very effective on random Euclidean instances and on instances with higher number of nodes per cluster. On non-Euclidean instances, TS2 mostly outperforms all other algorithms.

In Table 5 we compare our VNS to TS2, VNDS, SA, and also the GA on the TSPlib based instances. Results for the GA are adopted from Golden, Raghavan, and Stanojevic (2005), where only smaller instances up to pr226 have been considered. The listed CPU-times were the stopping criteria for TS2, VNDS, and VNS. SA needed up to 10000s for large instances as pcb442. The test runs indicate

that our VNS outperforms VNDS and SA significantly. Wilcoxon rank sum tests again yield error probabilities of less than 1% for the assumptions that the mean objective values from VNS are smaller. Judging by the few results for GA, VNS finds solutions which are at least as good as those of GA. Considering VNS and TS2, we cannot draw clear conclusions. Most of the time, these two algorithms generate comparable results under the same conditions. We omitted smaller TSPlib instances in Table 5 as the most capable algorithms TS2, GA, and VNS were all able to (almost) always provide optimal solutions as found by the exact Branch-and-Cut algorithm from Feremans, Labbe, and Laporte (2004). The latter could solve all instances with up to 200 nodes except d198 to provable optimality in up to 5254s CPU time.

In overall, VNS and TS2 are the most powerful algorithms among all considered approaches. Out of 46 instances we have tested, VNS produces strictly better results in 20 cases, TS2 is better in 16 cases, and on 10 instances, they are equally good.

4.3 Contributions of Neighborhoods

In order to analyze how the different neighborhood structures of VNS contribute to the whole optimization, we logged how often each one was able to improve on a current solution and their absolute gains. Table 6 shows the ratios of successful improvements in contrast to how often each neighborhood structure was evaluated. These values are grouped by the different types of input instances. On the other hand, Table 7 shows their absolute gains, i.e. their contribution in percentage to the difference between objective values of the starting and the final solutions.

In general, each neighborhood structure contributes substantially to the whole success. NEN and RNEN2 are most effective in terms how often they improve on a solution, whereas the differences in the objective values achieved by single improvements are significant larger in case of GEEN. Considering that GSON operates on solutions which are already local optima with respect to all other neighborhoods, both its improvement ratios and its absolute gains are remarkable. Regarding the different instance sets, we also observe that the improvement ratio of GEEN generally increases with the size of nodes per cluster.

In addition, Table 8 and 9 show tests on switching particular neighborhood structures off. We compare results obtained by using all neighborhood structures, turning off NEN, turning off NEN as well as RNEN2, turning off GEEN, and turning off GSON. Obviously, omitting NEN and RNEN2

performs worst. By switching off GEEN, we get comparable results on Group Euclidean instances and Non-Euclidean instances, but significantly worse results on Random Euclidean instances. Data on runs without GSON are taken from our previous paper (Hu, Leitner, and Raidl, 2005). These results are generally inferior compared to the current results.

4.4 Adjusting the Size of the Global Subtree Optimization Neighborhood

The primary adjustment parameter for GSON is the size of the subtrees to be optimized via MIP. These subtrees contain at least $N_{\min}$ and at most $N_{\max}$ clusters. In Table 10 and 11 we study the influence of different values for these parameters. In general, results are ambiguous. On Random Euclidean instances, a tendency towards smaller sizes yielding better results is noticeable. On some other instances, the search process benefits from larger values as the neighborhood is searched more extensively. We decided to set $N_{\min} = 5$ and $N_{\max} = 6$ as default behavior for a balanced behavior.

4.5 Using Different Starting Solutions

Table 12 and 13 show the importance of having good starting solutions for our VNS. We compare the quality of the final solutions when using MDH/IKH as initialization heuristics in contrast to starting with random solutions. The latter are constructed by choosing a random node for each cluster and connecting them via Kruskal's MST algorithm. Using MDH and IKH improves the quality of final solutions in most cases. On smaller instances, starting with a random solution leads to the same results as VNS is powerful enough and has enough time available. When facing more difficult instances, time becomes more crucial and therefore starting with a superior solution proves to be advantageous.

5 Summary and Conclusions

In this paper, we proposed a general Variable Neighborhood Search (VNS) approach for solving the Generalized Minimum Spanning Tree problem. For initializing the solution, we use the Minimum Distance Heuristic and the Improved Adaption of Kruskal's MST Heuristic, which are both based on Kruskal's classical algorithms for determining a MST. Though their performance depends on the instance type, the latter construction heuristic mostly yields better results.

