A Computational Investigation on Heuristic Algorithms for 2-Edge-Connectivity Augmentation

Jørgen Bang-Jensen Marco Chiarandini Peter Morling

Deptartment of Mathematics and Computer Science

University of Southern Denmark

Campusvej 55, DK-5230 Odense M – Denmark

{jbj,marco,morling}@imada.sdu.dk

Abstract

We consider the 2-edge-connectivity augmentation problem: given a graph $S = (V, E)$ which is not 2-edge-connected and a set of new edges $E' \subseteq V \times V$ with non-negative weights, find a minimum cost subset X of E' such that adding the edges of X to S results in a 2-edge-connected graph. A practical application is the extension of an existing telecommunication network to become robust against single link failures.

We compare, experimentally, different algorithms for solving general and large-scale instances. This includes exact methods based on mathematical programming, simple construction heuristics and metaheuristics. As part of the design of heuristics, we consider different neighborhood structures for local search, among which a very large scale neighborhood. In all cases, we exploit approaches through the graph formulation as well as through an equivalent set covering formulation. The results indicate that exact solutions by means of a basic integer programming model can be obtained in reasonably short time even on networks with 800 vertices and around 287.000 edges. Alternatively, an advanced heuristic algorithm based on subgradient optimization and iterated greedy finds often the optimal solution and is very fast. All previous benchmark instances are easily solved to optimality and new, larger, instances are introduced and studied.

Keywords: Edge-Connectivity, Augmentation, Survivable Networks, Local Search, Heuristics, Set Covering, Very Large Scale Neighborhood.

1 Introduction

In telecommunication networks it is often important to ensure that communication can continue after one or a few link failures. One way to do this is to compute network topologies that provide protection against damages and outages on the connections [27]. Typically there are at least two scenarios of significant practical relevance:

(a) We are given a collection of nodes (switches, base stations, routers, etc.) and possible connections between them (electrical cables, optical fibers) and we want to construct a network so that the sum of the costs associated with the connections is minimized and some given requirements for the number of paths using different connections between every pair of nodes are satisfied. Formally, this leads to the definition of the *Network Design Problem with Edge-Connectivity Requirements* [27, 33].

(b) We are given an already existing network made of nodes and connections and we want to add to it a cheapest set of new connections from a prescribed set of possible ones so that the resulting network obtained with the additional connections satisfies the requirement of survivability against possible link failures. Formally, this leads to the definition of the *Edge-Connectivity Augmentation Problem* (ECAP) on graphs [11, 18, 22].

The network design and augmentation problems are closely related. In particular, every network design problem (as described in (a)) can be seen as an augmentation problem where we start from a graph on a vertex set and no edges. There is a large body of literature providing theoretical results on both network design and augmentation problems (see [27, 33] and the references therein and [11, 18, 22]).

In the case of uniform costs, i.e., all edge weights are the same, the ECAP has the *successive augmentation property* [38]. This says that one can optimally augment a given input graph G from being k-edge-connected to being $(k+\delta)$-edge-connected by adding a set of new edges $F = \cup_{i=1}^{\delta} F_i$ with the property that adding the edges of $\cup_{i=1}^{r} F_i$, $r \leq \delta$ to G gives an optimal augmentation of G to edge-connectivity $k + r$. It is well known that this property does not generalize to the case of non-uniform weights. For an example, see Figure 1.

In this paper we restrict to the study of augmentation problems. When every connection uv is feasible, multiple copies are available of each new edge and all new edges have the same cost, there is a polynomial algorithm for finding a cheapest augmentation that will guarantee even prescribed local edge-connectivities for every pair of distinct vertices in the graph [22, 41]. Yet, the problem becomes NP-hard already for uniform costs and uniform connectivity requirements if multiple copies of an edge are not allowed [6]. In this case there is still a polynomial time algorithm when the edge-connectivity requirement is uniform and not part of the input [6]. However, for practical problems, where not all

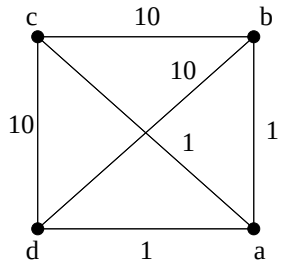

Figure 1: An example showing that the consecutive augmentation property does not hold in the weighted version of the edge-connectivity augmentation problem. The starting graph is $G = (\{a, b, c, d\}, \emptyset)$. The unique optimal augmentation of G to a connected graph is to take the edges ab, ac, ad and there is no optimal augmentation of G to a 2-edge-connected graph which contains all of these edges.

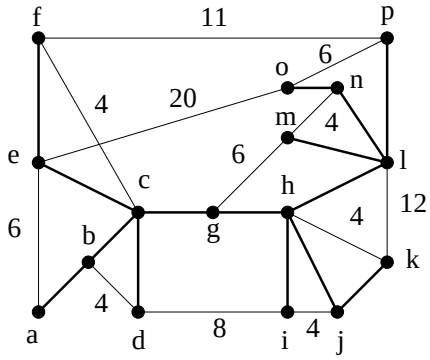

Figure 2: An instance of E1-2AUG consisting of a graph G with a spanning tree T shown with bold edges. All edges not in T have costs as indicated

connections are possible and/or costs are not uniform, the edge-connectivity augmentation problem is hard.

We focus on the simplest edge-connectivity problem which is NP-hard, that is, augmenting a given graph to a 2-edge-connected supergraph on the same vertex set. We restrict our attention to the case where the starting graph is already connected. Formally, we are given an undirected, 2-edge connected graph $G = (V, E)$, a spanning connected subgraph $S = (V, F)$, and weights on the edges of $E' = E \setminus F$. The goal is to find a minimum weight subset X of E' so that $A = (V, F \cup X)$ is 2-edge-connected. We denote this problem E1-2AUG.[1] An example of E1-2AUG is given in Figure 2. The E1-2AUG problem is NP-hard [23] and it was shown [14] that it is NP-hard even in the case in which S is a spanning tree of G, E' forms a cycle on the leaves of S and all edges in E' have the same cost.

A few 2-approximation algorithms for E1-2AUG have been found [23, 28, 29]. To our knowledge 2 is still the best approximation guarantee for non-uniform edge costs. In the case of uniform costs the current best approximation guarantee is 3/2 [19]. There are also a number of papers dealing with polynomial instances of E1-2AUG [16, 18].

[1] In the literature, this problem is also know under the abbreviated name E2AUG [39].

Only few computational studies exist in the literature. Zhu et al. [28, 46] introduce an instance generator for the E1-2AUG problem where S is a tree, and report results of a heuristic algorithm for this case. The heuristic is an enhancement of the algorithm in [23] with the same worst case approximation factor of 2 but with better performance in practice. First, it constructs a directed graph G_D from G and S. This is done by selecting a vertex r as root and directing all edges in F toward r while adding one or two directed arcs for each edge in E'. Then, a 2-edge-connected augmentation X is derived by the arcs of an out-branching from r, that is, a connected subgraph of G_D in which each vertex has a directed path from the root r and an in-degree of 1. In order to select the branching that gives rise to the cheapest augmentation, Zhu [46] uses a procedure that iteratively reduces to zero the weight of some arcs, determined by an importance measure, until the minimum weight out-branching from r in G_D has cost zero. This construction heuristic is then iterated by choosing all different vertices in V as root vertex. The heuristic is tested on graphs of up to 50 vertices.

Ljubic and Raidl proposed an evolutionary algorithm [31], and repeatedly improved it [32, 39]. In [39], the algorithm maintains a population of candidate augmentations which are repeatedly recombined and mutated. The recombination performs simply the union of two augmentations while the mutation removes an edge and introduces a new one such that the 2-edge connectivity is maintained. In addition, a so called stochastic local improvement is applied after each of these operations to remove randomly redundant edges from the augmentations while maintaining 2-edge connectivity. Although very simplistic in its components, the evolutionary algorithm in [39] is shown to perform much better than the heuristic by Zhu. Ljubic and Raidl extend the Zhu's generator and present results on much larger graphs, up to 400 vertices and about 20.000 edges. Later, Xhafa [44] applies a general Memetic Algorithm framework to the problem. The instantiated method is very similar to the Evolutionary Algorithm by Ljubic and Raidl. It uses the same stochastic local improvement and recombination component while it substitutes the mutation operator with a similar local search. An iteration of this local search consists in changing an edge of the augmentation by another not in the current augmentation such that a cost improvement is attained and the 2-edge-connectivity maintained. The experimental tests conducted on the same instances of [39] show much worse results, probably due to a less involved implementation.

In this article, we present a thorough computational study of integer programming models, construction heuristics, local search and highly effective hybrid heuristics. Over all, we investigate two different approaches to the problem, based on the graph formulation and a set covering formulation. Both these two approaches inspired new algorithms for the E1-2AUG problem.

The integer programming model is a basic set covering model. From the set covering formulation we derive also a construction heuristic and a local search algorithm based on the famous greedy covering algorithm by Chvátal [15]. Other construction heuristics and local search procedures are based on the graph representation. In particular, we devise a novel way to induce 2-edge-connectivity between two

vertices based on shortest path calculations. In the local search this gives rise to a very large scale neighborhood that can be searched efficiently.

The experiments conducted on construction heuristics and local search indicate that the set covering approach is preferable. Consequently, we develop an efficient hybrid heuristic for solving the set covering encoding of the E1-2AUG problem, drawing inspiration from existing set covering solvers [12, 13, 30, 34, 45]. It is based on Lagrangian relaxation and subgradient optimization and uses as sub-components the best construction heuristic and local search found in our study combined within an iterated greedy framework. We called this final algorithm Lagrangian Multi-Start (LMS) heuristic.

The computational results show that the basic integer programming model can be solved relatively fast by SCIP, the ILP solver of the the ZIBopt suite, for most types of large instances. This leads us to solve easily to optimality all previous test instances proposed by Ljubic et al. [39]. We report the results on these instances in the appendix. We then enlarge the set of test instances by generating new larger ones and by considering a larger gamma of structured graphs. We finally focus on four types of graphs of sizes up to 800 vertices and around 287.000 edges. The performances of the algorithms on these instances differ and allow us to detect differences among the algorithms studied. The use of well established statistical methodologies [35] to analyze the results of all the experiments in this paper guarantee the reliability and reproducibility of our conclusions. We characterize the growth in computation time with respect to instance size for all the solvers in our study and show that for some types of instances the LMS heuristic finds often the optimal solution and has a much slower growth than the exact solver.

The paper is organized as follows. Section 2 introduces the notation, formalizes the problem and gives basic algorithms. Section 3 describes the test instances and Section 4 describes the experiments with the integer programming model. Section 5 treats the construction heuristics and Section 6 the local search algorithms. The LMS heuristic is described and analyzed in Section 7. Finally, Section 8 resumes the conclusions arose in this work.

2 Definitions, Basic Algorithms and Set Covering Formulation

We use standard graph theory notation [5] and introduce few connectivity definitions. Let $G = (V, E)$ be a connected graph. An *edge-cut* in G is a minimal (w.r.t. inclusion) set of edges $C \subseteq E$ so that removing the edges of C from E disconnects the resulting graph. The size of an edge-cut C is $|C|$. With an edge-cut C we can associate a partition of V into two sets W and $V \setminus W$ so that every edge in C has one end in W and the other in $V \setminus W$. Thus deleting C from G will separate every vertex of W from every vertex of $V \setminus W$. A graph is *k-edge-connected* if it has no edge-cuts of size smaller than k. An edge-cut of size k is also called a k-edge-cut and a 1-edge-cut is also called a *bridge*. Hence a graph is 2-edge-connected if and only if it is connected and has no bridges.

In an instance of the E1-2AUG problem, we are given an undirected 2-edge-connected graph $G = (V, E)$, a fixed spanning connected subgraph of G, $S(G) = (V, F)$, and a non-negative cost function ω on $E' = E \setminus F$ (or equivalently on E with all edges in F of cost zero). The task is finding a subset X of E' of minimal cost so that $A(G) = (V, F \cup X)$ is 2-edge-connected.[2]

By an *augmentation* we always mean a subset $X \subseteq E'$, that is a set of edges not in F. We call an augmentation X *proper* if $S + X$ is 2-edge-connected. Given X we can check whether it is a proper augmentation in $O(|V| + |X|)$ time via an application of the algorithm by Tarjan [40] to find bridges in $A = (V, F \cup X)$, that is, if no bridge is found X is a proper augmentation. The graph G does not contain loops but may contain parallel edges and, in particular, some edges in E' may be parallel to an edge in F.

2.1 Reductions

In the general E1-2AUG problem it is possible to reduce G to G' by reducing the subgraph S and the set of edges available for augmentation. The reduction is such that the value of an optimal solution for G' is equal to the value of an optimal solution for G and an optimal solution in G can be derived easily from an optimal solution in G'. Like in [39], we consider the three following reduction rules.

2.1.1 Reducing S to a tree

The E1-2AUG problem can be reduced to the case in which the starting graph S is a spanning tree [23]. This follows from the fact that every spanning tree of S contains all bridges of S and thus it is sufficient to consider any spanning tree T of S to find a minimum cost set of edges whose addition to T results in a 2-edge-connected graph. Indeed, an edge in E' that lies inside a 2-edge-connected component of S will never be chosen and the component can be contracted in a single vertex while the edges can be removed from E'. Similarly, an edge in E' joining two 2-edge-connected components of S can only belong to an optimal solution to E1-2AUG if it is the cheapest of all such edges between the two components [23]. Finding connected components can be done in $O(|V| + |F|)$ additional time after knowing the bridges in S [40]. An example of application of this rule is given in Figure 3, in the passage from the left most to the central graph.

2.1.2 Edge elimination

When S is a tree T there is, for each edge e in E' with end vertices u and v, a unique (u, v)-path $P(e)$ in T, whereby we say that e *covers* the edges in the (uv)-path in T. Now, if for two edges $e, e' \in E'$ we have $P(e) \subseteq P(e')$ and $c(e) \geq c(e')$, then e is redundant and it can be removed from E'. See, for example, the edges e_6 and e_7 in Figure 3, center.

[2]For reasons of lighter notation, in the following, the dependency of S and A from G will be left implicit. Moreover, we will sometime abuse set notation by writing $W = S + X$ instead of $W = (V, F \cup X)$.

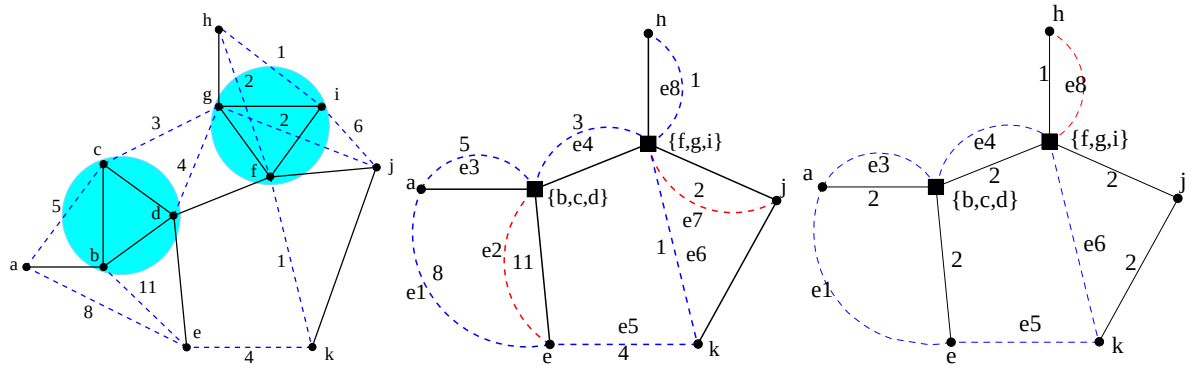

Figure 3: An example of reduction in an instance of the E1-2AUG problem with dotted edges belonging to E'. In the left figure, 2-edge-connected components are emphasized and contracted resulting in the middle graph. In the middle figure, the edges e_2 and e_7 are redundant and can be removed. In the right figure, the edge e_8 can be fixed. The resulting graph will have a 2-edge-connected component and hence it can be further contracted.

A dynamic programming procedure to identify redundant edges that runs in $O(|V|^2)$ is described in [23]. It computes a "distance" function that allows to determine, through an edge-naming function, the most convenient edge in E' to cover each pair of vertices in V. The algorithm first sorts all pairs of vertices of V in non-decreasing order of the number of edges that lie on the path between them in T. Then, in this sorted order, it updates the distance function and consequently the edge-naming function.

