A Fast, Adaptive Variant of the Goemans-Williamson Scheme for the Prize-Collecting Steiner Tree Problem

Chinmay Hegde, Piotr Indyk, and Ludwig Schmidt

Massachusetts Institute of Technology, Cambridge MA 02139, USA

Abstract. We introduce a new variant of the Goemans-Williamson (GW) scheme for the Prize-Collecting Steiner Tree Problem (PCST). Motivated by applications in signal processing, the focus of our contribution is to construct a very fast algorithm for the PCST problem that still achieves a provable approximation guarantee. Our overall algorithm runs in time $O(d\,m \log n)$ on a graph with m edges, where all edge costs and node prizes are specified with d bits of precision. Moreover, our algorithm maintains the Lagrangian-preserving factor-2 approximation guarantee of the GW scheme.

Similar to [Cole, Hariharan, Lewenstein, and Porat, SODA 2001], we use dynamic edge splitting in order to efficiently process all cluster merge and deactivation events in the moat-growing stage of the GW scheme. Our edge splitting rules are more adaptive to the input, thereby reducing the amount of time spent on processing intermediate edge events.

Numerical experiments based on the public DIMACS test instances show that our edge splitting rules are very effective in practice. In most test cases, the number of edge events processed per edge is less than 2 on average. On a laptop computer from 2010, the longest running time of our implementation on a DIMACS challenge instance is roughly 1.3 seconds (the corresponding instance has about 340,000 edges). Since the running time of our algorithm scales nearly linearly with the input size and exhibits good constant factors, we believe that our algorithm could potentially be useful in a variety of applied settings.

1 Introduction

The *prize-collecting Steiner tree problem* (PCST) is a generalization of the classical Steiner tree problem. Given a weighted graph with designated terminal nodes, the Steiner tree problem requires us to find a minimum-cost spanning tree of the terminal nodes. The prize-collecting variant of the problem relaxes the connectivity constraint by assigning a *prize* to each terminal node: instead of connecting a terminal node to our spanning tree, we can instead choose to forego the prize of the omitted terminal. The goal is to minimize the sum of costs incurred by adding edges to and omitting terminals from our spanning tree. This formulation naturally captures the trade-off between the costs of connecting a terminal and omitting it from the solution. The PCST problem has found applications in communications network design [JMP00], computational biology [IOSS02,DKR+08], and event detection in social networks [RAGT14].

Since it is a generalization of the Steiner tree problem, the PCST problem is NP-hard [Kar72]. Therefore, research on algorithms with guaranteed polynomial running time has focused on providing good *approximation* guarantees. The first algorithm with a constant-factor guarantee appeared in [BGSLW93], which also introduced the PCST problem. The algorithm gives a factor-3 guarantee and is based on solving a linear program. Building on [AKR91], the seminal work of Goemans and Williamson [GW95] achieves a factor-2 guarantee with a purely combinatorial primal-dual algorithm. Following this line of work, the currently best known approximation ratio for the PCST problem is 1.9672 [ABHK11,HK13]. On the question of hardness of approximation, the authors of [CC02] show that it is NP-hard to approximate the Steiner tree problem, and hence also PCST, within a factor of 96/95.

The specific motivation for our contribution arises from applications in signal processing and sparse approximation. In this context, we are interested in solving PCST instances on large graphs, e.g., a grid graph with $n = 10^6$ nodes corresponding to a digital image with a resolution of 1 Megapixel. Since the originally proposed GW scheme has a running time of $O(n^2 \log n)$ (or strictly speaking even $O(n^3 \log n)$ for the unrooted PCST problem), a direct implementation of the GW algorithm quickly becomes impractical in such a regime of parameters. Therefore, our focus is on algorithms with a very fast running time that still offer a good approximation guarantee.

In this paper, we propose a fast, *data-adaptive* variant of the GW scheme that has a running time of $O(m \log n)$ when the input is specified with constant precision. In particular, our algorithm builds on [CHLP01], who give an implementation of the GW scheme running in $O(m \log^2 n)$ time where m is the number of edges in the input graph. In comparison, our algorithm saves an extra logarithmic factor in the running time, while still achieving the same factor-2 approximation guarantee as the GW scheme. Our experimental results show that our algorithm is highly efficient; on the test cases of the DIMACS challenge, the longest running time for a single instance is roughly 1.3s on a laptop computer from 2010. This instance has about 170,000 nodes and 340,000 edges (`PCSPG-hand/handbd07.stp`). To the best of our knowledge, our algorithm is

the first practical implementation of a fast GW scheme following an approach similar to [CHLP01].

This paper is organized as follows. We introduce the necessary notation and give a high-level description of the GW scheme in Section 2. We then present our algorithm in Section 3 and analyze its performance in Section 4. Finally, we show results of our computational experiments on the public test data provided by DIMACS, as well as some of our own test cases, in Section 5.