Our Variable Neighborhood Descent combines three neighborhood types: For the Node Exchange Neighborhood, solutions are represented by the spanned nodes and one node is replaced by another of the same cluster. Optimal edges are derived by determining a classical MST on these nodes. The Global Edge Exchange Neighborhood works in a complementary way by considering for a solution primarily its global connections, i.e. pairs of clusters which are directly connected. Neighbors are all solutions differing in exactly one global connection. Knowing this global structure for a solution, dynamic programming is used to determine the best suited nodes and concrete edges. For both of these neighborhoods, incremental evaluation schemes have been described, which speed up the whole computation considerably. For the Global Subtree Optimization Neighborhood, we consider subsets of clusters which are reorganized via Mixed Integer Programming and then reconnected to the remainder as well as possible.

Tests were performed on TSPlib instances, grouped Euclidean instances, random Euclidean instances, and non-Euclidean instances. Results show that the proposed VNS algorithm is able to reliably produce high quality solutions, sometimes having significantly lower objective values than the solutions from previously leading metaheuristic approaches. This holds in particular for instances with large number of nodes per cluster. On grouped Euclidean and TSPlib based instances, the differences between the objective values of the final solutions obtained by our VNS and the other candidate algorithms are relatively low, which indicates that the structure of these instances is simpler. Differences between the considered algorithms are largest on random Euclidean instances. In this case, VNS produces substantially better results due to the effectiveness of the Global Edge Exchange Neighborhood.

References

Dror, M., M. Haouari, and J. S. Chaouachi (2000). "Generalized spanning trees." *European Journal of Operational Research* 120, 583–592.

Duin, C. W., A. Volgenant, and S. Voß (2004). "Solving group Steiner problems as Steiner problems." *European Journal of Operational Research* 154, issue 1, 323–329.

Feremans, C. (2001). *Generalized Spanning Trees and Extensions*. Ph.D. thesis, Universite Libre de Bruxelles.

Feremans, C. and A. Grigoriev (2004). "An Approximation Scheme for the Generalized Geometric Minimum Spanning Tree Problem with Grid Clustering." Technical Report NEP-ALL-2004-09-30, Maastricht: METEOR, Maastricht Research School of Economics of Technology and Organization.

Feremans, C., M. Labbe, and G. Laporte (2002). "A Comparative Analysis of Several Formulations for the Generalized Minimum Spanning Tree Problem." *Networks* 39(1), 29–34.

Feremans, C., M. Labbe, and G. Laporte (2004). "The Generalized Minimum Spanning Tree Problem: Polyhedral Analysis and Branch-and-Cut Algorithm." *Networks* 43, issue 2, 71–86.

Fischetti, M., J. J. S. González, and P. Toth (1997). "A branch-and-cut algorithm for the symmetric generalized traveling salesman problem." *Operations Research* 45, 378–394.

Ghosh, D. (2003). "Solving Medium to Large Sized Euclidean Generalized Minimum Spanning Tree Problems." Technical Report NEP-CMP-2003-09-28, Indian Institute of Management, Research and Publication Department, Ahmedabad, India.

Golden, B., S. Raghavan, and D. Stanojevic (2005). "Heuristic Search for the Generalized Minimum Spanning Tree Problem." *INFORMS Journal on Computing* 17(3), 290–304.

Hansen, P. and N. Mladenovic (1999). "An introduction to Variable Neighborhood Search." In *Meta-heuristics, Advances and trends in local search paradigms for optimization*, S. Voss, S. Martello, I. H. Osman, and C. Roucairol, eds., Kluwer Academic Publishers. 433–458.

Hansen, P. and N. Mladenovic (2003). "A tutorial on Variable Neighborhood Search." Technical Report G-2003-46, Les Cahiers du GERAD, HEC Montreal and GERAD, Canada.

Hu, B., M. Leitner, and G. R. Raidl (2005). "Computing Generalized Minimum Spanning Trees with Variable Neighborhood Search." In *Proceedings of the 18th Mini Euro Conference on Variable*

Neighborhood Search, P. Hansen, N. Mladenović, J. A. M. Pérez, B. M. Batista, and J. M. Moreno-Vega, eds. Tenerife, Spain.

Ihler, E., G. Reich, and P. Widmayer (1999). "Class Steiner Trees and VLSI-design." *Discrete Applied Mathematics* 90, 173–194.

Kruskal, J. B. (1956). "On the shortest spanning subtree and the traveling salesman problem." In *Proceedings of the American Mathematical Society*. volume 7, 48–50.

Myung, Y. S., C. H. Lee, and D. W. Tcha (1995). "On the Generalized Minimum Spanning Tree Problem." *Networks* 26, 231–241.

Pop, P. C. (2002). *The Generalized Minimum Spanning Tree Problem*. Ph.D. thesis, University of Twente, The Netherlands.

Pop, P. C., G. Still, and W. Kern (2005). "An Approximation Algorithm for the Generalized Minimum Spanning Tree Problem with Bounded Cluster Size." In *Algorithms and Complexity in Durham 2005, Proceedings of the first ACiD Workshop*, H. Broersma, M. Johnson, and S. Szeider, eds. King's College Publications, volume 4 of *Texts in Algorithmics*, 115–121.