2.1.3 Edge fixing

If an edge uv in F is covered by a unique edge in E' then that edge will be chosen in every proper 2-edge-connected augmentation. It is therefore possible to "fix" such edges by moving them from E' to F. See example in Figure 3, right. Finding all edges to fix can be easily done in $O(|E'||V|)$ by counting the number of times an edge in F is covered and keeping track of the edges that cover it.

Clearly, the fixing process generates new 2-edge-connected components. Hence, the three reduction rules can be applied in sequence and if one rule is successful, further reductions may become possible for the other rules. The iteration ends when no rule is able to reduce further the graph. In this case, we are left with a reduced graph $G = (V, E)$ and a spanning tree $T(G) = (V, F)$ with no parallel edges in E'. From now on, we only consider the case of the E1-2AUG problem on spanning trees and no parallel edges in E'. Note that there might still be parallel edges in $G' = (V, E)$, that is, edges in E' may be parallel to edges in F (see Figure 3 right most part).

2.2 Trimming an augmentation

Every optimal edge augmentation X is minimal, that is, no further edge can be deleted without creating a bridge in the graph. If a given augmentation is not minimal it can be made so by means of a *trimming* procedure.

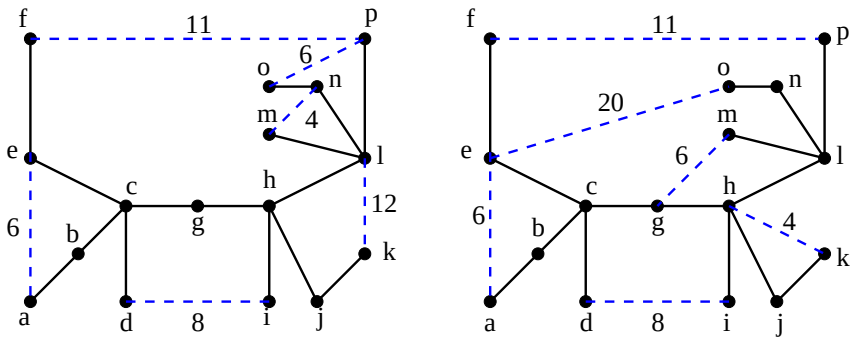

Figure 4: The effect of the trimming heuristic may depend on the ordering of the edges. Given the instance of Figure 2, the graph on the left shows the result of applying trim$(T,\emptyset,E')$ heuristic and trimming edges in lexicographic order $(ae, bd, cf, di, eo, fp, gm, hk, ij, lk, mn, op)$. The resulting augmentation has cost 47. The graph on the right shows the result when the order is reversed. The cost is 55.

Let X and Y be disjoint sets of edges not in T so that $X \cup Y$ is a proper augmentation. The procedure of trimming Y in $T + X + Y$ (denoted trim(T,X,Y)[3]) replaces Y by $Y' \subseteq Y$, so that $T + X + Y'$ is 2-edge-connected but $T + X + Z$ has a bridge for every proper subset Z of Y'. The procedure runs in $O(|Y|(|V|+|F|+|X|+|Y|))$ using Tarjan's algorithm for finding bridges [40] on each edge in Y. Observe that trim is in fact a heuristic since we have the freedom to consider the edges of Y in many different orders and the outcome of trimming in different orders may vary greatly (e.g., see Figure 4). In this paper, we always trim edges in non-increasing order of weights. In [39, 44] the order is random and the procedure is called stochastic local search.

2.3 Polynomially solvable cases of E1-2AUG

The following section describes two cases where the E1-2AUG problem is polynomially solvable. The reason for treating these cases here is that the solution algorithms will be used in the heuristic devised later. Other polynomially solvable cases can be retrieved in [24].

2.3.1 Augmenting from the complete graph at uniform cost

If $G = K_n$, i.e., the original graph is complete, and the weights on the edges are uniform then there exists a linear time algorithm based on depth-first search (DFS) to solve the E1-2AUG problem [18].[4] This algorithm, called pair(), is reported in Figure 5 and an application example is given in Figure 6.

2.3.2 Augmenting a path to a 2-edge-connected graph

A polynomial algorithm for E1-2AUG exists also in the case when the tree T to be augmented is actually a path with end-vertices x and y. This follows from a more general result on special classes of set covering

[3]The reason for distinguishing two sets of edges X and Y in E' will become clear in Section 5.1.3.

[4]In this case, weights do not have any influence and the problem corresponds to finding a minimum cardinality set of edges from K_n whose addition to a spanning tree T results in a 2-edge-connected graph. The answer is always $\lceil \frac{k}{2} \rceil$ where k is the number of leaves in T.

1 **Function** pair(T) % T is a tree with k leaves
2 Fix a leaf u of T and perform a DFS from u labeling the leaves $u_1, u_2, \ldots, u_k$ as they are encountered in the search (thus yielding an ordering of the leaves);
3 Let $X = \{u_i u_{i+\lfloor \frac{k}{2} \rfloor} : 1 \leq i \leq \lceil \frac{k}{2} \rceil\}$;
4 **return** X.

Figure 5: Optimum 2-edge-connectivity augmentation of a tree in the case of uniform edge-weights and all connections possible.

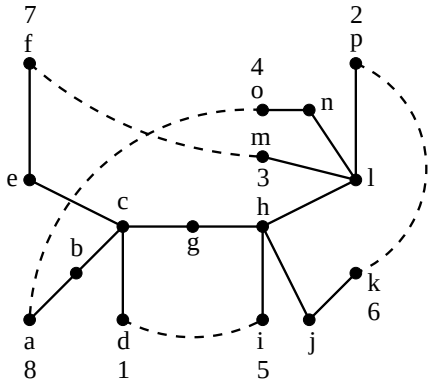

Figure 6: The spanning tree T in bold with four new dotted edges which form an optimum augmentation of T to a 2-edge-connected graph when all connections are possible. The numbers on the leaves show the numbering obtained via DFS by labeling a leaf v when it is reached (that is, leafs are labeled in preorder). The DFS is started at d and visited the leafs in the order d, p, m, o, i, k, f, a.

problems as we shall see in Section 2.4 but there is also a simple direct algorithm derived from a shortest path calculation in an auxiliary graph.

Order the vertices of T as $x = x_1, x_2, \ldots x_n = y$ corresponding to the order in which they appear on the path from x to y. Thus any edge $x_i x_j \in E'$ covers exactly the subpath $x_i x_{i+1} \ldots x_j$ of T. Form a digraph D by orienting T as a directed path from x_n to x_1 and orienting all edges of E' in the opposite direction. That is, an edge $x_i x_j$ in E' with $i < j$ is oriented from x_i to x_j. Finally, assign to every arc of the kind $x_i \to x_{i-1}$ cost zero and give every arc $x_i \to x_j$ the same cost as the edge $x_i x_j$. In this setting, every proper, minimal augmentation X corresponds to a directed path from x_1 to x_n in D whose cost is equal to the cost of X, and conversely. Thus, we can find the optimum augmentation by finding a shortest (x_1, x_n)-path in D. A similar dynamic programming algorithm of same complexity is given in [24].

2.4 E1-2AUG as a set covering problem

The *s*et covering problem consists formally in the following. Given a ground set Q and a collection $\mathcal{F}$ of subsets of Q, with each $I \in \mathcal{F}$ associated to a non-negative real-valued weight $\omega(I)$; find a minimum cost sub-collection $\mathcal{F}'$ of $\mathcal{F}$ so that $\bigcup_{I \in \mathcal{F}'} I = Q$. This problem is one of the most important problems in discrete optimization due to a wide range of real-life applications that can be modeled in this way.

The E1-2AUG problem is a special case of the general set covering problem. This claim is not new

(see [14, 16]) but we show it here since it plays an important role in the rest of the paper. Let us remind that an edge $uv \in E'$ covers the edge $f \in F$ if f is one of the edges on the unique (u, v)-path in T. Given $G = (V, E' \cup F)$, $T = (V, F)$, and a weight function ω on E', we define a matrix M, which we call the *cover matrix* of T. The rows and columns of M are indexed by the edges of $F = \{f_1, f_2, \ldots, f_{n-1}\}$ and $E' = \{e_1, e_2, \ldots, e_m\}$ where $n = |V|$ and $m = |E'|$, respectively, and for each entry (f_i, e_j), with $e_j = uv$, we have

$$M_{ij} = \begin{cases} 1 & \text{if } e_j \text{ covers } f_i \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Adding a set of edges $X \subseteq E'$ to T will result in a 2-edge-connected graph precisely when the edges in X cover every edge of T at least once. Now it is easy to see that E1-2AUG is equivalent to the following instance of the set covering problem. Let $Q = F$ and for each edge $e \in E'$ let $I_e \subseteq F$ be those edges in T that are covered by e. Furthermore, let the cost of I_e equal the cost $\omega(e)$ of e in G. Using the cover matrix the E1-2AUG problem can be expressed as the following integer program:

$$z_{SCP} = \min\{\omega^T x : Mx \geq 1, x \in \{0, 1\}^m\} \tag{2}$$

which corresponds to a set covering integer programming formulation where $x(e) = 1$ if and only if the set I_e is part of the solution.

Conforti et al. [16] used this set covering formulation to derive a number of polynomially solvable instances of E1-2AUG. Note that for the degenerate case treated in Section 2.3.2, where T is a path, the set covering formulation of E1-2AUG, via the cover matrix M, has the so-called consecutive ones property for the columns.[5] In this case M is a totally unimodular matrix and hence (2) can be solved in polynomial time via linear programming [43, Section 3.2].

3 Problem Sets

In the previous computational studies [28, 31] on instances of the E1-2AUG problem, the instance generator by Zhu was used to create randomly generated graphs with different characteristics. Ljubic et al. [31] enhanced the generator to produce larger instances. We report data and results of our algorithms on Ljubic's instances in the appendix. In our study we use newly generated instances.[6] The choice to use new instances is motivated by two facts: even the largest instances in [31] were found easy to solve; the organization of these instances does not permit a systematic study on the feature of the instances and their influence on the results.

[5] A matrix has the consecutive ones property for the columns if it is possible to permute the rows so that for every column all entries with value 1 occur consecutively.

[6] All the new instances are publically available at `www.imada.sdu.dk/~marco/e12aug`.

The new instances consist of random graphs with the following structure.

Type `T+C`: The graph G is a *tree and cycle* graph obtained by the composition of a tree T and a cycle on the leaves of the tree T. These graphs are constructed by a generator that adds vertices of the tree to the l leaves until all leaves are connected to a tree vertex and the maximum vertex degree does not exceed a given value D. The graphs are determined by the two parameters l and D. Integer weights are assigned on the leaf cycle uniformly at random from the interval $[1, l]$. The E1-2AUG remains NP-hard on `T+C` instances even for uniform weights [14].

Type `Unif`: The graph G is a *uniform*, random graph of size n obtained by selecting each of the $\binom{n}{2}$ possible edges independently at random with probability p. Integer weights are chosen uniformly at random from the interval $(1, 10000)$. The spanning tree $T(G)$ is one randomly chosen among those of minimal cost.

Type `Euc`: The graph G is a *geometric* graph of size n generated from points in a two dimensional grid with integer coordinates in $[1, 10000)$. Edges are added between any pair of vertices if their Euclidean distance, rounded to the closest integer, is less than an integer parameter f. Weights on the edges correspond to the Euclidean distance between their end points. The spanning tree $T(G)$ is one randomly chosen among those of minimal cost.

Type `SmallWorld`: The graph G is obtained by a random rewiring procedure: starting from a ring lattice with n nodes and d edges per vertex (and no multiple edges), each edge is rewired at random with probability p. This procedure allows to tune the graphs between regularity ($p = 0$) and disorder ($p = 1$). The characteristics of these graphs are quantified in [42]. It is observed that there is a wide interval of p over which the graph remains highly clustered while the typical separation between two vertices in the graph (or, more formally, the diameter of the graph) is small. This is due to the immediate drop in the separation between vertices caused by the introduction of a few long-range edges that create short-cuts connecting vertices otherwise far apart. This particular "small-world" topology of the network is relevant for modeling power supply grids and dynamic systems such as social networks and spread of infectious diseases [42]. Integer weights on the edges are chosen uniformly at random from the interval $(1, 10000)$ and the spanning tree $T(G)$ is one randomly chosen.

On the instances `T+C` we consider l equal to $\{200, 400, 800\}$ and D equal to $\{2, 3, \ldots, 30\}$ and $\{39, 49, \ldots, l-1\}$ and generate one instance for each combination. For a fixed l the number of vertices tends to be high for very small values of D, thus yielding sparse graphs, while it tends to stabilize to values slightly larger than l for values of D above 20.

On `Unif`, `Euc` and `SmallWorld` instances we consider 3 sizes of graphs: 200, 400, 800. The parameter f on the `Euc` instances determines the edge density, computed as $\rho(G) = |E'|/\binom{|V|}{2}$. As an approximation,

$f = \{2000, 5100, 15000\}$ yield an edge density similar to the one produced by $p = \{0.1, 0.5, 0.9\}$ on the graphs of type `Unif`. We use these values for the parameters of `Unif` and `Euc` and indicate them in both cases by {L,M,H}, meaning low, medium and high, respectively. Similarly, on the `SmallWorld` instances we choose d equal to $\{0.1, 0.5, 0.9\}$ times the number of vertices. For the rewiring probability we consider three values: $\{0, 0.05, 1\}$, the first produces regular graphs, the second small world graphs and the last one disordered graphs. Each combination of the instance parameters gives rise to an instance classes for which we generate 10 instances for types `Unif` and `Euc` and 5 for `SmallWorld`. See the Table 5 in Appendix 8 for descriptive statistics of these instances before and after a preprocessing phase obtained by the application of the reductions rules described in Section 2.1.

In the rest of the paper we always assume that instances are preprocessed before being solved by any algorithm for the E1-2AUG problem. Moreover, all computational experiments are conducted on machines Intel Core 2 CPU 6300 at 1.86GHz, with 2 GB RAM, running Ubuntu 8.04.

4 Exact Solution

In order to transform the E1-2AUG in a set covering problem we need to determine the cuts of the graph, that is, the sets I_e for each e in T. We can do this in $O(|V| + |E|)$ time using the Tarjan algorithm for finding bridges [40]. The algorithm is modified in such a way that for each vertex w visited in postorder the edges that join descendants of w with non-descendants of w are recorded as covering the bridge incident to w. The computation takes negligible time on all instances, never exceeding 2.7 seconds on the largest `Euc` and `Unif` instance or 6.5 seconds on the largest `SmallWorld` instances.

The set covering integer program (2) can be solved by encoding it in one of the existing softwares to solve these standard problems. We have chosen the non-commercial mixed integer programming solver SCIP 1.0 with SoPlex 1.3.2[7] and run it on all our test instance with a time limit of 1 hour. Beside implementing the currently fastest techniques for solving integer programs by branch-and-bound and primal-dual solvers, SCIP allows also to integrate constraint programming techniques such as constraint propagation and presolving. These techniques, try to exploit problem-specific knowledge. For the set covering model, SCIP implements more efficient domain propagation rules using the "two watched literal scheme" borrowed from the satisfiability problem. Set covering constraints can, indeed, be seen as disjunctive clauses of literals [1]. We decided to test two versions of the solver. One version, LINEAR, uses only the standard integer programming techniques and treats the constraints as linear constraints without specific knowledge. Another version, COVERING, interprets the constraints as set covering constraints and exploits specific constraint propagation rules.

All `T+C` instances turn out to be very easy to solve to optimality. The largest solution time registered was 3.7 seconds, while the median on the largest instances (l=800) was 0.18 seconds. We decided therefore

[7] Zuse Institute Berlin (ZIB). `http://scip.zib.de/`. Last updated: February 27, 2008. Downloaded: March 10, 2008.

to abandon these instances in the rest of the paper.[8]

The `Unif` and `Euc` and `SmallWorld` instances are instead more challenging. The results are visualized in Figures 7 and 8 and numerically reported in the Appendix, Table 5.

For all `Unif`, `Euc` and `SmallWorld` instances, the figures evidence that uniform weights make instances harder to solve than non-uniform weights. This might be in part due to the fact that preprocessing is more effective in the non-uniform cases (cfr. Table 6 in the Appendix).