2 Preliminaries

Unless noted otherwise, we assume the following notation in the rest of the paper. $G = (V, E)$ is a connected, undirected graph with edge weights $c : E \to \mathbb{R}_0^+$ and node prizes $\pi : V \to \mathbb{R}_0^+$. The number of nodes is $n = |V|$ and the number of edges is $m = |E|$. For a subset of nodes $U \subseteq V$, we write $\pi(U) = \sum_{u \in U} \pi(u)$, and adopt the same convention for a subset of E. We use $\overline{U}$ to denote the complement with respect to V, i.e., $\overline{U} := V \setminus U$. Finally, we sometimes use a subgraph $H = (V_H, E_H)$ in place of the corresponding node or edge sets if the meaning is clear from context.

Definition 1 (Prize-collecting Steiner tree problem (PCST)). *Find a subtree T of G such that $c(T) + \pi(\overline{T})$ is minimized.*

This definition is for the *unrooted* variant of the PCST problem. The original GW algorithm was for the *rooted* variant in which a designated root node is required to be part of the solution T. While the unrooted variant can always be reduced to the rooted variant by running the rooted algorithm with all n choices of the root node, this incurs a factor-n penalty in the running time. Later works address this issues and introduce algorithms that directly solve the unrooted variant [JMP00,FFFdP10]. In the rest of the paper, we limit our attention to the unrooted PCST problem defined above.

The GW algorithm satisfies the following approximation guarantee for the PCST problem:

$$c(T) + 2\pi(\overline{T}) \;\leq\; 2c(T_{OPT}) + 2\pi(\overline{T_{OPT}}) \; , \tag{1}$$

where T_{OPT} is a tree minimizing $c(T_{OPT}) + \pi(\overline{T_{OPT}})$. It is worth noting that this guarantee is stronger than a standard factor-2 approximation guarantee because of the $2\pi(\overline{T})$ term on the left hand side (for a standard factor-2 guarantee, simply $\pi(\overline{T})$ would suffice). This guarantee is sometimes called a *Lagrangian-preserving* guarantee and was introduced for the GW algorithm in [CRW04][1]. Our algorithm will also satisfy this stronger type of guarantee.

[1] In fact, the stronger Lagrangian-preserving guarantee was already implicitly provided by the approach in [GW95]; see [ABHK11] for a more detailed account of these results.

2.1 The Goemans-Williamson (GW) algorithm

We now give a high-level overview of the GW algorithm. For a more detailed exposition, we refer the reader to [GW95,WS11,FFFdP10]. Before we begin with our description, we introduce the definition of a *laminar family*. Intuitively, a laminar family describes a recursive partitioning of the nodes into nested subsets. From a bottom-up perspective, this corresponds to a *clustering* of the nodes in which new clusters are formed by merging existing clusters. Here and in the rest of the paper, we use the term "cluster" when we refer to elements in a *laminar family*, defined as follows.

Definition 2 (Laminar family). *A family $\mathscr{L}$ of non-empty subsets of V is a* laminar family *if one of the following three cases holds for all $L_1, L_2 \in \mathscr{L}$: either $L_1 \cap L_2 = \{\}$, or $L_1 \subseteq L_2$, or $L_2 \subseteq L_1$.*

We say that a set $L \in \mathscr{L}$ is maximal *if there is no $L' \in \mathscr{L}$ with $L \subsetneq L'$.*

The GW algorithm can be divided into two stages:

1. In the growth stage, the algorithm maintains a set of clusters (a laminar family) and merges or deactivates clusters until only a single active cluster remains. In parallel, the algorithm maintains a spanning tree for each cluster.
2. In the pruning stage, the algorithm removes unnecessary nodes from the spanning tree of the last active cluster identified in the growth stage.

The original pruning strategy introduced in [GW95], which we call GW pruning, depends on the laminar family found in the growth stage. The paper [JMP00] introduces another pruning strategy, so-called strong pruning, which only depends on the spanning tree identified at the end of the growth stage and also achieves an approximation ratio at least as good as GW pruning. In essence, strong pruning solves the PCST problem exactly on this spanning tree via a dynamic program (DP) that exploits the tree structure. Since this DP can be implemented relatively easily in linear time [JMP00], we focus on the GW growth stage in the rest of this paper and refer the reader to [JMP00] for the pruning stage.

We now describe the growth stage of the GW algorithm in more detail, using the "moat-growing" interpretation introduced in [JP95]. The growth stage clusters the nodes of G in a laminar family $\mathscr{L}$ such that every node of V is always in a cluster, i.e., the maximal sets of $\mathscr{L}$ partition V. Every cluster in $\mathscr{L}$ is either active or inactive and only maximal sets can be active. Initially, each node forms its own active singleton cluster.