Prim, R. C. (1957). "Shortest connection networks and some generalisations." *Bell System Technical Journal* 36, 1389–1401.

Reich, G. and P. Widmayer (1989). "Beyond Steiner's Problem: A VLSI Oriented Generalization." In *Graph-Theoretic Concepts in Computer Science WG89*. 196–210.

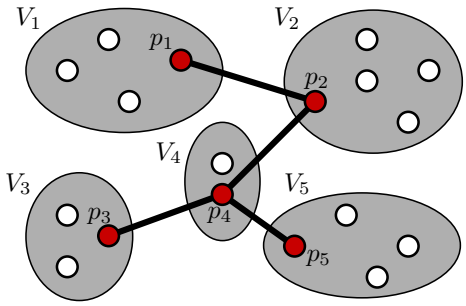

Figure 1: Example for a GMST solution.

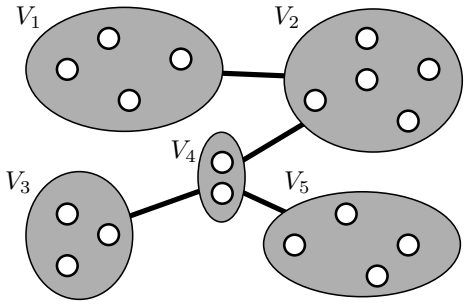

Figure 2: A global graph G^{g}.

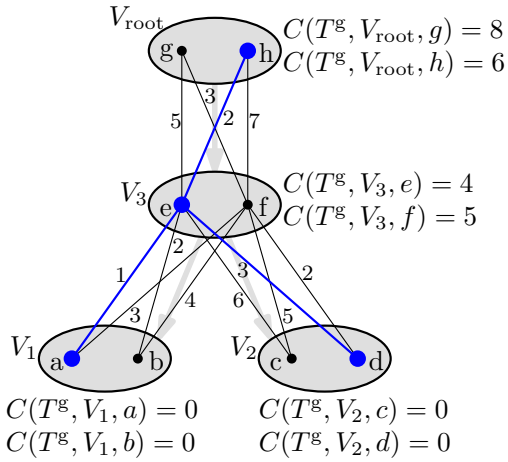

Figure 3: Determining the minimum-cost values for each cluster and node. The tree's total minimum costs are $C(T^{\mathrm{g}}, V_{\mathrm{root}}, h) = 6$, and the finally selected nodes are printed bold.

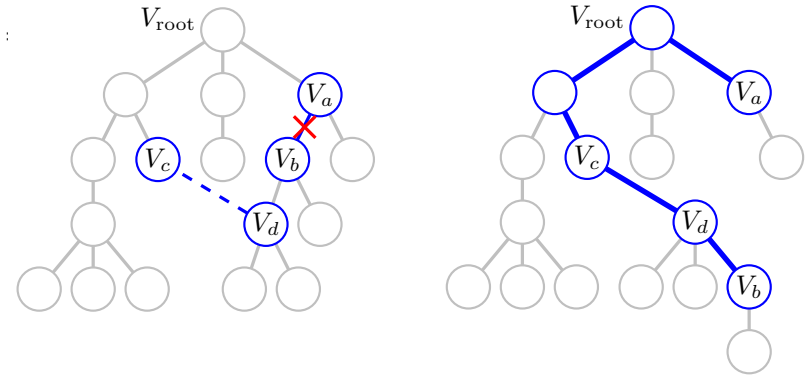

Figure 4: After removing (V_a, V_b) and inserting (V_c, V_d), only the clusters on the paths from V_a to V_{root} and V_b to V_{root} must be reconsidered.

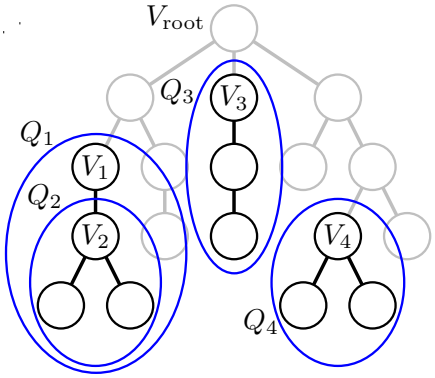

Figure 5: Selection of subtrees to be optimized via MIP.