For both the `Unif` and the `Euc` instances 3600 seconds are sufficient to solve them to optimality. The largest time registered is 2675 seconds for the LINEAR version of the solver on an `Unif` instance with 800 vertices and 0.9 edge density (which is reduced to 0.3 after preprocessing). The COVERING version has instead peak times of 2000 seconds on the same instance. The `Euc` instances are slightly easier with largest solution time of 2060 seconds for LINEAR version and 1232 seconds for the COVERING version. In the figure, we superimpose a linear regression to the data. The number of edges in E' after preprocessing seems enough to explain a linear trend of growth in computational cost to solve these instances. Since the plot is in logarithmic scale we can deduce, empirically, a polynomial growth rate. More precisely the linear regression is $t = 10^a \cdot |E'|^a$ where t is time in seconds and a and b are determined by the least squares method and range from 3.3 to 3.8 and from 0.34 to 0.5, respectively.

For the instances `SmallWorld` 3600 seconds are not always sufficient to solve to optimality. In particular, regularity ($p = 0$) and small world topology ($p = 0.05$) seem to make these graphs harder to solve. For these cases, and with size 800 and edge density 0.9 the solver did not succeed either in finishing the presolving phase and hence it did not to produce any primal or dual bound (see details on these cases later in Section 7.2). The number of unsolved instances is clearly smaller for the COVERING version than for the LINEAR version of the solver.

The `SmallWorld` instances motivate the study of heuristics that can be used for solving the problem directly or to speed up an integer programming solver. Moreover, heuristics can still be the choice in very fast applications or with even larger instances.

5 Construction Heuristics

Construction heuristics incrementally build an augmentation by adding edges according to what appears to be the "best" immediate choice. They are the fastest methods to generate good quality solutions. Furthermore, they can be used to produce starting solutions for local search and metaheuristic methods. The algorithms by [23, 29, 46] are construction heuristics for the E1-2AUG problem with a known worst case guarantee of 2. In Ljubic et al. [31] the initial augmentation is the whole set E' followed

[8] In addition to the graphs discussed in the text, we tried other kind of structured graphs of similar sizes. Among the graphs tried, there are Delaunay triangulation graphs, planar graphs, rectangular graphs, triangular graphs, Waxman graphs, grid and 2D and 3D torus. All the instances derived from these graphs turned out to be computationally easy to solve.

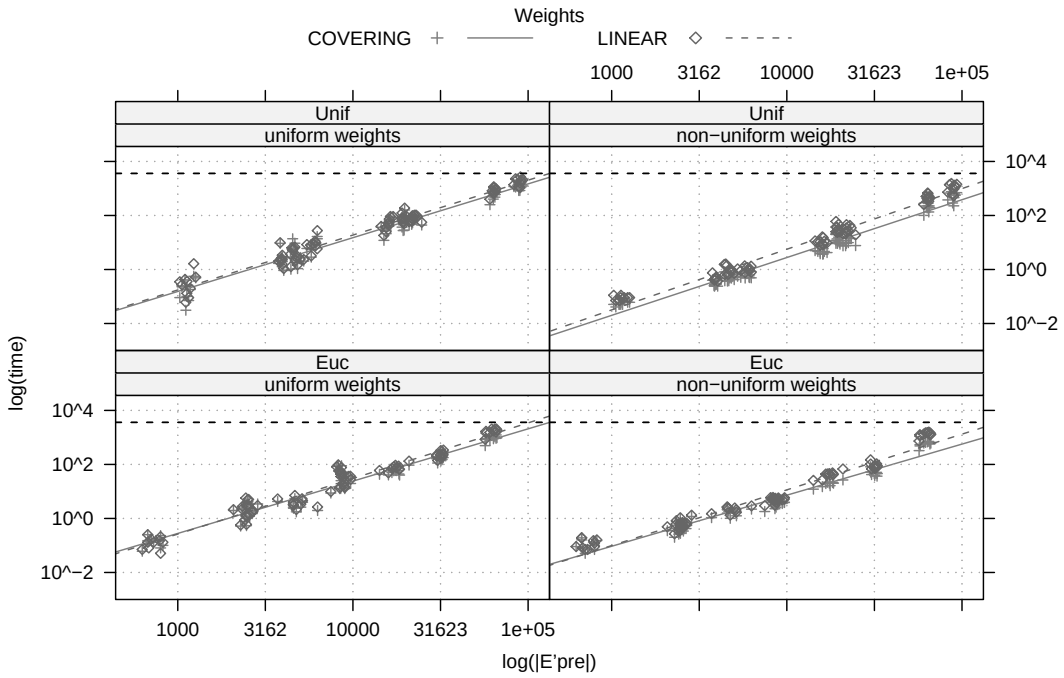

Figure 7: Computation time to solve `Unif` and `Euc` instances with SCIP 1.0 SoPlex 1.3.2. A logarithmic transformation is applied on both the x- and y-axis and a linear regression line is superimposed. The instances are preprocessed and on the x-axis we report the number of edges left in E'. Both x- and y-axis are in logarithmic scale. A line superimposed indicates the linear regression. See Tables 5 and 6 in the Appendix for details on these graphs.

by an application of their stochastic local improvement (our trimming). In this section, we study new construction heuristics for the E1-2AUG problem.

5.1 The algorithms

We consider 3 different approaches. All the algorithms in this section build a proper augmentation by a one single pass rule.

5.1.1 Random and lightest edge addition

A basic rule to construct a proper augmentation X is given in Figure 9. Iteratively an edge from E' is considered for insertion in X. If it covers at least one previously uncovered edge of F then it is inserted in X otherwise it is discarded and not considered anymore. The critical step in the procedure is the selection of the edge to consider next in this sequence (line 5). We consider two alternatives: random and smallest weight first. This gives rise to two algorithms that we call random_add and lightest_add. The edge selection procedure can lead to an augmentation which is not minimal and hence the trimming procedure defined in Section 2 is applied on line 10.

The set of edges in E'' on line 3 and 5 could be reduced to only those that make a spanning forest of the graph $G' = (V, E')$, that is, the union of a set of spanning trees, one for each connected component

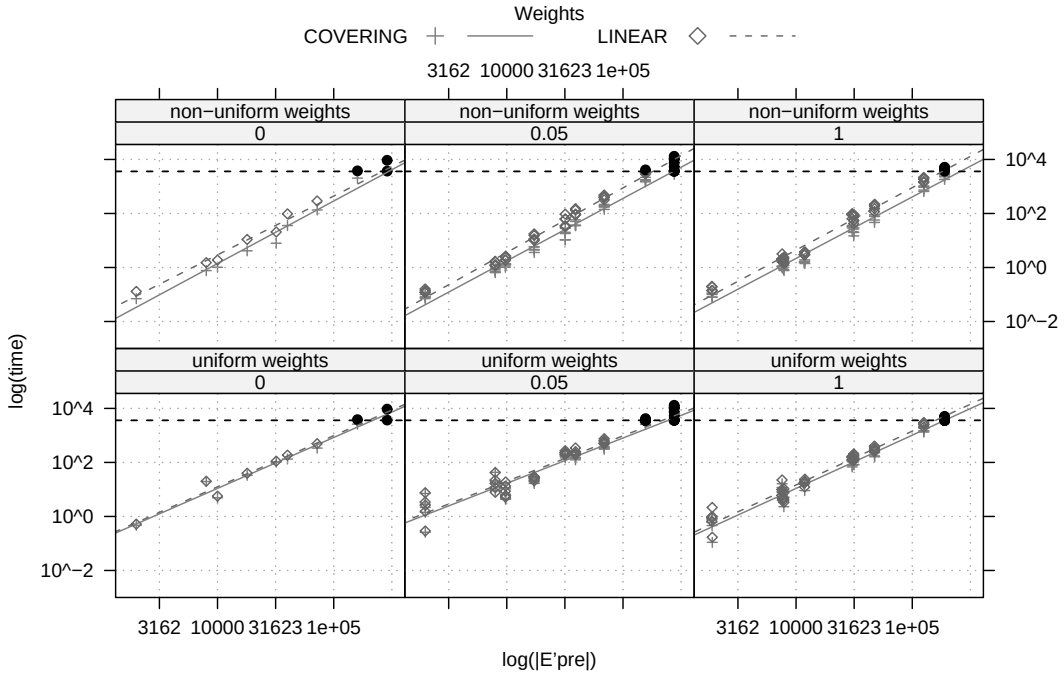

Figure 8: Computation time to solve `SmallWorld` instances with SCIP 1.0 SoPlex 1.3.2. The instances are distinguished in the columns of the matrix plot by the rewiring probability p. Note that the class of instances with $p = 0$ has less data because in this case graphs are deterministic and hence we generated only one instance. On the x-axis we report the number of edges in E' after preprocessing. Both x- and y-axis are in logarithmic scale. Filled black circles represent runs of either solvers that were truncated before reaching optimality. A linear regression, ignoring filled circles, is superimposed on the data. See Tables 5 and 6 in the Appendix for details on these graphs.

1 **Function** random_add(G, T);
2 $Z := F$ (No edge in T is covered);
3 $E'' := E'$; $X := \emptyset$;
4 **while** $Z \neq \emptyset$ **do**
5 Choose a random edge $uv \in E''$ and delete it from E'';
6 Let P_{uv} be the (edge set of) the (u, v)-path in T;
7 **if** $P_{uv} \cap Z \neq \emptyset$ **then**
8 $X := X + e$;
9 $Z := Z \setminus P_{uv}$;
10 $X' :=$ trim$(T, \emptyset, X)$;
11 **return** X'

Figure 9: Random addition heuristic. If we replace the random selection by a selection of the edge with the smallest weight on line 5 we obtain lightest_add.

of $G' = (V, E')$. This is idea is used in [37]. However, on our test instances, after trimming this choice produces always the same solutions as the lightest addition heuristic and therefore we do not consider it here.

In both versions of random and lightest insertion the procedure in Figure 9 is unlikely to produce very high quality solutions. Nevertheless, we include them both in our study to make certain that other, more complex, algorithms do, in fact, better.

5.1.2 The greedy set covering heuristic

The set covering formulation of the E1-2AUG problem allows us to apply successful greedy set covering heuristics. They work on the matrix M from Section 2.4 and iteratively add columns (i.e., edges from E') to a partial candidate solution X until all rows (i.e., edges of F) are covered. One successful such greedy algorithm is the one by Chvátal [15] reported in Figure 10.

```
1  Function greedy_cov(M, C = {I_1, I_2, ..., I_m}, ω)        % I_j ⊆ F
2  Z := F;
3  X := ∅;
4  while Z ≠ ∅ do
5  |  Select I_j : I_j ∈ C such that |I_j ∩ Z| ≠ ∅ and I_j minimizes ω(I_j)/|I_j ∩ Z|;
6  |  Z := Z \ I_j;
7  |  C := C \ {I_j};
8  |  X := X ∪ {j};
9  X :=trim(T, ∅, X);
10 return X.
```

Figure 10: Greedy set covering heuristic. See Section 2.4 for the correspondence between set covering notation and E1-2AUG notation.

The set Z contains, at each stage, the remaining uncovered rows and the set X is the cover under construction. The greedy step is on line 5 where the column j chosen to be added is the one with the smallest cost relative to the number of uncovered rows that it covers. This computation is bounded above by $O(|F||E'|)$. Note that the covering attained with the loop on lines 4-8 corresponds to a proper augmentation which might not be minimal, as each time a column is added some previously added columns may become redundant. Hence on line 9 we remove redundant columns by the trimming procedure of Section 2.2.[9] The loop on lines is repeated $O(\min(|F|, |E'|))$ times.

It can be shown (e.g., [17, Section 35.3]) that greedy_cov is an $O(\ln|V| + 1)$-approximation algorithm for the general set covering problem and that, on the E1-2AUG problem, this algorithm does not improve the best known approximation guarantee of 2. Figure 11 shows an example with uniform costs where the approximation is, indeed, a factor of 2 from optimum.[10]

5.1.3 The shortest path heuristic

The function pair() described in Figure 5 finds a pairing of leaves (with one repeated vertex if the number of leaves is odd) so that adding a new edge between each of these pairs will give a proper augmentation. This augmentation is optimal, if the graph is complete and costs are uniform. However, if these conditions do not hold, an edge uv determined by pair() might not be available in E' or there might exist a cheaper way than the one determined by pair() to add a set of edges Y from E' so that, after adding Y to T, u

[9] Calling the function trim() in Figure 10, as well as in other parts of the paper, we leave implicit the correspondence between arguments in graph notation and arguments in set covering notation.

[10] The example is due to Matthias Kriesell. Private communication.

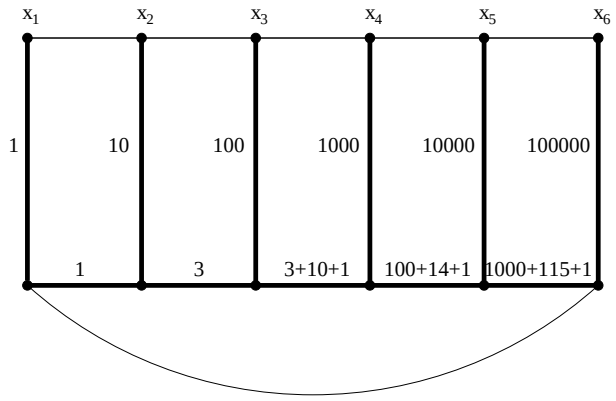

Figure 11: An example of a 2-approximation of greedy_cov on the E1-2AUG with uniform costs. T is indicated by edges in bold. The weights of edges in E' are uniform and not indicated in the picture. The numbers on the edges of T represent the number of edges in the corresponding subpath of T. These numbers have been chosen so that greedy_cov is forced to take all the edges of E' except the bottom one whereas the optimal solution is made by the edge on the bottom plus every second edge on the top starting from the right most x_5x_6. In the specific example greedy_cov will use 5 edges while the optimum is 4. It is easy to generalize this example so that the ratio converges to 2 from below. Denoting with i the ith vertical edge going from left to right and calling $i = 1$ the left most edge, one has to ensure that for $i \geq 2$ the number of edges on the ith subpath of the horizontal part of T is larger than the sum of the edges in the $(i-1)$st horizontal subpath plus the vertical part just before the $(i-1)$st horizontal part. In this way, greedy_cov will include all the top edges starting with the right most one, $x_{n-1}x_n$.

and v will be in the same 2-edge-connected component. When trying to find such a possible set Y it is reasonable to substitute the edge uv by a shortest path between u and v in a certain digraph (see details below). The resulting algorithm is given in Figure 12 and it is illustrated in Figure 13.

The procedure works as follows. First lines 1-4 determine the vertex pairs, arbitrarily ordered, and initialize the data. Then, lines 5-11 go through the list of pairs and determine a shortest path between each pair in an auxiliary digraph. For each pair i and current augmentation X, the digraph D_i is constructed in such a way that, searching for a path from u_i to v_i in G, edges from $T + X$ can be used for free, edges from P_{uv} that are currently 1-edge-cuts in $T + X$ may only be used in one direction, and all other edges from E' may be used in both directions with costs equal to their original cost. In an implementation it is sufficient to create the digraph D of G on line 6 only once, storing it in a matrix and updating it when edges are covered. The shortest path can be computed by Dijkstra's algorithm [17] with Fibonacci heap as priority queue and takes $O(|E| + |V| \ln |V|)$. The proof that shortest_path is correct is given in the following proposition.

Proposition 1. *Let X be the set of edges returned by* shortest_path *in Figure 12. The graph $T + X$ is 2-edge-connected.*

Proof: For each $i \in \{1, 2, \dots p\}$ let P_i be the unique path in T between u_i and v_i and let D_i, $i = 1, 2, \dots, p$ be as in Figure 12. Observe that in D_1 every (u_1, v_1)-path must use an edge not in T to cross each of the directed 1-edge-cuts corresponding to the edges in P_1. Hence after the first iteration all edges on P_1 are covered by X (see also the argument in Section 2.3.2). In the ith iteration, $i \geq 2$, all edges of P_i that are still not covered by X will be directed 1-edge-cuts in D_i and hence every (u_i, v_i)-path in D_i will

```
1  Function shortest_path(G,T)
2  Let {u_1v_1, u_2v_2, ..., u_pv_p} be the set of connections returned by pair(T) ;
3  X := ∅ ;
4  Z := F;
5  for i = 1 to p do
6  |  Construct a digraph D_i from G by directing every edge of P_{u_iv_i} ∩ Z towards u_i and replacing
   |  every other edge in G by a directed 2-cycle;
7  |  Let P be a shortest path from u_i to v_i in D_i where the cost of every edge is as in G except
   |  for edges in T + X which have cost zero;
8  |  Y := (E(P) ∩ E');
9  |  X := X ∪ Y;
10 |  Let C(P) be those edges in Z which are covered by Y;
11 |  Z := Z \ C(P);
12 X' := trim(T,∅,X);
13 return X'
```

Figure 12: Shortest path heuristic. Recall that by P_{uv} we denote the unique path between u and v in T.

use an edge not in $T + X$ to cross this cut and after adding these new edges to X every edge of P_i is covered. Now the claim follows easily by induction on i. Note that trimming will not destroy a proper augmentation so that the final X is also a proper augmentation. □

The quality of the solution of the shortest path heuristic may depend on the order in which the pairs are examined in the loop of lines 5-11 of shortest_path. We studied different orders: (a) the pairs are sorted according to the length of a shortest (u_i, v_i)-path in the digraph D_i computed independently from the other pairs;[11] (b) random order; (c) pairs with $u_iv_i \in E'$ sorted in non-decreasing order of weight of the corresponding edges u_iv_i and pairs with $u_iv_i \notin E'$ added at the end of the order; (d) same as (c) but pairs with $u_iv_i \notin E'$ added at the beginning of the order; (e) pairs with $u_iv_i \in E'$ sorted in non-increasing order of weight of the corresponding edges u_iv_i and pairs with $u_iv_i \notin E'$ added at the end of the order.