Furthermore, the algorithm maintains a *moat* $y_C \in \mathbb{R}_0^+$ for each cluster $C \in \mathscr{L}$. Initially, all moat values are 0. Only the moats of currently active clusters grow, but inactive clusters keep their moats. In the primal-dual interpretation of the GW algorithm, the moats can be viewed as the dual variables. In particular, during the execution of the algorithm, the following two properties always hold:

- For every edge $e = (u, v)$, the following holds: let $\mathscr{C}$ be the set of clusters $C \in \mathscr{L}$ such that either $u \in C$ or $v \in C$, but not both. Then we have $\sum_{C \in \mathscr{C}} y_C \leq$

$c(e)$, i.e., the moats on an edge never "spill over". If this inequality is tight, we say that the edge constraint is tight.
- For every cluster $C \in \mathscr{L}$, the following holds: let $\mathscr{C}$ be the set of clusters $C' \in \mathscr{L}$ such that $C' \subseteq C$ (so $\mathscr{C}$ contains all clusters that were merged into C or one of its predecessors over the course of the algorithm). Then we have $\sum_{C' \in \mathscr{C}} y_{C'} \leq \sum_{v \in C} \pi(v)$, i.e., the moats in a cluster C and its child-clusters never exceed the "worth" of the cluster (the sum of prizes). If the ineqality is tight, we say that the cluster constraint is tight.

Finally, the algorithm maintains a set of edges F such that F restricted to any cluster $C \in \mathscr{L}$ forms a spanning tree of C.

We now state the GW algorithm in pseudo code (see Algorithm 2.1). The algorithm can be viewed as a simulation of the moat-growing process described above. Based on the current state of the "moat world", the algorithm decides when the next event will occur, moves the global time forward to this event, and processes the event by merging or deactivating clusters accordingly.

Algorithm 1 The GW algorithm for the PCST problem. The growth stage iteratively clusters the nodes of the graph and builds a corresponding spanning forest. For details of the pruning stage, see [GW95] or [JMP00].

```
 1: function GWAlgorithm(V, E, c, π)
 2:     Initialize singleton active clusters, moat values, and the spanning forest F.
 3:     while there is at least one active cluster do
 4:         Increase the moat values y_C for active clusters C until either an edge
                    constraint becomes tight or a cluster constraint becomes tight.
 5:         if an edge constraint for e = (u, v) becomes tight then
 6:             Let C_u and C_v be the maximal clusters in ℒ containing u and v,
                                                                      respectively.
 7:             Deactivate C_u and C_v.
 8:             Let C' = C_u ∪ C_v and add C' as an active cluster to ℒ.
 9:             Set y_C' = 0.
10:             Remove all edges e' with both endpoints contained in C' from E.
11:             Add e to F.
12:         if a cluster constraint for C becomes tight then
13:             Mark C as inactive.
14:     Run a pruning scheme on F restricted to the active cluster.
```

We now discuss the running time of the GW algorithm. It is easy to see that the while-loop in GWAlgorithm (lines 3 to 13) is executed at most $2n$ times: every iteration either reduces the number of clusters by one or deactivates a cluster while creating no new clusters. The main difficulty lies in quickly determining the next event in line 4. The cluster constraints are (relatively) easy to handle, for example, using a priority queue. However, detecting when an edge constraint will become tight requires more care. A straightforward implementation iterating over all edges would use at least $\Omega(mn)$ time. A more efficient

approach is to maintain another priority queue containing edges sorted by their remaining slack values. However, care has to be taken when clusters are merged or deactivated. Note, for instance, that an edge $e = (u, v)$ with previously deactivated endpoints can become active again when a cluster containing u or v is merged with an active cluster because the new cluster will grow a moat on edge e. In [GW95], the authors observe that it is sufficient to keep only a subset of the edges, i.e., for every pair of clusters, we only need to keep track of the edge with the smallest remaining slack values. Therefore, every cluster has only n incident edges at any time and we can update the priority queue for all $O(n)$ affected edges at merge or deactivation events in $O(n \log n)$ time. The primary contribution of [CHLP01], as well as our work in this paper, is to implement this moat-growing scheme in nearly-linear time.

2.2 Faster variants of the GW algorithm

The difficulties encountered when accelerating the GW scheme arise from the fact that there are three different types of edges: "active" edges where both endpoints are in active clusters, "semi-active" edges where only one endpoint is in an active cluster, and "inactive" edges with both endpoints inactive. Inactive edges can be ignored when looking for the next tight edge constraint. However, consider the active and semi-active edges. The slack values on active edges shrink twice as fast as the slack values on semi-active edges. Moreover, the type of an edge can change repeatedly over the course of the GW algorithm as clusters become inactive and then are activated again after merging with other active clusters. As a result, the time at which an edge constraint becomes tight can vary for many edges when the state of a single cluster changes.

There have been multiple approaches to reduce the running time of the GW algorithm by keeping track of only a subset of the edges and / or processing several edges in bulk. The work of [Kle94] assigns each edge to one of the endpoints so that the edges assigned to a changing cluster can be updated efficiently. However, edges incident to a changing cluster that is not the cluster they were assigned to have to be handled individually. The resulting running time is $O(n\sqrt{m} \log n)$. The algorithm in [GGW98] maintains only a small set of "best" (i.e., smallest amount of slack remaining) edges for each cluster. Their approach requires this set of best edges to be updated after a certain number of merge or deactivation events, which leads to a running time of $O(n^2 + n\sqrt{m \log \log n})$.