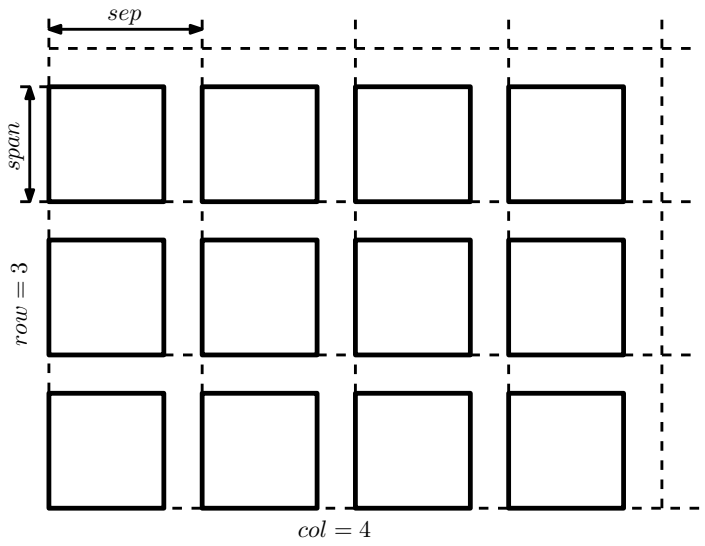

Figure 6: Creation of Grouped Euclidean Instances.

Table 1: Benchmark instance sets adopted from Ghosh (2003) and correspondingly created new sets (marked by *). Each instance has a constant number of nodes per cluster.

Instance set	$\|V\|$	$\|E\|$	r	$\frac{\|V\|}{r}$	col	row	sep	$span$
Grouped Eucl 125	125	7750	25	5	5	5	10	10
Grouped Eucl 500	500	124750	100	5	10	10	10	10
Grouped Eucl 600*	600	179700	20	30	5	4	10	10
Grouped Eucl 1280	1280	818560	64	20	8	8	10	10
Random Eucl 250	250	31125	50	5	-	-	-	-
Random Eucl 400	400	79800	20	20	-	-	-	-
Random Eucl 600*	600	179700	20	30	-	-	-	-
Non-Eucl 200	200	19900	20	10	-	-	-	-
Non-Eucl 500	500	124750	100	5	-	-	-	-
Non-Eucl 600*	600	179700	20	30	-	-	-	-

Table 2: TSPlib instances with geographical clustering (Feremans, 2001). Numbers of nodes vary for each cluster.

Instance name	$\|V\|$	$\|E\|$	r	$\frac{\|V\|}{r}$	$d_{\min}$	$d_{\max}$
gr137	137	9316	28	5	1	12
kroa150	150	11175	30	5	1	10
d198	198	19503	40	5	1	15
krob200	200	19900	40	5	1	8
gr202	202	20301	41	5	1	16
ts225	225	25200	45	5	1	9
pr226	226	25425	46	5	1	16
gil262	262	34191	53	5	1	13
pr264	264	34716	54	5	1	12
pr299	299	44551	60	5	1	11
lin318	318	50403	64	5	1	14
rd400	400	79800	80	5	1	11
fl417	417	86736	84	5	1	22
gr431	431	92665	87	5	1	62
pr439	439	96141	88	5	1	17
pcb442	442	97461	89	5	1	10

Table 3: Comparison of the construction heuristics MDH and IKH.

Instance Type	MDH better %	IKH better %	$\overline{\text{IKH/MDH}}$
TSBlib based	0	100	0.89
Grouped Euclidean	0	100	0.85
Random Euclidean	70	30	1.36
Non-Euclidean	0	100	0.16

Table 4: Results on instance sets from Ghosh (2003) and correspondingly created new sets, 600s CPU-time (except SA). Three different instances are considered for each set.