5.2 Experimental analysis

We compare the different heuristics by running them once on a set of instances equally sampled from the classes of `Unif` and `Euc` instances with non-uniform weights and `SmallWorld` instances with uniform and non-uniform weights. The results are transformed in percentage deviation from optimum (these values were computed in Section 4).

First we analyze the effect of the order of pairs in the shortest_path heuristic. An analysis of variance (ANOVA) [35] revealed no significant effect of the instance features in the results. On the other hand, the aggregate analysis over all instance classes unveils that the alternatives (a) (c) and (d) form a group significantly better than the alternatives (b) and (e). Since the alternative (d) result in faster executions we choose this one for use in the rest of the paper.

The second analysis compares the shortest_path heuristic with the random_add, lightest_add and greedy_cov

[11] This corresponds to treat every pair as if it were the first one in the algorithm of Figure 12.

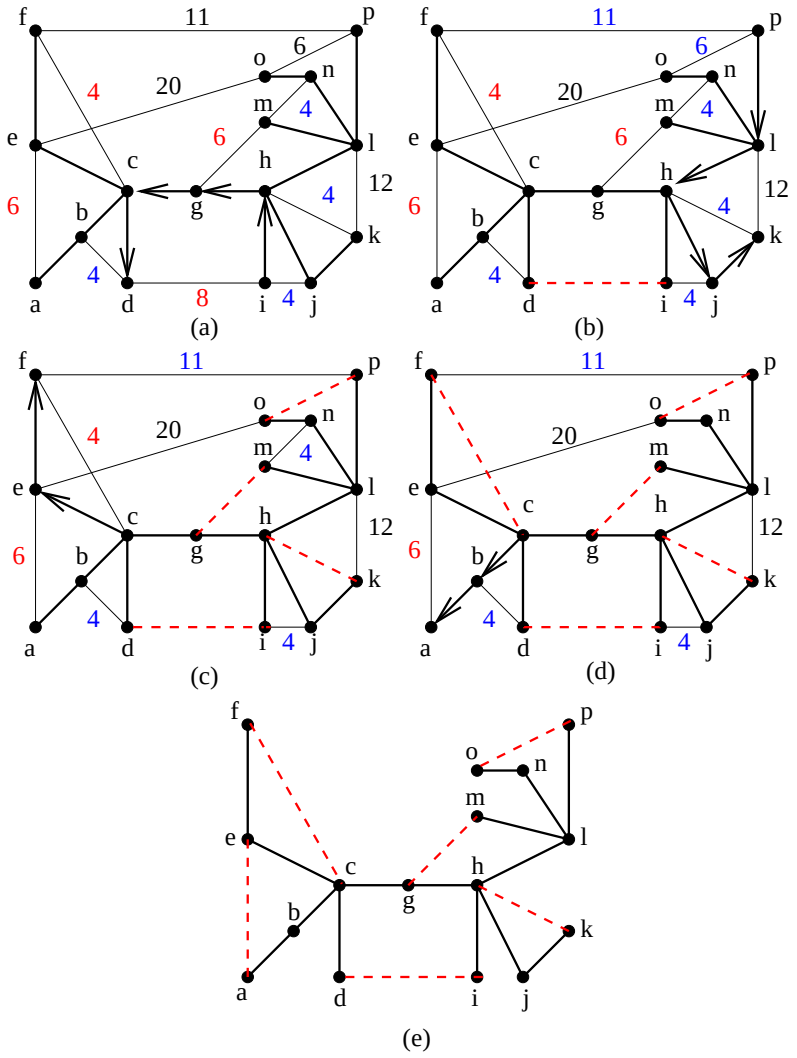

Figure 13: An illustration of the shortest path construction heuristic. The starting graph is represented in (a) with edges in T in bold. The pairs of leaves are as in Figure 6: di, kp, fm, ao. In Figure (a) the goal is to find a shortest path from d to i in the digraph shown (all edges that are not directed represent directed 2-cycles and, hence can be traversed in both directions at the same cost as the original edge). Edges with no cost listed have cost zero. The shortest (d, i)-path consists of the edge di of cost 8. In Figure (b) the goal is finding the shortest path between k and p, hence we direct towards k those edges on the path $kjhlp$ which are still bridges after the addition of di. The shortest (k, p)-path uses the new edges kh, gm and op of total cost 16. In Figure (c) the goal is finding the shortest path between m and f in the digraph obtained by orienting towards f those edges on the path $mlhgcef$ which are not covered by $\{di, gm, kh, op\}$. The shortest (f, m)-path in this digraph uses the new edge cf at cost 4. In Figure (d) we seek a shortest (a, o)-path which contains only one new edge ae at cost 6. Figure (e) shows the final augmentation (trimming does not remove any edge) whose cost 34 is the optimal solution to this instance of E1-2AUG.

heuristics. A preliminary analysis reveals that random_add attains very bad results in all classes of instances. Since it might generate outliers that bias the analysis we remove it from the analysis. An ANOVA analysis on the instance factors[12] type, size and density and the algorithmic factor indicates as significant the effect of the three heuristics, the type of instance and the interaction of these two factors. Since there is no influence of the other two factors in the percentage deviation from the optimal solution

[12]For compliance with statistical studies we call *factors* the instance features and the algorithmic parameters and levels the values they take.

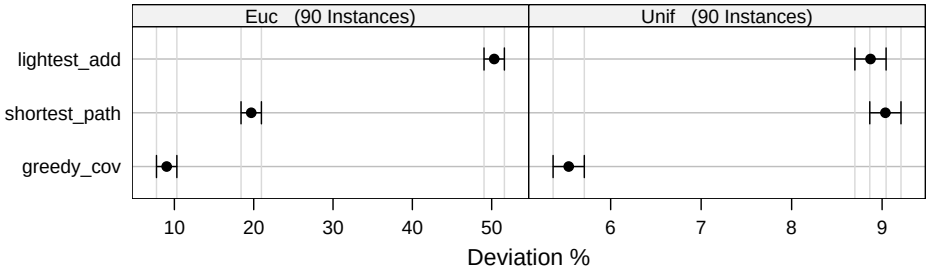

Figure 14: Multiple pairwise comparisons for the heuristics on the `Unif` and `Euc` instances. The confidence intervals are derived by the Tukey Honest Significant Differences method.

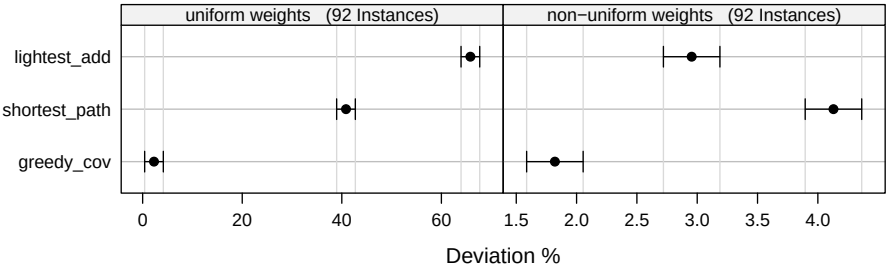

Figure 15: Multiple pairwise comparisons on the `SmallWorld` instances.

of the heuristics we can aggregate the analysis within a certain instance type. In Figure 14 we report the 95% confidence intervals derived by the Tukey Honest Significant Differences (THSD) [35] for the `Unif` and `Euc` instances. In the plots, two heuristics can be claimed significantly different if the corresponding intervals do not overlap. The greedy_cov heuristic is consistently the best heuristic and produces solutions which are on average less than 10% and 5.5% away from optimal solutions on instances of type `Euc` and `Unif`, respectively. The influence of the instance type can be seen in the two plots of Figure 14 by noting that the percentage error is generally larger on the `Euc` instances and that there are differences in the comparison between shortest_path and lightest_add. In Figure 15 we repeat the same analysis on the `SmallWorld` instances distinguishing the uniform from the non-uniform cases. The relative order of the heuristics is confirmed in both cases, furthermore greedy_cov produces a better approximation, attaining an average deviation of 2.3% on the uniform case and 1.8% on the non-uniform one.

An analysis of the computation time for the three heuristics, not reported here, confirms a polynomial growth for all the three heuristics with the greedy_cov growing faster than the other two. On the largest instances it took 25-30 seconds, though our implementation was not too involved and probably better performances could be obtained.

In the Appendix (Table 3 and 4) we report the numerical results of greedy_cov on the Ljubic's instances.

6 Local Search

Solutions returned by construction heuristics can often be improved by local search. A basic version of local search is iterative improvement [25]. In this section, we design and study experimentally three new iterative improvement algorithms for the E1-2AUG problem.

Basic local search may then be enhanced by metaheuristics, which are general guidance criteria to lead the search beyond local optima [25]. This will be the topic of Section 7.

6.1 The algorithms

The main point in the description of iterative improvement algorithms is the definition of a neighborhood structure that associates each candidate solution with a set of other solutions. This is done through a neighborhood operator that performs small changes to an incumbent solution. At each step an improving neighbor of the incumbent solution is chosen and the search is continued until no improving neighbor can be found, in which case the incumbent solution is called local optimum. We describe three neighborhood operators and the algorithms they give rise.

6.1.1 Addition neighborhood

The addition neighborhood is inspired by the local optimization for set covering described in [34]. We describe it using the set covering terminology in which edges in E' are the columns and edges in F are the rows of the matrix M. A candidate solution X is defined by a list of selected columns from E' such that X is a proper cover, i.e., all rows are covered. The evaluation of a candidate solution is given by the sum of the weights of the selected columns.

Let $cov(i, X)$ indicate the number of times the row i is covered by columns in X. We call a column j in $E' \setminus X$ *candidate superior* if $X \cup \{j\}$ contains one or more columns other than j, say $R = j'_1, \ldots, j'_l$, $l \geq 1$ (in [34] the authors use $l \geq 2$), such that for each j' in R and for each $i \in I_{j'}$ we have $cov(i, X \cup \{j\}) \geq 2$ and the sum of their weights is greater than the weight of j, i.e., $\sum_{h=1,\ldots,l} w_{j'_h} > w_j$. The k-addition operator adds k candidate superior columns, $A = \{j_1, \ldots, j_k\}$ and removes the columns other than A that become redundant in $X \cup A$. Note that the condition to be candidate superior is only necessary and not sufficient for the columns in R to be in fact all redundant in $X \cup A$. The local search procedure, which we call k-add, is sketched in Figure 16.

The set L of candidate superior columns on line 4 is a list sorted in non increasing order of gains, i.e., $\sum_{h=1,\ldots,l} w_{j'_h} - w_j$. Then, the first k best columns are added to the solution and redundant columns are removed (lines 8 and 9) with the trimming function. If this operation yields an improvement, the incumbent solution X is updated (line 11). The search ends when the list L has been entirely scanned without any improvement in the solution. Note that the operations on line 9 might make some edges in L not anymore superior, hence the loop on lines 6-13 might end with L not empty. Moreover, the solution

```
1  Function k-add(M,X);
2  improved := TRUE;
3  while improved do
4  |  Let L ⊆ E' \ X be the set of candidate superior edges;
5  |  improved := FALSE;
6  |  while at least k superior columns in L \ X do
7  |  |  X' = X;
8  |  |  Let A be the set of the k candidate superior columns in L \ X of maximal gains;
9  |  |  X* := A∪ trim(T, A, X) ;
10 |  |  if w(X*) < w(X) then
11 |  |  |  X := X*;
12 |  |  |  improved := TRUE;
13 |  |  L := L/A;
14 return X
```

Figure 16: k-addition iterative improvement.

X returned by k-add might not be a local optimum, as not all combinations of k candidate superior edges are tried. Trying all possible combinations would make the procedure computationally too expensive for a local search and we prefer trying to improve further the solution by a metaheuristic.

In the implementation of the algorithm, we maintain updated the values $cov(i, X)$ in order to speed up computations on line 4 and on line 9. Furthermore, in the function trim(), if the number of edges to trim is smaller than 5, we systematically enumerate all possible ordered subsets of candidates for removal and choose the one whose order leads to the proper augmentation of minimal cost. Line 4 takes $O(|E'|^2|F|)$ and updating $cov(i, X)$ and trimming (if in random order) can be done in $O(|F|)$. The loop 6-13 is repeated $O(|E'|)$ times.

6.1.2 Destruct-reconstruct neighborhood

Another neighborhood inspired by the set covering formulation is defined by the destruct-reconstruct operator. It consists in first removing k edges from X and then reconstructing the partial covering by means of the greedy set covering heuristic described in Section 5.1.2.

The local search procedure is given in Figure 17. A list L of randomly ordered columns is maintained and the set R of k columns to remove is chosen scanning this list (line 9). After removal the greedy construction heuristic is applied to attain a proper covering on lines 10-16 and the trim() function to remove redundant columns on line 17. This procedure, that we recall from Section 5.1.2 takes $O(|F||E'|\min(|F|, |E'|))$, though in practice much less here, is repeated when no improvement is found at most $O(|E'|)$ times, that is, until all edges in L are examined. Note that the reinsertion of removed columns is prohibited on line 13. However, it may happen that an edge is the only available to cover certain bridges and in this case we do allow its reinsertion. In order to speed up the search the list L is updated only after $|X|/2$ iterations if a change in X has occurred. The procedure ends when the whole list has been explored and no improvement found. Similarly to k-add the solution returned might not be

```
1  Function k-dr(M,X,w);
2  idle := 0;
3  while idle < (|X| − k) do
4      L := X randomly ordered;
5      i := 1; unchanged := TRUE;
6      while i < (|X|/2) or unchanged do
7          X′ := X;
8          Let R be the set of the next k columns in L ∩ X;
9          X′ := X′ \ R;
10         Let Z be the set of rows not covered in X′;
11         I := ∅;
12         while Z ≠ ∅ do
13             Select j : j ∈ E′ \ (X′ ∪ R) such that I_j ∩ Z ≠ ∅ and j minimizes w(j)/|I_j ∩ Z|;
14             Z := Z \ I_j;
15             X′ := X′ ∪ {j};
16             I := I ∪ {j};
17         X* :=trim(T, I, X′);
18         if w(X*) < w(X) then
19             X := X*;
20             idle := 0; unchanged := FALSE;
21         else
22             idle := idle + 1;
23         i := i + 1;
```

Figure 17: k-destruct-reconstruct iterative improvement. We recall that I_j is the set of rows covered by column j.

a local optimum, as not all possible combinations of k columns are tried.

6.1.3 The shortest path neighborhood

The third neighborhood operator is derived by the shortest path heuristic of Section 5.1.3.

The local search procedure is given in Figure 18. The procedure follows a similar scheme as k-dr. A list L of edges in random order is maintained and the next k edges from this list are in turn removed from X (line 9). Then, for $k > 1$, an extension of pair(), that copes with 2-edge-connected components and runs in $O(|V| + |E|)$ time, is used to find a combination of leaves or vertices arbitrarily chosen from the 2-edge-connected components (line 10). The loop on lines 13-19 implements the reconstruction of the augmentation by means of shortest paths and has running time dominated by k times $O(|E| + |V| \ln |V|)$ if the directed graph is maintained and needs only to be updated. Finally, redundant edges different from those inserted are trimmed on line 20. The update of the list L occurs only after $|X|/2$ iterations if improvements happened in the augmentation. The whole procedure is repeated at most $O(|E|)$ times when no improvement is found.