We build on the approach described in [CHLP01], which introduces *dynamic edge splitting*. This approach converts active edges with two active endpoints to semi-active edges by effectively inserting an inactive sentinel node at the midpoint of the edge (the sentinels nodes do not need to be maintained explicitly as part of the graph). This ensures that all edges have at most one active endpoint and hence the remaining slack values of all edges with currently growing moats decrease at the same rate. When the algorithm encounters a merge event with a sentinel node, the sentinel node is removed and a new sentinel node is added at the midpoint of the remaining part of the edge. Therefore, the distance remaining on an edge e halves after every merge event involving a sentinel node on edge e.

The authors show that splitting an edge $O(k \log n)$ time suffices to get an approximation guarantee with an additional $\frac{2}{n^k}$ term, i.e., the cost of the tree returned by their algorithms is within a factor $2 + \frac{2}{n^k}$ of the optimal (k is a parameter that can be chosen). Combined with a careful choice of data structures, the algorithm runs in $O(km \log^2 n)$ time.

3 Algorithm

We now introduce our fast variant of the GW algorithm. To the best of our knowledge, our algorithm is the first practical implementation of a GW-like algorithm that runs in nearly linear time.

3.1 Overview

On a high level, our algorithm uses a more aggressive and *adaptive* dynamic edge splitting scheme than [CHLP01]: our algorithm moves previously inserted sentinel nodes in order to reach a tight edge constraint quicker than before. By analyzing the precision needed to represent merge and deactivation events in the GW algorithm, we prove that our algorithm runs in $O(dm \log n)$ time, where d is the number of bits used to specify each value in the input. For constant bit precision d (as is often the case in practical applications) our algorithm hence has a running time of $O(m \log n)$. Furthermore, our algorithm achieves the approximation guarantee (1) exactly without the additional $\frac{2}{n^k}$ term. From an empirical point of view, our more aggressive splitting scheme produces only very few additional edge pieces; the number of edge events processed is very close to $2m$, the number of edge events initially created. We demonstrate this empirical benefit in our experiments (see Section 5).

3.2 Detailed description

Similar to [CHLP01], our algorithm divides each edge $e = (u, v)$ into two *edge parts* e_u and e_v corresponding to the endpoints u and v. We say an edge part p is *active* if its endpoint is in an active cluster, otherwise the edge part p is *inactive*. The key advantage of this approach over considering entire edges is that *all active edge parts always grow at the same rate*. For each edge part p, we also maintain an *event value* $\mu(p)$. This event value is the total amount that the moats on edge part p are allowed to grow until the next event for this edge occurs. In order to ensure that the moats growing on the two corresponding edge parts e_u and e_v never overlap, we always set the event values so that $\mu(e_u) + \mu(e_v) = c(e)$. As for edges, we define the remaining slack of edge part e_u as $\mu(e_u) - \sum_{C \in \mathscr{C}} y_C$, where $\mathscr{C}$ is the set of clusters containing node u.

We say that an *edge event* occurs when an edge part has zero slack remaining. However, this does not necessarily mean that the corresponding edge constraint has become tight as the edge event might be “stale” since the other edge parts has become inactive and stopped growing since the last time the edge event

was updated. Nevertheless, we will be able to show that the total number of edge events to be processed over the course of the algorithm is small. Note that we can find the next edge event by looking at the edge events with smallest remaining slack values in their clusters. This is an important property because it allows us to organize the edge parts in an efficient manner. In particular, we maintain a priority queue Q_C for each cluster C that contains the edge parts with endpoint in C, sorted by the time at which the next event on each edge part occurs. Furthermore, we arrange the cluster priority queues in an overall priority queue resulting in a "heap of heaps" data structure. This data structure allows us to quickly locate the next edge event and perform the necessary updates after cluster deactivation or merge events.

Algorithm 2 Fast variant of the GW algorithm.

```
 1: function PCSTFast(edges, costs, prizes)
 2:     InitPCST(edges, costs, prizes)
 3:     t ← 0                                                    ▷ Current time
 4:     α ← n                                       ▷ Number of active clusters
 5:     while α > 1 do
 6:         ▷ Returns event time and corresponding edge part
 7:         (t_e, p_u) ← GetNextEdgeEvent()
 8:         ▷ Returns event time and corresponding cluster
 9:         (t_c, C) ← GetNextClusterEvent()
10:         if t_e < t_c then
11:             t ← t_e
12:             RemoveNextEdgeEvent()
13:             p_v ← GetOtherEdgePart(p_u)
14:             ▷ GetSumOnEdgePart returns the current moat sum on the edge part
15:             ▷ p_u and the maximal cluster containing u
16:             (s, C_u) ← GetSumOnEdgePart(p_u)
17:             (s', C_v) ← GetSumOnEdgePart(p_v)
18:             r ← GetEdgeCost(p_u) − s − s'          ▷ Remaining amount on the edge
19:             if C_u = C_v then   ▷ The two endpoints are already in the same cluster
20:                 continue                          ▷ Skip to beginning of while-loop
21:             if r = 0 then
22:                 MergeClusters(C_u, C_v)
23:             else
24:                 GenerateNewEdgeEvents(p_u, p_v)
25:         else
26:             t ← t_c
27:             RemoveNextClusterEvent()
28:             DeactivateCluster(C)
29:             α ← α − 1
30:     StrongPruning()
```

In addition to the edge events, we also maintain a priority queue of *cluster events*. This priority queue contains each active cluster with the time at

which the corresponding cluster constraint becomes tight. Using these definitions, we can now state the high-level structure of our algorithm in pseudo code (see Algorithm 2) and then describe the two subroutines MergeClusters and GenerateNewEdgeEvents in more detail.