Instances				TS2	VNDS	SA		VNS	
Set	$\|V\|$	r	$\|V\|/r$	$C(T)$	$\overline{C(T)}$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped Eucl 125	125	25	5	**141.1**	**141.1**	152.3	0.52	**141.1**	0.00
	125	25	5	**133.8**	**133.8**	150.9	0.74	**133.8**	0.00
	125	25	5	143.9	145.4	156.8	0.00	**141.4**	0.00
Grouped Eucl 500	500	100	5	**566.7**	577.6	642.3	0.00	567.4	0.57
	500	100	5	**578.7**	584.3	663.3	1.39	585.0	1.32
	500	100	5	**581.6**	588.3	666.7	1.81	583.7	1.82
Grouped Eucl 600	600	20	30	85.2	87.5	93.9	0.00	**84.6**	0.11
	600	20	30	**87.9**	90.3	99.5	0.28	**87.9**	0.00
	600	20	30	88.6	89.4	99.2	0.17	**88.5**	0.00
Grouped Eucl 1280	1280	64	20	327.2	329.2	365.1	0.46	**315.9**	1.91
	1280	64	20	322.2	322.5	364.4	0.00	**318.3**	1.78
	1280	64	20	332.1	335.5	372.0	0.00	**329.4**	1.29
Random Eucl 250	250	50	5	**2285.1**	2504.9	2584.3	23.82	2300.9	40.27
	250	50	5	**2183.4**	2343.3	2486.7	0.00	2201.8	23.30
	250	50	5	**2048.4**	2263.7	2305.0	16.64	2057.6	31.58
Random Eucl 400	400	20	20	**557.4**	725.9	665.1	3.94	615.3	10.8
	400	20	20	724.3	839.0	662.1	7.85	**595.3**	0.00
	400	20	20	604.5	762.4	643.7	14.54	**587.3**	0.00
Random Eucl 600	600	20	30	541.6	656.1	491.8	7.83	**443.5**	0.00
	600	20	30	540.3	634.0	542.8	25.75	**537.0**	10.2
	600	20	30	627.4	636.5	469.5	2.75	**469.0**	11.9
Non-Eucl 200	200	20	10	**71.6**	94.7	76.9	0.21	**71.6**	0.00
	200	20	10	**41.0**	76.6	41.1	0.02	**41.0**	0.00
	200	20	10	**52.8**	75.3	86.9	5.38	**52.8**	0.00
Non-Eucl 500	500	100	5	**143.7**	203.2	200.3	4.44	152.5	3.69
	500	100	5	**132.7**	187.3	194.3	1.20	148.6	4.27
	500	100	5	**162.3**	197.4	205.6	0.00	166.1	2.89
Non-Eucl 600	600	20	30	**14.5**	59.4	22.7	1.49	15.6	1.62
	600	20	30	17.7	23.7	22.0	0.82	**16.1**	1.24
	600	20	30	**15.1**	29.5	22.1	0.44	16.0	1.66

Table 5: Results on TSPlib instances with geographical clustering, $\frac{|V|}{r} = 5$, variable CPU-time.

TSPlib Instances				TS2	VNDS	SA		GA	VNS	
Name	$\|V\|$	r	time	$C(T)$	$\overline{C(T)}$	$\overline{C(T)}$	std dev	$\overline{C(T)}$	$\overline{C(T)}$	std dev
gr137	137	28	150s	**329.0**	330.0	352.0	0.00	**329.0**	**329.0**	0.00
kroa150	150	30	150s	**9815.0**	**9815.0**	10885.6	25.63	**9815.0**	**9815.0**	0.00
d198	198	40	300s	7062.0	7169.0	7468.73	0.83	**7044.0**	**7044.0**	0.00
krob200	200	40	300s	11245.0	11353.0	12532.0	0.00	**11244.0**	**11244.0**	0.00
gr202	202	41	300s	**242.0**	249.0	258.0	0.00	243.0	**242.0**	0.00
ts225	225	45	300s	62366.0	63139.0	67195.1	34.49	62315.0	**62268.5**	0.51
pr226	226	46	300s	**55515.0**	**55515.0**	56286.6	40.89	**55515.0**	**55515.0**	0.00
gil262	262	53	300s	**942.0**	979.0	1022.0	0.00	-	942.3	1.02
pr264	264	54	300s	**21886.0**	22115.0	23445.8	68.27	-	21886.5	1.78
pr299	299	60	450s	20339.0	20578.0	22989.4	11.58	-	**20322.6**	14.67
lin318	318	64	450s	18521.0	18533.0	20268.0	0.00	-	**18506.8**	11.58
rd400	400	80	600s	**5943.0**	6056.0	6440.8	3.40	-	5943.6	9.69
fl417	417	84	600s	7990.0	7984.0	8076.0	0.00	-	**7982.0**	0.00
gr431	431	87	600s	1034.0	1036.0	1080.5	0.51	-	**1033.0**	0.18
pr439	439	88	600s	51852.0	52104.0	55694.1	45.88	-	**51847.9**	40.92
pcb442	442	89	600s	**19621.0**	19961.0	21515.1	5.15	-	19702.8	52.11

Table 6: Individual improvement rates of NEN, GEEN, RNEN2, and GSON.

Instance Type	$\|V\|$	r	$\|V\|/r$	NEN	GEEN	RNEN2	GSON
TSBlib based	n.a.	n.a.	5	0.55	0.44	0.67	0.18
Grouped Euclidean	125	25	5	0.54	0.49	0.72	0.15
	500	100	5	0.55	0.41	0.76	0.16
	600	20	30	0.58	0.54	0.74	0.23
	1280	64	20	0.63	0.45	0.70	0.46
Random Euclidean	250	50	5	0.74	0.30	0.95	0.09
	400	20	20	0.59	0.42	0.88	0.10
	600	20	30	0.57	0.53	0.81	0.07
Non-Euclidean	200	20	10	0.78	0.43	0.60	0.06
	500	100	5	0.80	0.16	0.68	0.24
	600	20	30	0.79	0.49	0.56	0.09

Table 7: Individual absolute gains of NEN, GEEN, RNEN2, and GSON.