The neighborhood examined in the k-sp iterative improvement is a very large scale neighborhood according to the definition of Ahuja et al. in [2]. In particular, it implements a shortest path based search of an exponential neighborhood and hence it falls in the category of neighborhood search by means of

```
1  Function k-sp(T,X);
2  idle := 0;
3  while idle < (|X| − k) do
4      L := X randomly ordered;
5      i := 1; unchanged := FALSE;
6      while i < (|X|/2) or unchanged do
7          X′ := X;
8          Let R = {u1v1, u2v2, ..., ukvk} be the set of the next k edges in L ∩ X;
9          X′ = X′ \ R;
10         Let {u′1v′1, u′2v′2, ..., u′lv′l} be the set of connections returned by pair(T + X′) ;
11         Let Z be the edges in T not covered by an edge in X′;
12         Y := ∅;
13         for j = 1 to l do
14             Construct a digraph Di from G by directing every edge of P_{u′j v′j} ∩ Z towards u′j and
               replacing every other edge in G by a directed 2-cycle;
15             Let P be a shortest path from u′j to v′j in Dj where the cost of every edge is as in G
               except for edges in T + X which have cost zero;
16             Y′ := (E(P) ∩ E′);
17             X′ := X′ ∪ Y′; Y := Y ∪ Y′;
18             Let C(P) be the edges in Z which are covered by P;
19             Z := Z \ C(P);
20         X*:=trim(T,Y,X′);
21         if w(X*) < w(X) then
22             X := X*;
23             idle := 0; unchanged := FALSE;
24         else
25             idle := idle + 1;
26         i := i + 1;
```

Figure 18: k-shortest-path iterative improvement.

network flows algorithms defined in that survey. In Figure 19 we show an example in which 1-sp finds solutions better than it would be possible with 1-add or 1-dr.

6.2 Experimental analysis

We compare the three local search procedures on instances of type `Euc` and `Unif` with non-uniform weights. It is frequently conjectured that there is an interaction between the starting solution and the local search that follows, hence the construction heuristic that produces the best starting solution might not necessarily be the best one to combine with local search. Therefore we consider all the three construction heuristics of Section 5. Another element of inquiry is the value to assign to k and whether there are differences in the instance classes. Due to these considerations, we design an experiment with 3 treatment factors, the starting solution, the neighborhood type and the value of k, and 3 grouping factors, type, size and density of the instance. All factors have three levels except instance type that has two. In specific, for k we test values in $\{1, 3, 5\}$.

Each combination of treatment factors gives rise to an algorithmic configuration which we run on one single instance within each of the 18 instance classes generated by the combination of the grouping

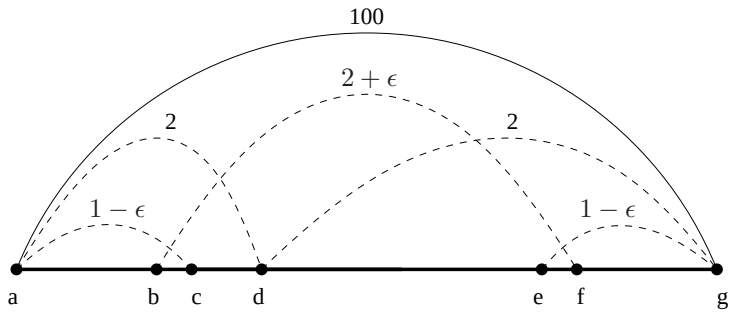

Figure 19: An example showing that the shortest path neighborhood can find better solutions than the other two neighborhoods. In the figure, bold edges belong to T, the edge ag to the current augmentation X and the dotted edges to $E' \setminus X$. The number on the edges of $E' \setminus X$ are the weights, with $\epsilon > 0$. Moreover, we assume that the subpaths ab, bc, cd, de, ef and fg in T contain $2k, 1, k, 4k, 1, 2k$ edges, respectively, with k any integer number larger than zero. In the addition neighborhood there is no candidate superior edge and hence the augmentation X cannot be improved. In the greedy covering neighborhood, if we remove from X the edge ag, then we add the edges dg, ac and ad and trim the edge ac, thus yielding a total cost of 4. In the shortest path neighborhood, removing the same edge ag, we add the edges ac, bf and eg, thus reaching the optimal solution with cost $4 - \epsilon$.

factors. Given the high number of configurations one single run is enough to detect main effects and low order interactions [35]. Each result is then transformed in percentage deviation from the optimal solution on the instance.

We analyze the data by means of ANOVA using a nested-mixed effects model [4, 35] which gives all the algorithmic main effects and second order interactions of these factors as significant. Among the grouping factors only the instance type results significant. We therefore split the analysis in the two instance types.

In order to obtain a synthetic view of the results with the 3 treatment factors we make use of regression trees [7, 10, 26]. Regression trees are decision trees on the factors of the design; binary splits occur if effects are statistically significant and the importance of the factor can be derived from the level where the split occurs in the tree. Each node reports the predicted value of the response variable for the selected configuration (i.e., the combination of factor levels that labels the path from the root to that node). In Figure 20, for the two instance types, we report the tree-structured regression model produced from our data by the non-parametric regression tree method developed by Hothorn et al. [26].[13] This method implements a conditional inference tree by binary recursive partitioning in a well defined theory of conditional inference procedures. In the figure at each node, following the right path leads to the better performing combination; that is, the right-most path corresponds to the best choice.

The analysis indicates that it is better to start from the best possible solution (greedy_cov) and that the k-add neighborhood is the best. No difference is, instead, detected between the three values of k although a closer look revealed that k-add works best with $k = 1$ while k-dr with $k = 5$ and k-sp with $k = 3$, thus indicating a different behavior with respect to this factor.

By comparing the predicted average values in Figure 20 with the average values of greedy_cov in Figure

[13]The method is available from the package party in R, the free software environment for statistical computing and graphics.

14 it is evident that the local search contributes to improve the solution. By further analysis we could conclude that the best algorithmic configurations, (i.e., greedy_cov, k-add neighborhood and $k = \{1, 3, 5\}$) perform a median number of 25 improvements on instances of type Euc and Unif and size 800 and around 6 on instances of size 200.

We sought confirmation of these results on the SmallWorld instances. In this case we do not have the optimal solution for all instances, hence we use ranks to compare the 27 algorithmic configurations. The analysis conducted on 198 instances equally distributed among the classes of SmallWorld instances (see Section 3 and Table 5 in the Appendix) confirmed the algorithmic combination greedy_cov, k-add neighborhood and $k = 1$ as the best one. The number of improvements attained by this configuration on these type of instances varies from a median of 2-3 improvements up to a maximum of 17-20 without a notable difference between instances with uniform and non-uniform weights.

The k-add local search is preferable also with respect to computation time. In Figure 21 we represent in logarithmic scale the computation time needed by local search to reach a local optimum. For each neighborhood type, data from different k and initial solution are aggregated and the time for generating an initial solution is removed. In general, local search is not guaranteed to finish in polynomial time but our empirical results seem to hint at a polynomial growth. Most important, in all three types of instances the k-add local search scores the lowest growth curve (though, this outcome must be treated with caution, because our procedures might not be fully optimized).

6.3 Remarks

In a preliminary analysis we tested also a local search with a k-exchange neighborhood operator that exchanges exactly k edges between X and $E' \setminus X$. However, this procedure is not successful for very strongly constrained problems like the E1-2AUG because once we have removed k edges there are only few proper solutions attainable by inserting a fixed number k of edges. A possible implementation of k-exchange should therefore allow also improper augmentations. In this case the local search becomes much more complex. A successful example in this sense is the 3-flip by Yagiura et al. [45] for the set covering problem.

The shortest path neighborhood is theoretically appealing because it explores efficiently a large neighborhood. However the experimental comparison indicated that the best neighborhood is the addition neighborhood. This is true not only in terms of computation time which might be expected but also, surprisingly, in terms of solution quality. It is difficult to give an explanation of why this is the case. One choice, whose impact we did not test, is to not allow the trimming procedure to remove redundant edges added in the last iteration (lines 9, 17 and 20 of Figures 16, 17, 18, respectively). This might hinder the discovery of new improvements in the cases $k > 1$ and k-dr and k-sp, while enhancing k-add with a mechanism similar to tabu search. However, in doing this, some care must be be in order to avoid that

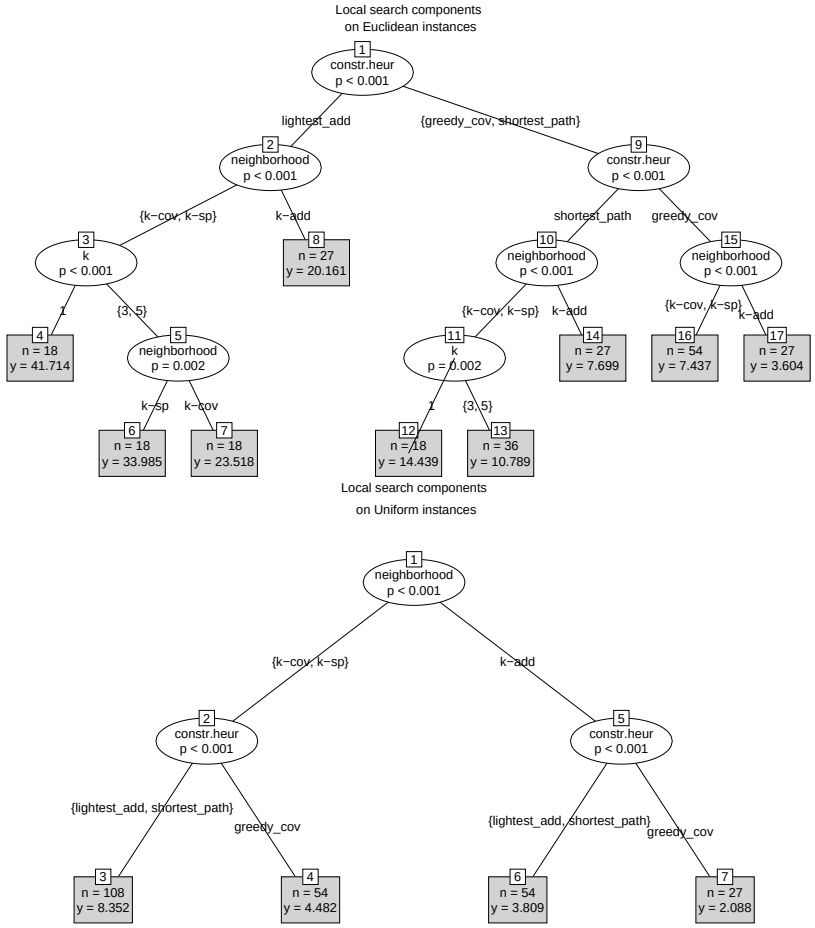

Figure 20: Regression tree analysis on local search components within the `Unif` and `Euc` instances. The factors in the analysis are: neighborhood type, initial solution and k. In the tree, the deviation from optimum is indicated by y which is the predicted value obtained by the n results that compose the leaf. A branching occurs when there is statistical significance in the differences. In this case the p value is reported.

a cycling behavior is triggered.

Even after the application of local search, however, the solution we attain is rarely optimal (around 3.6 and 2% away according to Figure 20). It is worthwhile then trying to further enhance these heuristics.

7 An Advanced Heuristic

The results of the previous two sections indicated that a set covering approach to the E1-2AUG problem leads to better results than a graph specific approach. In this section, we design a very efficient heuristic to solve set covering encoded E1-2AUG instances. In doing this we strongly rely on components of state-of-the-art set covering solvers. More specifically, we borrow ideas from the solvers by Caprara, Fischetti and Toth (1999) [12] and Marchiori and Steenbeek (2000) [34]. In particular, from the former we take the Lagrangian relaxation with subgradient optimization and the pricing scheme, which are elements widely used in set covering solvers (see [34, 30, 13, 45]). From the latter we take the idea of iterating destruction and re-construction of solutions, a procedure that is often referred to as iterated greedy. We

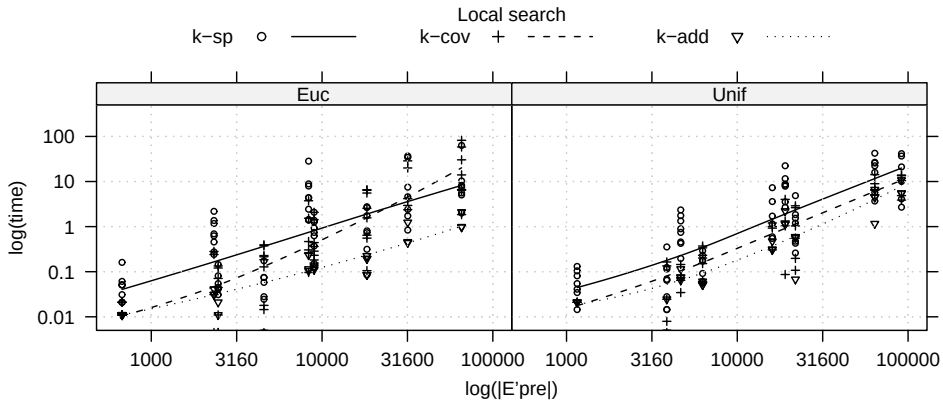

Figure 21: Log-log plots of computation time to reach a local optimum for the three local search procedures. For each neighborhood we aggregate data for different construction heuristics and different k. Each configuration is run once on a single instance. A smooth regression is superimposed to emphasize the trend.

combine these components with the local search optimization suggested from the previous section and with a perturbation mechanism in the fashion of iterated local search [25]. We call the final algorithm the *Lagrangian Multi-Start Heuristic* (LMS).

7.1 The algorithm

Crucial in the description of the algorithm is the Lagrangian relaxation of the set covering model (model 2). For a treatment of Lagrangian relaxation theory we refer to [21, 43]. Shortly, for every vector $u \in \mathbb{R}^m_+$ of Lagrangian multipliers associated with the constraints, the Lagrangian relaxation of model (2) is defined as:

$$z_{LR}(u) = \min_{x \in \{0,1\}^m} \sum_{j \in E'} c_j(u) x_j + \sum_{i \in F} u_i, \qquad x_j \in \{0,1\},\ j \in E' \tag{3}$$

where $c_j(u) = w_j - \sum_{i \in F} M_{ij} u_i$ is the Lagrangian cost associated with column $j \in E'$ and corresponds to the reduced cost of that column (we use set covering notation). The Lagrangian relaxation provides a lower bound on the optimal solution of model (2), i.e., $z_{LR}(u) \leq z_{SCP}$, $\forall u \geq 0$. An optimal solution to model (3) is given by $x_j(u) = 1$ if $c_j(u) \leq 0$ and $x_j(u) = 0$ if $c_j(u) > 0$. However, such solution might lead to covers X that are infeasible for the model (2). Hence, frequently, good feasible solutions to model (2) are searched also from near-optimal solutions to model (3).

To find the best choice u^* for the multiplier vector $u \in R^m_+$ (i.e., the vector u that provides the highest lower bound), we solve the Lagrangian dual problem, i.e., $z_{LD} = \max_{u \in R^m_+} z_{LR}(u)$. The most popular approach to solve the Lagrangian dual is the subgradient method. Given an arbitrary initial u_0 a sequence $\{u^k\}$ of multiplier vectors is generated by the rule $u_i^{k+1} = \max\{u_i^k + t^k s_i(u^k), 0\}$ with $s_i(u^k) = 1 - \sum_{j \in E'} M_{ij} x_j(u^k)$ for all $i \in F$ being the subgradient. The most commonly used scalar step

size t^k is

$$t^k = \frac{\lambda^k(UB_{LD} - z_{LD}(u^k))}{||s(u^k)||^2} \tag{4}$$

where λ^k is a scalar satisfying $0 < \lambda^k \leq 2$, UB_{LD} is an upper bound to z_{LD}, which is in practice substituted by a primal upper bound on z_{SCP}, and $x(u^k)$ is an optimal solution to (3) with $u = u^k$. The method is terminated upon reaching an arbitrary iteration limit and typically it converges quite fast to good solutions.

The Lagrangian costs may be used as indicators of the utility of selecting column j. Hence they can be used in construction heuristics, as for example in greedy_cov by replacing the original costs (precisely, on line 5 of Figure 10) to determine a *score* and, consequently, a ranking of the columns. A successful choice in the algorithm by Caprara et al. [12] is the application of a heuristic procedure at several near-optimal Lagrangian multiplier vectors: first a multiplier vector u^* is attained by an aggressive policy through the subgradient algorithm and then a neighborhood of u^* is exploited by applying a number of small perturbations to u^* followed by the greedy heuristic. A similar procedure is endorsed also by Balas and Carrera [3].