MergeClusters(C_u, C_v) : As a first step, we mark C_u and C_v as inactive and remove them from the priority queue keeping track of cluster deactivation events. Furthermore, we remove the priority queues Q_{C_u} and Q_{C_v} from the heap of heaps for edge events. Before we merge the heaps of C_u and C_v, we have to ensure that both heaps contain edge events on the "global" time frame. If C_u (or C_v) is inactive since time t' when the merge occurs, the edge event times in Q_{C_u} will have become "stale" because the moat on edge parts incident to C_u did not grow since t'. In order to correct for this offset and bring the keys in Q_{C_u} back to the global time frame, we first increase all keys in Q_{C_u} by $t - t'$. Then, we merge Q_{C_u} and Q_{C_v}, which results in the heap for the new merged cluster. Finally, we insert the new heap into the heap of heaps and add a new entry to the cluster deactivation heap.

GenerateNewEdgeEvents(p_u, p_v) : This function is invoked when an edge event occurs, but the corresponding edge constraint is not yet tight. Since the edge part p_u has no slack left, this means that there is slack remaining on p_v. Let $\mathscr{C}_u$ and $\mathscr{C}_v$ be the set of clusters containing u and v, respectively. Then $r = c(e) - \sum_{C \in \mathscr{C}_u \cup \mathscr{C}_v} y_C$ is the length of the part of edge e not covered by moats yet. We distinguish two cases:

1. The cluster containing the endpoint v is active.
   Since both endpoints are active, we expect both edge parts to grow at the same rate until they meet and the edge constraint becomes tight. Therefore, we set the new event values to $\mu(p_u) = \sum_{C \in \mathscr{C}_u} + \frac{r}{2}$ and $\mu(p_v) = \sum_{C \in \mathscr{C}_v} + \frac{r}{2}$. Note that this maintains the invariant $\mu(p_u) + \mu(p_v) = c(e)$. Using the new event values for p_u and p_v, we update the priority queues Q_{C_u} and Q_{C_v} accordingly and then also update the heap of heaps.
2. The cluster containing the endpoint v is inactive.
   In this case, we assume that v stays inactive until the moat growing on edge part p_u makes the edge constraint for e tight. Hence, we set the new event values to $\mu(p_u) = \sum_{C \in \mathscr{C}_u} + r$ and $\mu(p_v) = \sum_{C \in \mathscr{C}_v}$. As in the previous case, this maintains the invariant $\mu(p_u) + \mu(p_v) = c(e)$ and we update the relevant heaps accordingly. It is worth noting our setting of $\mu(p_v)$ reduces the slack for p_v to zero. This ensures that as soon as the cluster C_v becomes active again, the edge event for p_v will be processed next.

Crucially, in GenerateNewEdgeEvents, we set the new event values for p_u and p_v so that the next edge event on e would merge the clusters C_u and C_v, *assuming both clusters maintain their current activity status.* If one of the two clusters changes its activity status, this will not hold:

1. If both clusters were active and cluster C_u has become inactive since then, the next event on edge e will be part p_v reaching the common midpoint.

However, due to the deactivation of C_u, the edge part p_u will not have reached the common midpoint yet.

2. If C_v was inactive and becomes active before the edge event for p_u occurs, the edge event for p_v will also immediately occur after the activation for C_v. At this time, the moat on p_u has not reached the new, size-0 moat of C_v, and thus the edge constraint is not tight.

However, in the next section we show that if all input values are specified with d bits of precision then at most $O(d)$ edge events can occur per edge. Moreover, even in the general case our experiments in Section 5 show that the pathological cases described above occur very rarely in practice. In most instances, only two edge events are processed per edge on average.

4 Analysis

We now study the theoretical properties of our algorithm PCSTFast. Note that by construction, the result of our algorithm exactly matches the output of GWAlgorithm and hence also satisfies guarantee (1). Therefore, we refer the reader to [FFFdP10,JMP00] for an analysis of the GW scheme with strong pruning and focus on the running time bounds here.

First, we establish the following structural result for the growth stage of the GW algorithm (the "exact" algorithm GWAlgorithm, not yet PCSTFast). Informally, we show that a single additional bit of precision suffices to exactly represent all important events in the moat growth process.

Theorem 1. *Let all node prizes $\pi(v)$ and edge costs $c(e)$ be even integers. Then all cluster merge and deactivation events occur at integer times.*

Proof. We prove the theorem by induction over the cluster merge and deactivation events occuring in the GW algorithm, sorted by the time at which the events happen. We will show that the updates caused by every event maintain the following invariant:

Induction hypothesis Based on the current state of the algorithm, let t_e be the time at which the edge constraint for edge e becomes tight and t_C be the time at which the cluster constraint for cluster C becomes tight. Then t_e and t_C are integers. Moreover, if the merge event at t_e is a merge event between an active cluster and an inactive cluster C, then $t_e - t_{\text{inactive}(C)}$ is even, where $t_{\text{inactive}(C)}$ is the time at which cluster C became inactive.