Instance Type	$\|V\|$	r	$\|V\|/r$	NEN	GEEN	RNEN2	GSON
TSBlib based	n.a.	n.a.	5	16.58%	60.71%	15.64%	7.07%
Grouped Euclidean	125	25	5	18.02%	63.40%	13.61%	4.96%
	500	100	5	23.17%	45.19%	28.32%	3.31%
	600	20	30	13.60%	75.85%	7.29%	3.26%
	1280	64	20	16.72%	40.08%	20.45%	22.76%
Random Euclidean	250	50	5	16.90%	48.52%	16.06%	18.52%
	400	20	20	16.14%	66.75%	15.13%	1.98%
	600	20	30	21.30%	63.43%	13.65%	1.62%
Non-Euclidean	200	20	10	10.89%	48.07%	11.92%	29.13%
	500	100	5	11.74%	60.04%	24.75%	3.47%
	600	20	30	11.78%	74.80%	10.65%	2.78%

Table 8: Results on Ghosh' instance sets when switching off certain neighborhood structures.

Instances	VNS		w.o. NEN		w.o. all NENs		w.o. GEEN		w.o. GSON	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped Eucl 125	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00
	133.8	0.00	**133.8**	0.00	**133.8**	0.00	**133.8**	0.00	**133.8**	0.00
	141.4	0.00	**141.4**	0.00	**141.4**	0.00	**141.4**	0.00	**141.4**	0.00
Grouped Eucl 500	**567.4**	0.57	569.9	2.84	589.6	5.56	567.7	0.48	568.6	0.59
	585.0	1.32	584.0	0.92	601.8	5.33	586.5	1.18	**581.0**	1.39
	583.7	1.82	584.9	1.46	597.1	2.37	**583.3**	1.91	587.9	4.07
Grouped Eucl 600	**84.6**	0.11	**84.6**	0.00	84.8	0.27	**84.6**	0.11	84.8	0.27
	87.9	0.00	**87.9**	0.00	88.3	0.24	**87.9**	0.02	**87.9**	0.05
	88.5	0.00	**88.5**	0.00	88.7	0.12	**88.5**	0.00	**88.5**	0.00
Grouped Eucl 1280	315.9	1.91	316.8	2.54	327.1	4.30	**314.6**	1.10	321.8	2.41
	318.3	1.78	319.8	1.54	326.4	3.01	317.8	0.79	**316.3**	0.83
	329.4	1.29	330.7	0.94	339.9	3.87	329.6	2.16	334.3	2.13
Random Eucl 250	**2300.9**	40.27	2354.5	48.14	2646.6	28.49	2320.0	43.08	2336.9	34.23
	2201.8	23.30	2235.6	37.99	2576.2	112.15	2199.7	21.94	2304.1	47.95
	2057.6	31.58	2093.5	52.39	2460.6	131.82	2061.2	26.91	**2049.8**	15.29
Random Eucl 400	**615.3**	10.8	622.3	14.82	703.4	53.40	621.9	14.20	625.4	14.59
	595.3	0.00	**595.3**	0.00	671.4	25.82	**595.3**	0.00	**595.3**	0.14
	587.3	0.00	**587.3**	0.00	657.4	50.38	597.5	17.15	588.8	7.40
Random Eucl 600	**443.5**	0.00	450.5	20.85	506.5	46.08	452.9	33.57	443.5	0.00
	537.0	10.2	539.9	11.17	685.0	21.40	545.1	18.05	**535.2**	12.20
	469.0	11.9	474.5	20.57	643.4	63.68	493.8	35.76	479.9	26.55
Non-Eucl 200	**71.6**	0.00	**71.6**	0.05	97.3	14.23	**71.6**	0.00	**71.6**	0.02
	41.0	0.00	**41.0**	0.00	58.5	11.18	**41.0**	0.00	**41.0**	0.00
	52.8	0.00	**52.8**	0.00	56.8	0.69	**52.8**	0.00	**52.8**	0.00
Non-Eucl 500	**152.5**	3.69	159.3	3.41	203.3	0.00	155.4	3.94	173.4	8.40
	148.6	4.27	162.2	5.97	237.9	1.28	151.2	5.41	154.6	6.55
	166.1	2.89	176.0	4.72	282.5	0.00	167.5	4.36	180.1	3.67
Non-Eucl 600	15.6	1.62	18.9	3.22	47.5	9.40	**15.3**	1.35	15.9	2.07
	16.1	1.24	18.9	2.16	36.0	3.67	16.3	1.24	17.6	1.75
	16.0	1.66	18.8	2.29	42.0	5.87	16.2	1.97	**15.1**	0.22

Table 9: Results on TSPlib instances when switching off certain neighborhood structures.