Our overall LMS algorithm is outlined in Figure 22. The algorithm consists of a first initialization phase, where pricing, subgradient and local optimization are alternated on the lines 4-9. The algorithm then enters in the iterated greedy loop (lines 11-15), in which first a partial covering is selected from the incumbent solution, then a modified greedy heuristic is used to reconstruct a feasible solution from this partial covering and, finally, a local search is applied to improve the reconstructed solution. As suggested in [34] after every 100 iterations a new core is selected and a new subgradient phase applied (lines 20-21). If no improvement occurs for 200 iterations then a perturbation seeks to introduce diversification in the search (lines 16-19).

Subgradient phase Each iteration of subgradient_phase requires (a) computing the Lagrangian costs $c_j(u^k)$ associated with the current multiplier vector u^k, (b) deriving a solution $x_j(u^k)$ for model (3) and computing the subgradient $s_i(u^k)$, (c) updating the multiplier vectors using the rule for u_i^{k+1} and Formula (4).

At step (b) we use the *reduced cost heuristic* [3] that generates a full cover X and as a byproduct it also improves the dual solution u^k used to start the heuristic. This differs from past implementations where instead an aggressive policy is adopted, for example, setting $x_j(u^k) = 1$ if $c_j(u^k) \leq 0$ and zero otherwise, thus leading to covers x that might be infeasible for the model (2) [12]. The reduced cost heuristic sets first $x_j(u^k) = 1$ if $c_j(u^k) \leq 0$ and $x_j(u^k) = 0$ otherwise. Then, for each uncovered row i it picks the column l from $\{j : M_{ij} = 1\}$ with minimal $c_j(u^k)$ and sets $u_l^k := u_l^k + c_j(u^k)$. Finally, for all columns $\{j : M_{ij} = 1\}$ it sets $c_j(u^k) = c_j(u^k) - c_l(u^k)$ and $x_j(u^k) = 1$ if $c_j(u^k) = 0$. The complexity of

```
1  Function LMS (M, ω);
2  u_i^0 = min_{j∈J_i} w_j/|I_j| for all i ∈ F;
3  M^core := pricing (M, ω, u^0);
4  X := greedy_cov (M^core, ω);
5  (X, u*) := subgradient_phase (M^core, u_0);
6  X := local_optimization (X);
7  M^core := pricing (M, ω, u*);
8  (X, u*) := subgradient_phase (M^core, ω, u*);
9  j := 0; improved := FALSE;
10 repeat
11     for i := 1 to 100 do
12         X̄ := destruction (M^core, ω, X, u*);
13         X := construction (M^core, ω, X̄, u*) ;
14         X := local_optimization (M^core, ω, X);
15         j++; update improved;
16     if not improved and j ≥ 200 then
17         X := perturbation (M^core, X);
18         X := local_optimization (M^core, ω, X);
19         j := 0; improved := FALSE;
20     M^core := pricing (M, ω, u*);
21     (X, u*) := subgradient_phase (M^core, ω, u*);
22 until time limit not exceeded ;
```

Figure 22: The LMS heuristic. Not shown in the algorithm, an overall best augmentation is maintained and updated throughout the search. This augmentation is used to determine the update of `improved` on line 9, 15 and 19.

the heuristic is a linear function of the number of nonzero entries in the matrix M. We apply the *reduced cost heuristic* at each subgradient iteration, since each u^k might as well lead to a good near-optimal solution. This is similar in spirit to the successful choice of perturbing u^* used in [12].

At every call, the subgradient algorithm stops when the improvement of the lower bound z_{LR} remains lower than 0.1% for $\gamma \cdot 100$ iterations or when a maximum number of $\gamma \cdot m$ iterations is exceeded. We set $\gamma = 10$ in the first call on line 5 and $\gamma = 5$ in all other calls. This termination condition is compliant with [45].

As in [12] the first time subgradient_phase is called u_0 is set equal to u_0 defined in line 2 while the other times it is $u_i^0 = (1 + \delta_i)u^*$ where u^* is the best Lagrangian multiplier vector found so far and δ^i is a uniform random value in $[-0.1, 0.1]$. The parameter λ^k is set initially to 0.1 as in [12]. Then, every $p = 20$ subgradient iterations the difference between the best and worst lower bound saw in the last p iterations is computed and if this value is greater than 1%, λ^k is halved, if it is lower than 0.1%, λ^k is multiplied by 1.5 (unless it becomes larger than 2), otherwise, λ^k is left unchanged.

Pricing This procedure is inspired by [12] and consists in selecting a subset of promising columns with low Lagrangian costs. The starting problem is reduced to a core problem in order to save computing time. Like in [12], we compute the Lagrangian $c_j(u^k)$, $j \in E'$, associated with the current u^k, and define the subset of columns making the core as $C = C_1 \cup C_2$ where $C_1 = \{j \in E' \ : \ c_j(u^k) < 0.1\}$, and C_2

contains the smallest Lagrangian cost columns covering each row; if $|C_1| > 5n$ we keep in C_1 only the $5n$ columns of smallest Lagrangian cost. Differently from [12], who use a dynamic parameter to deciding the pricing frequency, we recompute the core only when indicated in Figure 22.

Destruction The destruction phase is part of the iterated greedy and it is implemented by selecting a subset of columns from the current solution while leaving out others. This procedure differs from the one in [34]. A counter $count(j)$ is maintained for each column $j \in E'$ which indicates the number of times a column j has been in the solution. Then columns in the current solution are sorted in non-decreasing order according to $count(j)$. The first 30% of the columns are selected with probability 0.85, the next 30% are selected with probability 0.8 and the remaining 40% are selected with probability 1 if $c_j(u^*) < -0.001$ and with probability 0.6 otherwise. The number of the columns selected, typically, ranges from 75% to 95% of the one in the incumbent solution. The output of the destruction procedure is a partial covering.

Construction This task is accomplished by using a modification of the greedy_cov heuristic of Figure 10. It takes as input a partial covering and the current vector $c(u^*)$ of reduced costs attained from the last subgradient phase. These costs are used to compute a score for each column and implement a different selection policy with respect to the one on line 5 of Figure 10. The new procedure selects the column j that minimizes the score $\frac{c_j}{\sum_{i \in I_j \cap Z} c^i_{j^*}(u^*)}$, where $c^i_{j^*}(u^*)$ is the minimal reduced cost of a column covering row i unless this value is smaller than 0.01, i.e., $c^i_{j^*}(u^*_i) = \max\{\min\{c_j(u^*) : j \in J_i\}, 0.01\}$. The set Z contains only the rows yet uncovered. When a feasible covering is obtained we ensure it is minimal by applying the trimming procedure, enhanced, as in the case of the local search algorithms, by an optimal search by enumeration when the number of redundant columns left becomes smaller than 5.

Local optimization This corresponds to the 1-add local search described in Section 6.

Perturbation This procedure first selects a subset of columns from the current solution to remove and then reestablishes the feasibility of the solution by means of a reconstruction phase. Columns are selected, similarly to destruction, by sorting them in non-decreasing order of $count(j)$, and then choosing the first 20% of columns in the order with probability 0.8 and each remaining column j only if $c_j(u^*) < -0.001$. The feasibility is reestablished by applying the greedy_cov of Figure 10 to the partial covering. The search continues from the perturbed solution.

7.2 Experimental analysis

The competitiveness of the LMS algorithm with respect to state-of-the-art set covering solvers was tested on well known set covering benchmark instances available from the OR-library[14] maintained by

[14] J. Beasly, "OR-Library" (1990). Last update: October 2005. `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`. Visited: February 2007.

J. Beasley [8]. The detailed results of this comparison are reported in [36]. The largest random instances in this library are those identified by the letters E and F which have SCP constraint matrix of size 500×5000 with 20% and 10% of elements equal to one, and those identified by the letters G and H which have matrix of size 1000×10000 with ones density of 5% and 2%. All the instances have edge weights randomly chosen in $[1, 100]$. These instances are quite similar to our instances in the E1-2AUG (see statistics in Appendix 8, Table 5).[15] The optimal solutions are not known and the best results are mainly due to [12, 45]. On all instances E-H our LMS, with a specific tuning of parameters, finds the best known solutions while it is not the case for other famous solvers such as [9, 3, 34, 13]. In terms of computation times, LMS has peaks of 92 and 114 seconds on the G-H instances, while the CFT [12] and the 3FNLS [45], after transformation of times, are much faster and would need less than 6 seconds to produce solutions of the same quality. A part of this difference is certainly due to the fact that while the CFT and 3FLNS programs are written in FORTRAN and in C and are highly optimized, our LMS program is written in C++ and is not optimized, as it was developed under a framework that allowed all the tests reported in this article.

On the E1-2AUG problem, LMS is often faster than SCIP to find an optimal solution but it does not always succeed to find it. In Figure 23 we compare the computation time of SCIP with the one of LMS. Note that the LMS heuristic does not provide a valid lower bound, because the Lagrangian Relaxation and the Lagrangian dual problem consider only a restricted core of the overall problem, hence the typical Lagrangian dual bound does not apply overall. Moreover, we do not solve the restricted problem to optimality and hence lower bounds of the types proposed, for example, in [20] do not apply either. In the experiments we use the dual bound returned by SCIP to prove the optimality of the solutions produced by the LMS. Although this procedure might appear unfair, it nevertheless allows us to understand which is the expected approximation for the LMS heuristic and which could a reasonable time to stop it. The user can then decide whether to use the heuristic alone and believe the solution is with high chances optimal or include it in a integer programming solver that provides lower bounds. When an optimal solution is not found the solver is run until its time limit, that is 3600 seconds.

In the figure, each plot is obtained by the results on 90 instances for the `Euc` and `Unif` cases and 99 instances for the `SmallWorld`. The results vary according to instance features. In particular, the nature of weights makes a clear difference in the comparison of the two solvers. In the uniform weights case, there is no difference in the performance and hence the exact solver has to be preferred. On the contrary, with non-uniform weights, the LMS heuristic gains advantage on the exact solver as the size of the instance increases. This advantage is more pronounced on the `SmallWorld` instances than on the `Unif` and `Euc` instances. In fact, on the `Unif` and `Euc` instances the LMS does not always find the optimal solution,

[15] In particular, these instances correspond to our instances of type U. We tested whether the difference in the range of weights on the edges might be reason for differences in the time to solve to optimality (see Section 4) instances of size 200. The result of the experiment showed with statistical significance that a range [1,10000] makes instances slightly harder to solve than a range [1,100].

Instances		SCIP 3600s		SCIP unlimited			LMS 3600s		
$\|V\|$-*d*-*p*-*s*	weights	LB	gap	LB	gap	time (sec.)	sol	gap	time best (sec.)
800-720-0-1	non-unif.	$-\infty$	∞	3787	0 % (1)	4791	3787	0 %	1
800-720-0.05-2	non-unif.	$-\infty$	∞	4184	0 % (1)	6427	4184	0 %	1
800-720-0.05-3	non-unif.	$-\infty$	∞	3825	0 % (1)	5233	3825	0 %	1
800-720-0.05-4	non-unif.	$-\infty$	∞	4226	0 % (1)	3779	4226	0 %	1
800-720-0.05-5	non-unif.	$-\infty$	∞	4722	0 % (1)	4748	4722	0 %	1
800-400-0.05-1	unif.	159	1.89 %	159	0 % (647)	3857	159	0 %	2393
800-400-0.05-5	unif.	163	0.61 %	163	0 % (386)	3682	164	0.61 %	2622
800-720-0-1	unif.	$-\infty$	∞	164	0 % (882)	9693	165	0.61 %	2925
800-720-0.05-1	unif.	$-\infty$	∞	161	0 % (764)	6936	161	0 %	2366
800-720-0.05-2	unif.	$-\infty$	∞	161.50	4.02 %	8682	163	1.55 %	3418
800-720-0.05-3	unif.	$-\infty$	∞	160	0 % (644)	7820	160	0 %	2325
800-720-0.05-4	unif.	$-\infty$	∞	161	0 % (606)	6519	162	0.62 %	2527
800-720-0.05-5	unif.	$-\infty$	∞	170	0 % (679)	7941	171	0.59 %	2878
800-720-1-1	unif.	167.50	8.06 %	168	0 % (301)	4764	168	0.30 %	2320
800-720-1-2	unif.	159	0.63 %	159	0 % (286)	3942	160	0.63 %	2469
800-720-1-3	unif.	161	0.62 %	161	0 % (292)	3474	162	0.62 %	2306
800-720-1-4	unif.	166	1.20 %	166	0 % (427)	4453	167	0.60 %	2899
800-720-1-5	unif.	157	8.28 %	157	0 % (383)	4033	161	2.55 %	3068

Table 1: Results on the `SmallWorld` instances that were not solved to optimality by SCIP in 3600 seconds. We give the lower bound and the approximation gap reached by SCIP with a time limit of 3600 seconds and without time limit. A number in parenthesis next to the gap indicates the number of solutions found. For LMS we provide the best solution, the gap and the time to the best solution found with a time limit of 3600 seconds. The gap of LMS is computed on the lower bound provided by SCIP run without time limit.

as it does instead the exact solver. More precisely, the number of instances, that LMS does not solve to optimality, is 12 in the `Unif` class with uniform weights and 3 and 1 in the `Euc` class with uniform and non-uniform weights, respectively. In the `SmallWorld` class, this number is 23 for the LMS against 18 for the exact solver of Section 4.

In Table 1, we analyze in more detail the `SmallWorld` instances not solved by SCIP. We observe that, within the same time limit of 1 hour, the LMS finds better solutions. It particular, it provides solutions where SCIP produces no dual and primal bounds[16] Most remarkably, LMS finds the only solution existing on the 5 instances with non-uniform weights in one second, while SCIP requires much more than one hour. On the other instances, like, for example, `sw-720-0.05-1`, SCIP without time limit, needs 3891 seconds to find the first primal bound with a gap of 88.41% against the bound found by LMS which is 0.61 % after 2925 seconds. This gap remained preferable until 9693 seconds when SCIP finally found a solution of better quality and proved optimality. Similar results hold for the other instances while on `sw-720-0.05-2` SCIP run out of memory and crashed.

These results suggest that the use of the LMS heuristic as a primal heuristic within the mathematical programming framework might be a promising further development, that arise from our work, to solve efficiently this specific problem.

Finally, we looked closer at the run profiles of the LMS heuristic to understand whether all the

[16]In these cases, SCIP does not complete the presolving phase within the imposed time limit. We observed that the presolving phase is typically very expensive in terms of computation time and does not produce any reduction (a possible explanation for this being that the instances are already preprocessed, in much more fast way, exploiting problem specific knowledge). As an example, if presolving is removed, the 5 instances in Table 1 can be solved within less than 700 seconds. In this article we decided to treat the ILP solver as a black-box solver and use it with its default parameters. The reader must keep in mind, however, that a proper tuning of parameters is likely to produce considerable improvements.

Type	weights	# of instances	line 5	line 6	line 7	lines 20-22
`Unif`	unif.	90	0	13	1	76
`Unif`	non-unif.	90	14	64	3	9
`Euc`	unif.	90	2	39	0	49
`Euc`	non-unif.	90	1	32	0	57
`SmallWorld`	unif.	99	0	14	1	84
`SmallWorld`	non-unif.	99	78	16	0	5

Table 2: The table reports the lines of the algorithm in Figure 22 where the LMS ended, that is, where it found the last best solution. Numbers refer to runs with one run performed on each instance of the class.

components of this algorithm are necessary. In Table 2, we report which line of the algorithm in Figure 22 determined the last improvement. The number of times the algorithm entered in the loop on lines 10-22, representing the iterated greedy phase, is considerable, hinting at the positive contribution of this part of the algorithm. However, on the instances `SmallWorld` with non-uniform weights, that phase does not seem to be important. Indeed, the 5 times in which the algorithm entered in the loop correspond to the instances reported in Table 1 where no improvement is detected after 1 second of run time.

8 Conclusions

We studied the problem of augmenting a spanning tree to a 2-edge-connected graph by using a set of available connections. In particular, we considered large size instances with no restriction on the density of the edges and the distribution of weights.