Clearly, the induction hypothesis holds at the beginning of the algorithm: all edge costs are even, so $t_e = \frac{c(e)}{2}$ is an integer. Since the node prizes are integers, so are the t_C. The assumption on merge events with inactive clusters trivially holds because there are no inactive clusters at the beginning of the algorithm. Next, we perform the induction step by a case analysis over the possible events:

– **Active-active**: a merge event between two active clusters. Since this event modifies no edge events, we only have to consider the new deactivation event

for the new cluster C. By the induction hypothesis, all events so far have occured at integer times, so all moats have integer size. Since the sum of prizes in C is also an integer, the new cluster constraint becomes tight at an integer time.

– **Active-inactive**: a merge event between an active cluster and an inactive cluster. Let e be the current edge, t_e be the current time, and C be the inactive cluster. The deactivation time for the new cluster is the same as that of the current active cluster, so it is also integer. Since every edge e' incident to C now has a new growing moat, we have to consider the change in the event time for e'. We denote the previous event time of e' with $t'_{e'}$. We distinguish two cases:
  - If the other endpoint of e' is in an active cluster, the part of e' remaining has size $t'_{e'} - t_e$ and e' becomes tight at time $t_e + \frac{t'_{e'} - t_e}{2}$ because e' has two growing moats. We have

    $$t'_{e'} - t_e \;=\; \big(t'_{e'} - t_{\text{inactive}(C)}\big) - \big(t_e - t_{\text{inactive}(C)}\big)\ .$$

    Note that both terms on the right hand side are even by the induction hypothesis, and therefore their difference is also even. Hence the new event time for edge e' is an integer.
  - If the other endpoint of e' is an inactive cluster, say C', we have to show that $t_{e'} - t_{\text{inactive}(C')}$ is even, where $t_{e'}$ is the new edge event time for e'. We consider whether C or C' became inactive last:
    * C became inactive last: from the time at which C became inactive we know that $t'_{e'} - t_{\text{inactive}(C')}$ is even. Moreover, we have that $t_{e'} = t'_{e'} + (t_e - t_{\text{inactive}(C)})$. Since $t_e - t_{\text{inactive}(C)}$ is even by the induction hypothesis, so is $t_{e'} - t_{\text{inactive}(C')}$.
    * C' became inactive last: from the time at which C' became inactive we know that $t'_{e'} - t_{\text{inactive}(C)}$ is even. The time of the new edge event can be written as $t_{e'} = t_e + t'_{e'} - t_{\text{inactive}(C')}$ (an integer by the induction hypothesis), which is equivalent to $t_{e'} - t'_{e'} = t_e - t_{\text{inactive}(C')}$. We now use this equality in the second line of the following derivation:

      $$\begin{aligned} t_{e'} - t_{\text{inactive}(C')} \;&=\; t_{e'} - t'_{e'} + t'_{e'} - t_e + t_e - t_{\text{inactive}(C')} \\ &=\; 2(t_{e'} - t'_{e'}) + t'_{e'} - t_e \\ &=\; 2(t_{e'} - t'_{e'}) + \big(t'_{e'} - t_{\text{inactive}(C)}\big) - \big(t_e - t_{\text{inactive}(C)}\big)\ . \end{aligned}$$

      Since $t_e - t_{\text{inactive}(C)}$ is even by the induction hypothesis, all three terms on the right hand side are even.

– **Cluster deactivation**: Clearly, a deactivation of cluster C leads to no changes in other cluster deactivation times. Moreover, edges incident to C and another inactive cluster will never become tight based on the current state of the algorithm. The only quantities remaining are the edge event times for edges e with another cluster endpoint that is active. Note that up to time t_C, the edge e had two growing moats and t_e was an integer. Therefore, the part of e remaining has length $2(t_e - t_C)$, which is an even

integer. The new value of t_e is $t_C + 2(t_e - t_C)$, and since $t_{\text{inactive}(C)} = t_C$ the induction hypothesis is restored.

Since the induction hypothesis is maintained throughout the algorithm and implies the statement of the theorem, the proof is complete.

We now use this result to show that the number of edge part events occuring in PCSTFast is small.

Corollary 1. *Let all node prizes $\pi(v)$ and edge costs $c(e)$ be specified with d bits of precision. Then the number of edge part events processed in* PCSTFast *is bounded by $O(dm)$.*

Proof. We look at each edge e individually. For every edge part event A on e that does not merge two clusters, the following holds: either A reduces the remaining slack of e by at least a factor of two or the event directly preceeding A reduced the remaining slack on e by at least a factor of two. In the second case, we charge A to the predecessor event of A.