Instances	VNS		w.o. NEN		w.o. all NENs		w.o. GEEN		w.o. GSON	
Name	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00
kroa150	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00
d198	**7044.0**	0.00	**7044.0**	0.00	7044.6	2.28	7044.3	1.64	**7044.0**	0.00
krob200	**11244.0**	0.00	**11244.0**	0.00	11264.0	22.6	**11244.0**	0.00	**11244.0**	0.00
gr202	**242.0**	0.00	**242.0**	0.00	242.2	0.48	242.1	0.25	**242.0**	0.00
ts225	**62268.5**	0.51	**62268.5**	0.51	62270.7	6.35	62269.9	4.58	62280.5	16.28
pr226	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00
gil262	**942.3**	1.02	943.3	1.69	947.0	3.63	942.9	1.36	943.2	1.63
pr264	**21886.5**	1.78	21890.4	5.78	21913.0	17.1	21890.5	5.84	21890.8	5.92
pr299	**20322.6**	14.67	**20322.6**	14.79	20422.2	44.85	20330.7	21.67	20347.4	28.09
lin318	**18506.8**	11.58	18514.4	13.68	18596.1	36.9	18521.5	15.96	18511.2	9.70
rd400	**5943.6**	9.69	5981.8	23.27	6067.8	48.1	5976.3	16.74	5955.0	7.57
fl417	**7982.0**	0.00	**7982.0**	0.00	7982.3	0.47	**7982.0**	0.00	**7982.0**	0.00
gr431	**1033.0**	0.18	1033.2	0.43	1037.2	1.64	1033.1	0.25	**1033.0**	0.25
pr439	**51847.9**	40.92	51850.5	36.46	52184.0	127.55	51893.5	65.6	51849.7	39.30
pcb442	**19702.8**	52.11	19793.9	44.91	20079.2	42.39	19796.7	35.51	19729.3	50.90

Table 10: Results on Ghosh' instance sets when tweaking the ILP size of GSON.

Instances	ILP size 3 – 4		ILP size 5 – 6		ILP size 7 – 8		ILP size 3 – 8	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped Eucl 125	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00	**141.1**	0.00
	133.8	0.00	**133.8**	0.00	**133.8**	0.00	**133.8**	0.00
	141.4	0.00	**141.4**	0.00	**141.4**	0.00	**141.4**	0.00
Grouped Eucl 500	**567.4**	0.65	**567.4**	0.57	567.5	0.49	567.5	0.76
	585.3	1.01	585.0	1.32	**584.4**	1.50	584.8	1.32
	584.2	1.76	**583.7**	1.82	**583.7**	1.43	584.0	1.73
Grouped Eucl 600	**84.6**	0.00	**84.6**	0.11	84.7	0.24	**84.6**	0.11
	87.9	0.00	**87.9**	0.00	88.4	0.39	**87.9**	0.00
	88.5	0.00	**88.5**	0.00	**88.5**	0.00	**88.5**	0.00
Grouped Eucl 1280	**315.9**	1.75	**315.9**	1.91	316.9	2.59	317.7	2.36
	318.4	1.58	**318.3**	1.78	319.5	1.51	319.6	1.97
	329.9	1.68	**329.4**	1.29	330.8	1.37	329.5	1.05
Random Eucl 250	2308.6	47.04	**2300.9**	40.27	2308.0	42.85	2320.8	50.63 3
	2208.6	31.66	**2201.8**	23.30	2198.6	20.56	2212.3	33.31 1
	2055.5	34.74	2057.6	31.58	2057.2	33.67	**2050.7**	15.97
Random Eucl 400	**611.5**	5.20	615.3	10.8	658.8	36.55	687.6	49.36
	595.3	0.00	**595.3**	0.00	618.3	24.62	621.6	25.02
	587.3	0.00	**587.3**	0.00	598.7	23.75	623.1	36.00
Random Eucl 600	**443.5**	0.00	**443.5**	0.00	657.7	0.00	657.7	0.00
	530.5	7.53	537.0	10.2	579.7	42.90	562.2	20.79
	466.8	0.00	469.0	11.9	560.7	63.25	551.9	65.76
Non-Eucl 200	**71.6**	0.00	**71.6**	0.00	**71.6**	0.00	**71.6**	0.00
	41.0	0.00	**41.0**	0.00	**41.0**	0.00	**41.0**	0.00
	52.8	0.00	**52.8**	0.00	**52.8**	0.00	**52.8**	0.00
Non-Eucl 500	153.2	2.82	152.5	3.69	**152.1**	4.19	153.9	3.64
	149.1	6.54	148.6	4.27	**148.4**	6.28	148.5	3.84
	167.6	3.29	**166.1**	2.89	167.5	2.81	166.2	2.82
Non-Eucl 600	16.0	1.97	**15.6**	1.62	16.1	1.65	16.2	1.97
	16.7	1.42	**16.1**	1.24	16.3	1.42	17.0	1.66
	15.2	0.51	16.0	1.66	16.1	1.78	15.8	1.46

Table 11: Results on TSPlib instances when tweaking the ILP size of GSON.