We solved with SCIP 1.0 the integer programs derived from the equivalent set covering formulation of the problem and showed that all the benchmark instances existing in the literature are easy to solve. We provided the optimal solutions, previously unknown, on all these instances in an appendix of this work. We, then, introduced new test instances of size up to 800 vertices and features systematically distributed so to allow a thorough study on the characteristics of the instances beside those of the algorithms. For several of these new instances we were still able to find the optimal solutions and this turned out to be a convenient aspect to characterize the performance of the heuristics designed. Instances derived by composition of tree and cycle are very easy to solve. Uniform and Euclidean instances are more challenging while the Small World instances are the hardest we could find for this problem. Overall, instances with uniform weights are harder than those with non-uniform weights.

The main focus of the paper has been on the computational study of exact and heuristic algorithms. Most of the heuristics were newly designed and all the methods were tested experimentally for the first time. The results indicated that greedy covering is the best construction heuristic among those studied on all types of instances. It produces approximations that are, on average, 5.5% from optimum in the Euclidean instances, 10% in the Uniform instances and 1.8-2.3% in the Small World instances. This heuristic uses the set covering formulation and outperforms others based on the graph representation, especially a promising one based on shortest path calculations. The concepts underlying these algorithms

were extended to implement local search procedures that use three new neighborhoods: addition, destruct-reconstruct and shortest path reconstruction. In particular, this latter implements a very large scale neighborhood search. The analysis indicated in the most simple approach, the addition neighborhood, the best choice both in terms of solution quality and computation cost. The number of improvements found by this local search over the solution provided by the construction heuristics was observed to range between 2 and 25 over all instances considered.

We, finally, enhanced further these heuristics by designing a hybrid method drawing on the previous results. The new algorithm that we call the Lagrangian Multi-Start (LMS) uses elements from different state-of-the-art set covering solvers. These elements are: (i) subgradient optimization of Lagrangian relaxation (ii) pricing (iii) iterated destruction and re-construction of the augmentation (iv) and use of the best construction heuristic and local search determined in this study.

The results indicated that set covering instances arising from the 2-edge-connectivity have a favorable structure that integer programming solvers can often solve efficiently. For instances with uniform weights and 800 vertices a state-of-the-art ILP solver, SCIP, needs the same time as the heuristics to find the optimal solutions. On instances with non-uniform weights, instead, the LMS heuristic finds with high chance the optimal solution and is quicker than the ILP solver. Although the LMS algorithm cannot prove the optimality of the solution, it exhibits a better growth curve and it is, therefore, appealing if very fast results are needed or if instances of even larger sizes than those here studied are to be solved. Finally, the ideas of LMS can also be combined within an integer programming framework to improve its performances on this specific problem.

Acknowledgments We thank the anonymous reviewers for their detailed comments that helped us in improving the article.

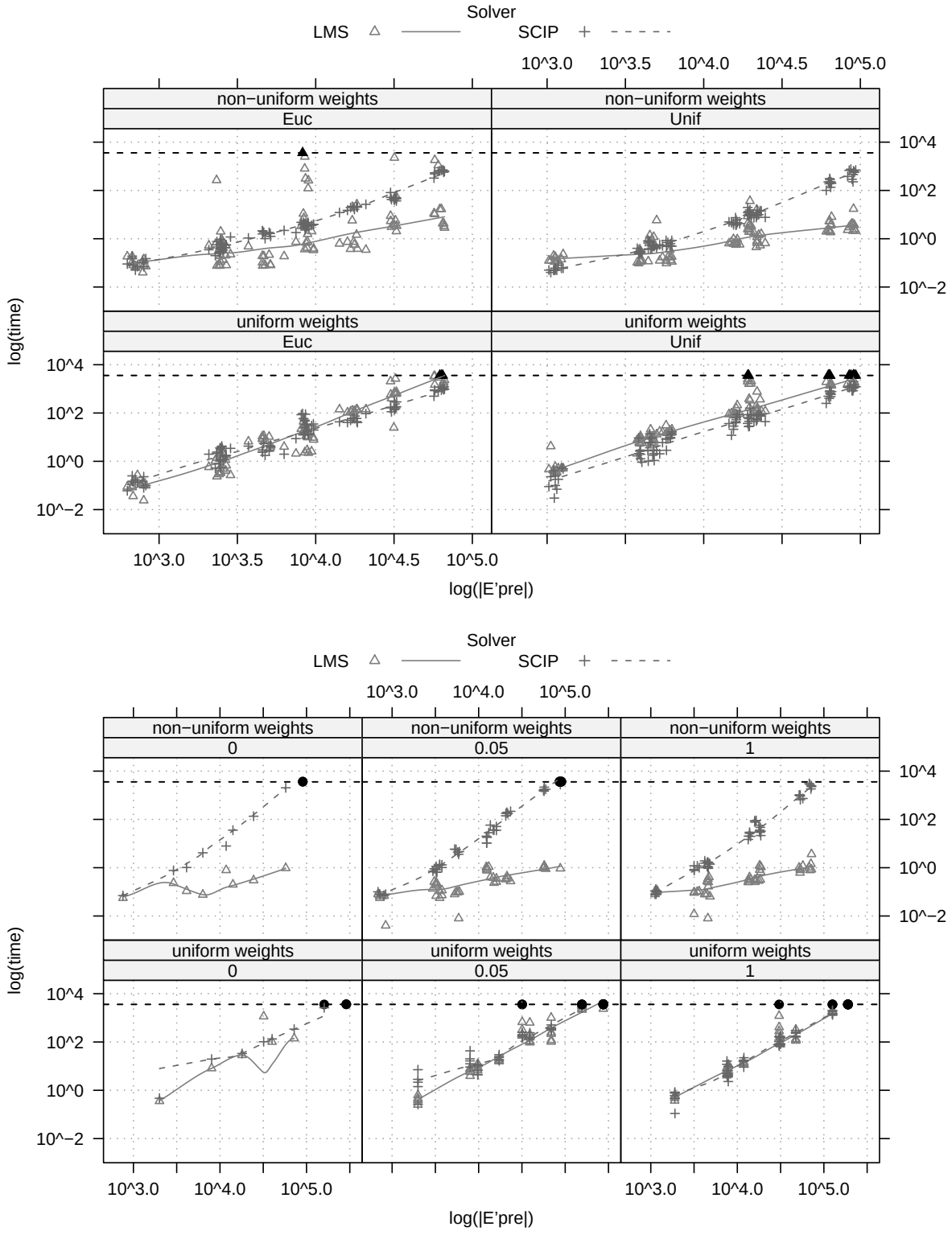

Figure 23: Computation time to solve `Unif`, `Euc` and `SmallWorld` instances with the SCIP and the LMS solver. A logarithmic transformation is applied to both axes. Time is expressed in seconds. Runs that ended without the optimal solution being found are indicated by triangles filled black for LMS and by circles filled black without distinction of the solvers. See Table 6 in the Appendix for numerical details.

Numerical Results

We report the numerical results on the instances presented by Raidl and Ljubic [39]. Details on the generator of these instances are available on the Internet.[17] Instances from A to R are random graphs and the others are derived from Euclidean TSPLIB graphs having included the complete or a fraction of shortest edges incident to each vertex. Spanning trees vary between being randomly chosen or being of minimal length.

In Table 3 we report the optimal results found by CPLEX 9.0 solver[18] on a machine more than 4 times slower than the one used for all the experiments described in this paper. The optimal solution values on these instances were not known and only heuristic results were reported in [28, 39, 44]. The hybrid evolutionary algorithm (HEA) from [39] was shown to outperform previous approaches and therefore we consider only this algorithm for comparison. In the table, we reprint its results published in [39] and transform times to make them comparable with our environment. Raidl and Ljubic used an Intel Pentium III, 500 MHz processor with 512 MB of RAM.[19] The conversion rates between machines were taken from the Standard Performance Evaluation Corporation (SPEC) CPU2000 benchmarks (`www.spec.org`). The results in column *Dev.* indicate the average percentage deviation in multiple runs from the best solution found X^* by HEA and in column *Rate* the percentage of times the best solution X^* was attained. For comparison we report the average results of 10 runs of the construction heuristic greedy_cov and of LMS. On greedy_cov we omit the computation times as they are in all cases below 10 milliseconds, on LMS we give the average time to find the optimal solutions. In Table 4, we report the solution found on the TSPLIB derived instances. Although mentioned in Ljubic's web page, results on these instances were never reported.

Tables 5 and 6 report statistics and results for the new instances used in our study. Times refer to the machine discussed in the Section 3.

[17] I. Ljubic. Generator Index Page. Last updated: November 2001.
`http://www.ads.tuwien.ac.at/research/NetworkDesign/Generator.html` Visited: May1 2007.
[18] CPLEX 9.0 is shown to be inferior in performance to SCIP 1.0 (`http://scip.zib.de/`, visited June 6, 2008).
[19] Personal communication.

	Instance			Exact (CPLEX)		greedy_cov	ELS (100 runs)			LMS
	$\|V\|$	$\|E'\|$	$\rho(G)$	Sol.	Time	Sol.	Dev.	Rate	Time	Time
A3	40	29	0.09	6607	0.02	6607	0.0	100	0.0	0.00
B1	60	55	0.06	15512	0.05	15512	0.0	100	0.0	0.00
B6	70	81	0.06	19022	0.01	19039	0.0	100	0.1	0.00
D3	90	366	0.11	20321	0.31	20991	0.0	100	0.9	0.04
D5	100	398	0.1	19355	0.36	19669	0.0	100	0.9	0.03
E1	200	19701	1	2873.8023	14.87	3112.2	1.0	23	63.0	0.12
E2	300	11015	0.25	9355.1971	23.20	10092.7	0.4	2	241.2	8.15
E3	400	7621	0.1	21326.1027	30.07	23079.2	0.6	0	478.9	11.99
M1	70	290	0.15	2940	0.20	3060	0.0	100	0.4	0.01
N1	100	1104	0.24	383	0.82	424	0.5	42	3.4	0.05
N2	110	1161	0.21	429	0.94	471	0.0	100	3.7	0.08
R1	200	9715	0.5	121.4188	11.25	124.9	0.0	100	44.6	0.21
R2	200	9745	0.5	320.4805	8.87	347.2	0.7	13	72.1	0.27

Table 3: Numerical results on the instances from [39]. In the columns concerning the instances, V is the set of vertices in G, $E' = E \setminus F$ is the set of augmenting edges, $\rho(G)$ is the edge density. Weights on the instances E1, E2 and E3 are Euclidean distances and are real valued, like those on the instances R1 and R2. On the other instances, weights are uniform random from varying ranges. Solution times are given in seconds and refer to a machine AMD Athlon XP 2000+, 1667 MHz, with 512 RAM. See text for details.

	Instance			Exact (CPLEX)		greedy_cov		LMS	
	$\|V\|$	$\|E'\|$	$\rho(G)$	Sol.	Time	Sol.	Time	Sol.	Time
lin318m0	318	50086	1	11520	78.02	12494	1.4	11520	3.1
lin318m10	318	5495	0.12	11520	13.11	12494	0.1	11520	4.0
pa561m0	561	156520	1	755	480.53	818	10.8	755	29.4
pa561m10	561	18504	0.12	755	72.50	818	0.6	755	14.8
pcb442m0	442	97020	1	10313	182.56	10918	3.1	10313	7.3
pcb442m10	442	10528	0.11	10313	25.40	10959	0.2	10313	59.7
pr226m0	226	25200	1	22574	33.08	28543	0.7	22574	0.9
pr226m15	226	3987	0.17	22574	10.89	28743	0.1	22574	0.3
pr439m0	439	95703	1	26030	229.89	31461	4.7	26030	1.6
pr439m10	439	11183	0.12	26030	37.40	31661	0.2	26030	1.2

Table 4: Numerical results on TSPLIB instances from the Ljubic's web site. For the meaning of the labels in the columns see caption of Table 3. Times refer to a machine Intel Celeron 2.40 GHz with 128 KB cache and 512 MB RAM.

Instances	$\|V\|$	$\|E'\|$	$\rho(G)$	w_{min}	w_{max}	$\|V_{pre}\|$	$\|E'_{pre}\|$	$\rho(G_{pre})$	$\rho(M_{pre})$
Unif-200-0.1	200	1781.4	0.1	589.2	9993.3	200	1138.5	0.067	0.074
Unif-200-0.5	200	9724.1	0.499	105	9998.8	200	3955.7	0.209	0.072
Unif-200-0.9	200	17683.1	0.899	59.3	9998.9	200	5899.4	0.306	0.071
Unif-400-0.1	400	7588.8	0.1	250.1	9998.6	400	4681.4	0.064	0.051
Unif-400-0.5	400	39479.9	0.5	49.6	9998.9	400	15814.2	0.203	0.047
Unif-400-0.9	400	71373.7	0.899	28.5	9999	400	22338.2	0.285	0.048
Unif-800-0.1	800	31237.5	0.1	109.5	9999	800	19495.1	0.063	0.032
Unif-800-0.5	800	159011.3	0.5	23.9	9999	800	63758	0.202	0.031
Unif-800-0.9	800	286806.6	0.9	13.8	9999	800	88973.3	0.281	0.033
Euc-200-0.1	200	1899.1	0.105	191.5	1999.8	199	729.8	0.047	0.038
Euc-200-0.5	200	9666.4	0.496	191.4	5100	200	2499.8	0.136	0.082
Euc-200-1	200	19701	1	191.4	13215.1	200	4846.8	0.254	0.125
Euc-400-0.1	400	7984.2	0.105	84.6	2000	400	2473.7	0.036	0.03
Euc-400-0.5	400	38948.4	0.493	84.6	5100	400	8899.7	0.117	0.065
Euc-400-1	400	79401	1	84.6	13408.9	400	17499.3	0.224	0.1
Euc-800-0.1	800	32880.5	0.105	61.6	2000	800	8455.9	0.029	0.025
Euc-800-0.5	800	158203.7	0.498	61.6	5100	800	31617.9	0.101	0.053
Euc-800-1	800	318801	1	61.6	13686.2	800	62038.2	0.197	0.078

Instances	$\|V\|$	d	p	$\|E'\|$	$\rho(G)$	$\|V_{pre}\|$	$\|E'_{pre}\|$	$\rho(G_{pre})$	$\rho(M_{pre})$
sw-400-40-0-1	400	40	0	8000	0.1	400	8000	0.1	0.034
sw-400-40-0-w	400	40	0	8000	0.1	400	2905	0.036	0.039
sw-400-40-0.05-1	400	40	0.05	7959.1	0.1	400	7959.1	0.1	0.032
sw-400-40-0.05-w	400	40	0.05	7959.1	0.1	400	3158.2	0.04	0.039
sw-400-40-1-1	400	40	1	7584.6	0.095	400	7584.6	0.095	0.042
sw-400-40-1-w	400	40	1	7584.6	0.095	400	4617.1	0.058	0.046
sw-400-200-0-1	400	200	0	40000	0.501	400	40000	0.501	0.043
sw-400-200-0-w	400	200	0	40000	0.501	400	14118	0.177	0.047
sw-400-200-0.05-1	400	200	0.05	39033.5	0.489	400	39033.5	0.489	0.043
sw-400-200-0.05-w	400	200	0.05	39033.5	0.489	400	14827	0.186	0.046
sw-400-200-1-1	400	200	1	31381.5	0.393	400	31381.5	0.393	0.045
sw-400-200-1-w	400	200	1	31381.5	0.393	400	13508.7	0.169	0.049
sw-400-360-0-1	400	360	0	72000	0.902	400	72000	0.902	0.037
sw-400-360-0-w	400	360	0	72000	0.902	400	24466	0.307	0.04
sw-400-360-0.05-1	400	360	0.05	68916.9	0.864	400	68916.9	0.864	0.047
sw-400-360-0.05-w	400	360	0.05	68916.9	0.864	400	21878.6	0.274	0.051
sw-400-360-1-1	400	360	1	47355	0.593	400	47355	0.593	0.046
sw-400-360-1-w	400	360	1	47355	0.593	400	17213.3	0.216	0.05
sw-800-80-0-1	800	80	0	32000	0.1	800	32000	0.1	0.02
sw-800-80-0-w	800	80	0	32000	0.1	800	11787	0.037	0.023
sw-800-80-0.05-1	800	80	0.05	31844.5	0.1	800	31844.5	0.1	0.027
sw-800-80-0.05-w	800	80	0.05	31844.5	0.1	800	12416.3	0.039	0.031
sw-800-80-1-1	800	80	1	30425.2	0.095	800	30425.2	0.095	0.029
sw-800-80-1-w	800	80	1	30425.2	0.095	800	18516.6	0.058	0.031
sw-800-400-0-1	800	400	0	160000	0.501	800	160000	0.501	0.034
sw-800-400-0-w	800	400	0	160000	0.501	800	57528	0.18	0.036
sw-800-400-0.05-1	800	400	0.05	156150.2	0.489	800	156150.2	0.489	0.032
sw-800-400-0.05-w	800	400	0.05	156150.2	0.489	800	57148.4	0.179	0.034
sw-800-400-1-1	800	400	1	125753.5	0.393	800	125753.5	0.393	0.029
sw-800-400-1-w	800	400	1	125753.5	0.393	800	53737.8	0.168	0.031
sw-800-720-0-1	800	720	0	288000	0.901	800	288000	0.901	0.03
sw-800-720-0-w	800	720	0	288000	0.901	800	90479	0.283	0.032
sw-800-720-0.05-1	800	720	0.05	275662.6	0.863	800	275662.6	0.863	0.03
sw-800-720-0.05-w	800	720	0.05	275662.6	0.863	800	87607.7	0.274	0.031
sw-800-720-1-1	800	720	1	189672.6	0.593	800	189672.6	0.593	0.03
sw-800-720-1-w	800	720	1	189672.6	0.593	800	70402.9	0.22	0.032

Table 5: Statistics about the new random instances, `Euc`, `Unif` and `SmallWorld`. Each line represents an instance class consisting of 10 or 5 instances and data are averaged among those instances. In the columns, V is the set of vertices in G, $E' = E \setminus F$ is the set of augmenting edges, w_{min} and w_{max} give the range of weights on the edges of E', $\rho(G)$ is the edge density in G, the subscript *pre* indicates the values after preprocessing, and $\rho(M_{pre})$ is ones density in the matrix M_{pre} of the corresponding set covering formulation. On the `SmallWorld` instances d indicates the vertex degree and p the rewiring probability. Weights for these class are randomly chosen from $[1, 10000]$.