So after $O(d)$ edge parts events on e, the remaining slack on e is at most $\frac{c(e)}{2^d}$. Theorem 1 implies that the minimum time between two cluster merge or deactivation events is $\frac{c(e)}{2^{d+1}}$. So after a constant number of additional edge part events on e, the edge constraint of e must be the next constraint to become tight, which is the last edge part event on e to be processed. Therefore, the total number of edge part events on e is $O(d)$.

We now show that all subroutines in PCSTFast can be implemented in $O(\log n)$ amortized time, which leads to our final bound on the running time.

Theorem 2. *Let all node prizes $\pi(v)$ and edge costs $c(e)$ be specified with d bits of precision. Then* PCSTFast *runs in $O(dm \log n)$ time.*

Proof. The requirements for the priority queue maintaining edge parts events are the standard operations of a mergeable heap data structure, combined with an operation that adds a constant offset to all elements in a heap in $O(\log n)$ amortized time. We can build such a data structure by augmenting a pairing heap [FSST86] with an offset value at each node.[2] Due to space constraints, we omit the details of this construction here. For the outer heap in the heap of heaps and the priority queue containing cluster deactivation events, a standard binomial heap suffices.

We represent the laminar family of clusters in a tree structure: each cluster C is a node, the child nodes are the two clusters that were merged to form C, and the parent is the cluster C was merged into. The initial clusters, i.e., the individual nodes, form the leaves of the tree. By also storing the moat values at each node, the GetSumOnEdgePart operation for edge part p_u can be implemented by traversing the tree from leaf u to the root of its subtree. However, the depth of this tree can be up to $\Omega(n)$. In order to speed up the data structure,

[2] We also choose a pairing heap due to its good performance in practice [SV87,MS91].

we use path compression in essentially the same way as standard union-find data structures. The resulting amortized running time for GetSumOnEdgePart and merging clusters then becomes $O(\log n)$. See [MS08, p. 225] for an anlysis of union-find data structures with path compression only.

This shows that all subroutines in PCSTFast (Algorithm 2) can be implemented to run in $O(\log n)$ amortized time. Since there are at most $O(dm)$ events to be processed in total, the overall running time bound of $O(dm \log n)$ follows.

5 Experiments

We now present the results of our computational experiments. As mentioned in the introduction, we are primarily interested in the running time of our algorithm on large data sets (while maintaing the approximation guarantee of the GW-algorithm). First, we test the performance of our algorithm on the public test of the DIMACS challenge and report the running times and solution values achieved. In order to investigate the effectiveness of our edge splitting heuristics, we then look at the number of edge events processed by our algorithm. Finally, we demonstrate that our algorithm can also solve larger instances fast by reporting its running time on test cases derived from our signal processing applications.

All experiments were conducted on a laptop computer from 2010 (Intel Core i7 with 2.66 GHz, 4 MB of cache, and 8 GB of RAM). We used the Debian GNU/Linux distribution and `g++` 4.8 as compiler with the `-O3` flag. All reported running times are averaged over 11 trials after removing the slowest run. The time spent on reading the input is excluded from the running times stated here, but all time used for setting up data structures in the algorithm is included.

5.1 Public DIMACS challenge instances

Figure 5.1 shows the running time of our algorithm on the public DIMACS instances for the unrooted prize-collecting Steiner tree problem (PCSPG). For a single instance, the maximum running time of our algorithm is roughly 1.3 seconds and most instances are solved significantly faster. The scatter plot also demonstrates the nearly-linear scaling of our running time with respect to the input size.

Due to space constraints, we only show detailed results for the ACTMODPC instances here and refer the reader to Appendix B for the other test case groups. The Appendix also contains results for the RPCST (rooted PCST) and MWCS (maximum weight connected subgraph) instances provided by DIMACS. The "GW LB" column in the results table is a simple lower bound derived from Equation 1. For some of the test case groups, significantly stronger bounds are avaible. We refer refer the reader to [CRR01] for tables containing the JMP and CRR lower bounds. As Table 1 shows, the GW scheme achieves a very good approximation ratio on these instances, which were derived from problems in biological network analysis.

Table 1: Results for the PCSPG-ACTMODPC test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
HCMV	3863	29293	43.130	7375.706	7366.34212	0.13 %
drosophila001	5226	93394	122.449	8286.432	8286.432	0.00 %
drosophila005	5226	93394	160.624	8208.637	7787.0005	5.41 %
drosophila0075	5226	93394	174.134	8136.466	7622.126	6.75 %
lymphoma	2034	7756	5.309	3382.776	3277.43175	3.21 %
metabol_expr_mice_1	3523	4345	6.985	11405.49	10817.35	5.44 %
metabol_expr_mice_2	3514	4332	6.775	16261.0	16021.0	1.50 %
metabol_expr_mice_3	2853	3335	5.505	17052.0	16392.0	4.03 %

5.2 Effectiveness of our edge splitting heuristic

As pointed out in our running time analysis in Section 4, the number of edge part events determines the overall running time of our algorithm. For input values specified with d bits of precision, our analysis shows that the algorithm encounters at most $O(d)$ events per edge. In order to get a better understanding of our empirical performance, we now look at the number of edge part events encountered by our algorithm (see Figure 5.2).