Instances	ILP size 3 – 4		ILP size 5 – 6		ILP size 7 – 8		ILP size 3 – 8	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00	**329.0**	0.00
kroa150	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00	**9815.0**	0.00
d198	**7044.0**	0.00	**7044.0**	0.00	**7044.0**	0.00	**7044.0**	0.00
krob200	**11244.0**	0.00	**11244.0**	0.00	**11244.0**	0.00	**11244.0**	0.00
gr202	**242.0**	0.00	**242.0**	0.00	**242.0**	0.18	**242.0**	0.00
ts225	**62268.2**	0.38	62268.5	0.51	62269.4	4.68	62268.3	0.47
pr226	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00	**55515.0**	0.00
gil262	942.1	0.57	942.3	1.02	**942.0**	0.00	942.1	0.57
pr264	**21886.2**	1.28	21886.5	1.78	21887.1	3.09	21887.6	4.23
pr299	20320.9	12.98	20322.6	14.67	**20316.1**	0.55	20316.8	2.07
lin318	**18504.4**	7.25	18506.8	11.58	18508.0	8.99	18506.0	8.01
rd400	5947.8	10.25	**5943.6**	9.69	5943.9	9.99	5945.8	10.57
fl417	**7982.0**	0.00	**7982.0**	0.00	**7982.0**	0.00	**7982.0**	0.00
gr431	**1033.0**	0.00	**1033.0**	0.18	**1033.0**	0.00	**1033.0**	0.18
pr439	**51837.0**	31.71	51847.9	40.92	51847.4	43.77	51841.4	41.27
pcb442	**19693.1**	49.05	19702.8	52.11	19712.4	53.58	19699.6	53.78

Table 12: Results on Ghosh' instance sets when using different starting solutions.

Instances	MDH and IKH		random init	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
Grouped Eucl 125	**141.1**	0.00	**141.1**	0.00
	133.8	0.00	**133.8**	0.00
	141.4	0.00	**141.4**	0.00
Grouped Eucl 500	**567.4**	0.57	577.2	3.85
	585.0	1.32	585.4	2.63
	583.7	1.82	585.0	3.51
Grouped Eucl 600	**84.6**	0.11	84.7	0.25
	87.9	0.00	88.0	0.16
	88.5	0.00	**88.5**	0.06
Grouped Eucl 1280	**315.9**	1.91	320.7	3.7
	318.3	1.78	320.9	4.48
	329.4	1.29	333.2	2.58
Random Eucl 250	**2300.9**	40.27	2363.3	40.33
	2201.8	23.30	2306.1	48.63
	2057.6	31.58	2153.1	92.65
Random Eucl 400	**615.3**	10.8	626.9	17.72
	595.3	0.00	**595.3**	0.00
	587.3	0.00	**587.3**	0.00
Random Eucl 600	**443.5**	0.00	**443.5**	0.00
	537.0	10.2	541.4	14.06
	469.0	11.9	470.7	12.36
Non-Eucl 200	**71.6**	0.00	**71.6**	0.00
	41.0	0.00	**41.0**	0.00
	52.8	0.00	**52.8**	0.00
Non-Eucl 500	**152.5**	3.69	177.3	16.54
	148.6	4.27	166.0	9.06
	166.1	2.89	187.5	10.11
Non-Eucl 600	**15.6**	1.62	17.8	2.93
	16.1	1.24	18.4	1.81
	16.0	1.66	16.7	2.34

Table 13: Results on TSPlib instances when using different starting solutions.

Instances	MDH and IKH		random init	
Set	$\overline{C(T)}$	std dev	$\overline{C(T)}$	std dev
gr137	**329.0**	0.00	**329.0**	0.00
kroa150	**9815.0**	0.00	**9815.0**	0.00
d198	**7044.0**	0.00	**7044.0**	0.00
krob200	**11244.0**	0.00	11246.0	6.32
gr202	**242.0**	0.00	**242.0**	0.00
ts225	**62268.5**	0.51	62268.8	0.42
pr226	**55515.0**	0.00	**55515.0**	0.00
gil262	**942.3**	1.02	943.3	1.49
pr264	**21886.5**	1.78	21890.0	5.54
pr299	**20322.6**	14.67	20341.6	27.26
lin318	**18506.8**	11.58	18517.2	14.57
rd400	**5943.6**	9.69	5969.0	26.52
fl417	**7982.0**	0.00	**7982.0**	0.00
gr431	**1033.0**	0.18	1034.4	1.84
pr439	**51847.9**	40.92	51855.3	63.74
pcb442	**19702.8**	52.11	19735.0	56.57