Instance Class	Optimal Sol. median	Optimal Sol. IQR	# Sol. median	# Sol. IQR	Seconds (SCIP 1.0) median	Seconds (SCIP 1.0) IQR	Sol. (LMS) median	Sol. (LMS) IQR	Seconds (LMS) median	Seconds (LMS) IQR
U-400-0.1-1	81	2.75	17.5	16.75	2.87	3.71	81	2.75	7.81	5.04
U-400-0.1-w	96747	8810	1	0	0.5	0.24	96747	8810	0.8	0.48
U-400-0.5-1	82.5	4.5	99	120.25	41.42	30.59	82.5	4.5	90.85	10.48
U-400-0.5-w	19500	573	1	0	4.56	0.97	19500	573	0.65	0.13
U-400-0.9-1	80.5	2.5	85	63.75	66.6	30.38	80.5	2.5	128.11	64.86
U-400-0.9-w	10633.5	735.75	1	0	9.64	5.37	10633.5	735.75	0.99	0.87
U-800-0.1-1	162.5	7.75	92	100.75	61.64	43.36	164	7.75	1064.91	1580.96
U-800-0.1-w	99889.5	3805.25	1	0	11.12	3.75	99889.5	3805.25	3.41	1.69
U-800-0.5-1	165.5	2.5	323	117.5	595.87	264.57	166	1.75	1479.51	510.8
U-800-0.5-w	20119	1034	1	0	212.81	35.76	20119	1034	2.87	2.96
U-800-0.9-1	161	2.75	399.5	51.75	1164.07	219.88	163.5	4.5	2835.32	1184.14
U-800-0.9-w	11050.5	573.75	1	0	570.19	271.05	11050.5	573.75	3.35	1.62
G-400-0.1-w	36512.5	2984.5	8	0.75	0.42	0.14	36512.5	2984.5	0.7	0.33
G-400-0.5-1	45.5	2.5	67.5	58.5	19.45	18.37	45.5	2.5	5.39	7.42
G-400-0.5-w	36512.5	2932.75	8	0	3.54	0.78	36512.5	2932.75	0.71	0.96
G-400-1-1	45.5	2.5	64	57.25	54.08	15.82	45.5	2.5	128.94	24.33
G-400-1-w	36512.5	2932.75	8	2.5	20.15	4.38	36512.5	2932.75	0.68	0.85
G-800-0.1-1	88.5	3.75	334.5	82.5	54.3	48.17	88.5	3.75	19.63	12.4
G-800-0.1-w	48865	1701.75	13.5	3.75	4.11	0.7	48865	1701	66.57	673.16
G-800-0.5-1	88.5	3.5	258	71	170.73	28.16	88.5	3.5	650.14	136.37
G-800-0.5-w	48780	1652	18	5.25	45.91	6.81	48780	1652	4.64	4.56
G-800-1-1	88.5	4.25	275.5	88.5	915.38	177.87	89.5	4.5	2069.28	1641.85
G-800-1-w	48788	1748	18	5.5	614.48	108.17	48788	1748	10.75	12.56

Instance Class	Optimal Sol. median	Optimal Sol. IQR	Gap. max	Seconds (SCIP 1.0) median	Seconds (SCIP 1.0) IQR	Sol. (LMS) median	Sol. (LMS) IQR	Seconds (LMS) median	Seconds (LMS) IQR
sm-400-40-0-1	80	0	0	19.61	0	80	0	8.14	0
sm-400-40-0-w	34115	0	0	0.77	0	34115	0	0.23	0
sm-400-40-0.05-1	81	7	0	16.49	7.57	81	7	5.94	1.58
sm-400-40-0.05-w	35831	3475	0	0.72	0.16	35831	3475	0.21	0.06
sm-400-40-1-1	81	4	0	6.44	6.96	81	4	6.2	2.17
sm-400-40-1-w	37634	1685	0	1.1	0.07	37634	1685	0.28	0.09
sm-400-200-0-1	81	0	0	133.77	0	81	0	101.58	0
sm-400-200-0-w	7077	0	0	35.71	0	7077	0	0.2	0
sm-400-200-0.05-1	83	8	0	152.92	48.78	83	8	108.72	105.04
sm-400-200-0.05-w	7545	372	0	36.69	14.52	7545	372	0.35	0.12
sm-400-200-1-1	82	1	0	149.32	66.98	82	1	90.01	6.56
sm-400-200-1-w	9455	825	0	20.37	6.51	9455	825	0.31	0.08
sm-400-360-0-1	87	0	0	339.79	0	87	0	140.75	0
sm-400-360-0-w	3918	0	0	135.03	0	3918	0	0.3	0
sm-400-360-0.05-1	79	2	0	357.93	66	79	2	216.58	128.78
sm-400-360-0.05-w	3873	372	0	177.34	22.82	3873	372	0.34	0.03
sm-400-360-1-1	78	7	0	229.36	67.32	78	7	125.95	12.42
sm-400-360-1-w	5954	94	0	76.83	31.55	5954	94	0.31	0.03
sm-800-80-0-1	164	0	0	102.29	0	164	0	1145.41	0
sm-800-80-0-w	37515	0	0	7.91	0	37515	0	0.79	0
sm-800-80-0.05-1	160	4	0	185.11	35.3	160	4	669.39	3277.02
sm-800-80-0.05-w	34249	1648	0	18.07	9.44	34249	1648	0.96	0.23
sm-800-80-1-1	161	5	0	105.63	43.83	161	6	409.55	924.49
sm-800-80-1-w	36212	2172	0	31.92	3.24	36212	2172	1.05	0.38
sm-800-400-0-1	164	0	0	2613.8	0	167	0	3598.3	0
sm-800-400-0-w	6897	0	0	2036.98	0	6897	0	0.96	0
sm-800-400-0.05-1	163	7	1.89	3107.86	672.13	164	7	2393.42	584.48
sm-800-400-0.05-w	7337	443	0	1731.73	193.24	7337	443	1.01	0.07
sm-800-400-1-1	161	2	0	1485.7	387.49	162	2	3598.44	1656.14
sm-800-400-1-w	9165	686	0	925.48	283.92	9165	686	0.88	0.2
sm-800-720-0-1	-1e+20	0	∞	3600.00	0	165	0	3597.62	0
sm-800-720-0-w	-1e+20	0	∞	3600.00	0	3787	0	3598.39	0
sm-800-720-0.05-1	-1e+20	0	∞	3600.00	0	162	2	3597.5	0.14
sm-800-720-0.05-w	-1e+20	0	∞	3600.00	0	4226	353	3598.33	0.13
sm-800-720-1-1	161	7	8.28	3600.00	0	162	6	3598.06	0.12
sm-800-720-1-w	5641	351	0	2442.05	458.64	5641	351	1.02	0.58

Table 6: Results on the new instances. Every line accounts for 10 or 5 instances and median and interquartile (IQR) values are reported.

References

[1] T. Achterberg, Constraint integer programming, Ph.D. thesis, Technische Universität Berlin (2007).

[2] R. Ahuja, O. Ergun, J. Orlin, A. Punnen, A survey of very large scale neighborhood search techniques, Discrete Applied Mathematics 123 (2002), 75–102.

[3] E. Balas, M. Carrera, A dynamic subgradient-based branch-and-bound procedure for set covering, Operations Research 44 (1996), 875–890.

[4] J. Bang-Jensen, M. Chiarandini, Y. Goegebeur, B. Jørgensen, Mixed models for the analysis of local search components, in: T. Stützle, M. Birattari, H. Hoos (eds.), Engineering Stochastic Local Search Algorithms: Designing, Implementing and Analyzing Effective Heuristics. International Workshop, SLS 2007, vol. 4638 of Lecture Notes in Computer Science, Springer Verlag, Berlin, Germany, 2007.

[5] J. Bang-Jensen, G. Gutin, Digraphs: Theory, Algorithms and Applications, Springer Verlag, London, 2001.

[6] J. Bang-Jensen, T. Jordán, Edge-connectivity augmentation preserving simplicity, SIAM Journal on Discrete Mathematics 11 (1998), 603–624.

[7] T. Bartz-Beielstein, S. Markon, Tuning search algorithms for real-world applications: A regression tree based approach, in: Congress on Evolutionary Computation (CEC'04), IEEE Press, Piscataway NJ, 2004.

[8] J. Beasley, OR-library: distributing test problems by electronic mail, Journal of the Operational Research Society 41 (1990), 1069–1072.

[9] J. Beasley, P. Chu, A genetic algorithm for the set covering problem, European Journal of Operational Research 94 (1996), 392–404.

[10] L. Breiman, Classification and regression trees, Wadsworth, Belmont, CA, USA, 1984.

[11] G.-R. Cai, Y.-G. Sun, The minimum augmentation of any graph to a k-edge-connected graph, Networks 19 (1989), 151–172.

[12] A. Caprara, M. Fischetti, P. Toth, A heuristic method for the set covering problem, Operations Research 47 (1999), 730–743.

[13] M. Caserta, Tabu search-based set covering algorithm, in: K.F. Doerner et al. (ed.), MIC2005: The Sixth Metaheuristics International Conference, Vienna, Austria, 2005.

[14] J. Cheriyan, T. Jordán, R. Ravi, On 2-coverings and 2-packings of laminar families., in: J. Nesetril (ed.), Algorithms - ESA '99 Proceedings, vol. 1643 of Lecture Notes in Computer Science, Springer Verlag, 1999.

[15] V. Chvatal, A greedy heuristic for the set-covering problem, Mathematics of Operation Research 4 (1979), 233–235.

[16] M. Conforti, A. Galluccio, G. Proietti, Edge-connectivity augmentation and network matrices, in: WG 2004, vol. 3353 of Lecture Notes in Computer Science, Springer Verlag, 2004.

[17] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., MIT press, 2001.

[18] K. Eswaran, R. Tarjan, Augmentation problems, SIAM Journal on Computing 5 (1976), 653–665.

[19] G. Even, J. Feldman, G. Kortsarz, Z. Nutov, A 3/2-approximation algorithm for augmenting the edge-connectivity of a graph from 1 to 2 using a subset of a given edge set, in: M.X. Goemans et al. (ed.), 4th Int. Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2001), vol. 2129 of Lecture Notes in Computer Science, Springer Verlag, 2001.

[20] A. Farley, A note on bounding a class of linear programming problems, including cutting stock problems, Operations Research 38 (1990), 922–923.

[21] M. L. Fisher, The Lagrangian relaxation method for solving interger programming problems, Management Science 27 (1981), 1–18.

[22] A. Frank, Augmenting graphs to meet edge connectivity requirements, SIAM Journal on Discrete Mathematics 5 (1992), 25–53.

[23] G. Frederickson, J. JáJá, Approximation algorithms for several graph augmentation problems, SIAM Journal on Computing 10 (1981), 270–283.

[24] A. Galluccio, G. Proietti, Polynomial time algorithms for 2-edge-connectivity augmentation problems, Algorithmica 36 (2003), 361–374.

[25] H. Hoos, T. Stützle, Stochastic Local Search: Foundations and Applications, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.

[26] T. Hothorn, K. Hornik, A. Zeileis, Unbiased recursive partitioning: A conditional inference framework, Journal of Computational and Graphical Statistics 15 (2006), 651–674.

[27] H. Kerivin, A. R. Mahjoub, Design of survivable networks: A survey, Networks 46 (2005), 1–67.

[28] S. Khuller, B. Raghavachari, A. Zhu, A uniform framework for approximating weighted connectivity problems, in: SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[29] S. Khuller, R. Thurimella, Approximation algorithms for graph augmentation, Journal of Algorithms 14 (1993), 214–225.

[30] L. Lessing, I. Dumitrescu, T. Stützle, A comparison between aco algorithms for the set covering problem., in: M. Dorigo, M. Birattari, C. Blum, L. M. Gambardella, F. Mondada, T. Stützle (eds.), Ant Colony Optimization and Swarm Intelligence, 4th International Workshop, Brussels, Belgium, vol. 3172 of Lecture Notes in Computer Science, Springer, 2004.

[31] I. Ljubic, G. Raidl, J. Kratica, A hybrid GA for the edge-biconnectivity augmentation problem, in: X. Y. K. Deb, G. Rudolph, H.-P. Schwefel (eds.), Proceedings of the 2000 parallel problem solving from Nature VI Conference, vol. 1917 of Lecture Notes of Computer Science, 2000.

[32] I. Ljubic, G. R. Raidl, An evolutionary algorithm with stochastic hill-climbing for the edge-biconnectivity augmentation problem., in: E.J.W. Boers et al. (ed.), EvoWorkshops, vol. 2037 of Lecture Notes in Computer Science, Springer Verlag, 2001.

[33] T. L. Magnanti, S. Raghavan, Strong formulations for network design problems with connectivity requirements, Networks 45 (2005), 61–79.

[34] E. Marchiori, A. G. Steenbeek, An evolutionary algorithm for large scale set covering problems with application to airline crew scheduling, in: Cagnoni et al. (ed.), EvoWorkshops 2000, vol. 1803 of Lecture Notes in Computer Science, Springer Verlag, 2000.

[35] D. C. Montgomery, Design and Analysis of Experiments, sixth ed., John Wiley & Sons, 2005.

[36] P. Morling, Heuristics for increasing the edge-connectivity of a given weighted network, Master's thesis, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark (2007).

[37] H. Nagamochi, T. Ibaraki, Computing edge-connectivity in multigraphs and capacitated graphs, SIAM Journal on Descrete Methods 5 (1992), 54–66.

[38] D. Naor, D. Gusfield, C. U. Martel, A fast algorithm for optimally increasing the edge-connectivity, in: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, IEEE Computer Society, 1990.

[39] G. R. Raidl, I. Ljubic, Evolutionary local search for the edge-biconnectivity augmentation problem, Information Processing Letters 82 (2002), 39–45.

[40] R. E. Tarjan, A note on finding the bridges of a graph, Information Processing Letters 2 (1974), 160–161.

[41] T. Watanabe, A. Nakamura, Edge-connectivity augmentation problems, Journal of Computer and System Sciences 35 (1987), 96–144.

[42] D. J. Watts, S. H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (1998), 440–442.

[43] L. A. Wolsey, Integer programming, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, New York, USA, 1998.

[44] F. Xhafa, An implementation of a generic memetic algorithm for the edge biconnectivity augmentation problem, Tech. Rep. LSI-03-47-R, Department of Languages and Informatics Systems, Polytechnic University of Catalonia, Barcelona, Spain (2003).

[45] M. Yagiura, M. Kishida, T. Ibaraki, A 3-flip neighborhood local search for the set covering problem, European Journal of Operational Research 172 (2006), 472–499.

[46] A. Zhu, A uniform framework for approximating weighted connectivity problems, Master's thesis, University of Mayland, MD, USA (1999).