The scatter plots show that for all instances, the average number of events per edge is less than 3. These results demonstrate the effectiveness of our more adaptive edge splitting heuristics. Moreover, the number of edge events encountered explains the small running times on the large i640 instances in Figure 5.1.

5.3 The HAND instances

We have produced the HAND instances from our application of the PCST problem in signal processing. In this application, we are interested in finding a sparse representation of an input image. Therefore, our graph is a grid graph with one node for each pixel of the input image. The node prizes correspond to pixel intensities and we choose the edge costs so that the resulting solution contains roughly the desired number of pixels (all edges have the same cost).

We have divided our instances into four sets: two of them are smaller images (around 40,000 pixels) and the other two larger images (around 160,000 pixels). Within each size category, one image is derived from the handwritten text “DIMACS” and the other from the handwritten text “ICERM”. Figure 3 in Appendix A contains two example images. The results in Figures 5.1 and 5.2 show that the HAND instances are among the most challenging instances for our algorithm, both due to their size and due to their large number of edge events. Moreover, the performance of our algorithm varies widely even for the same image size. Most likely, this is due to the different noise levels in the images.

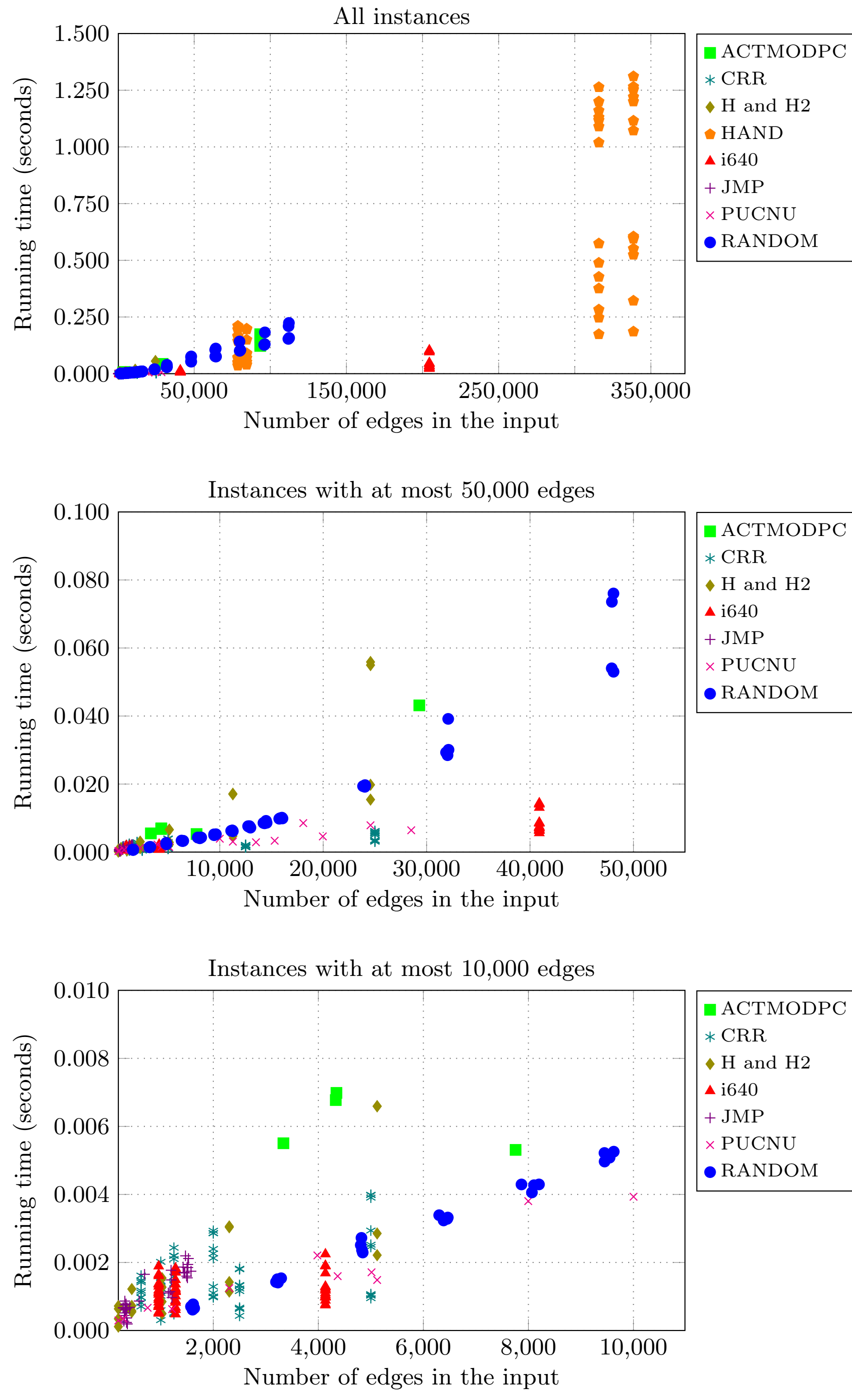

Fig. 1. Running times for the PCSPG instances of the DIMACS challenge. Each color corresponds to one test case group.

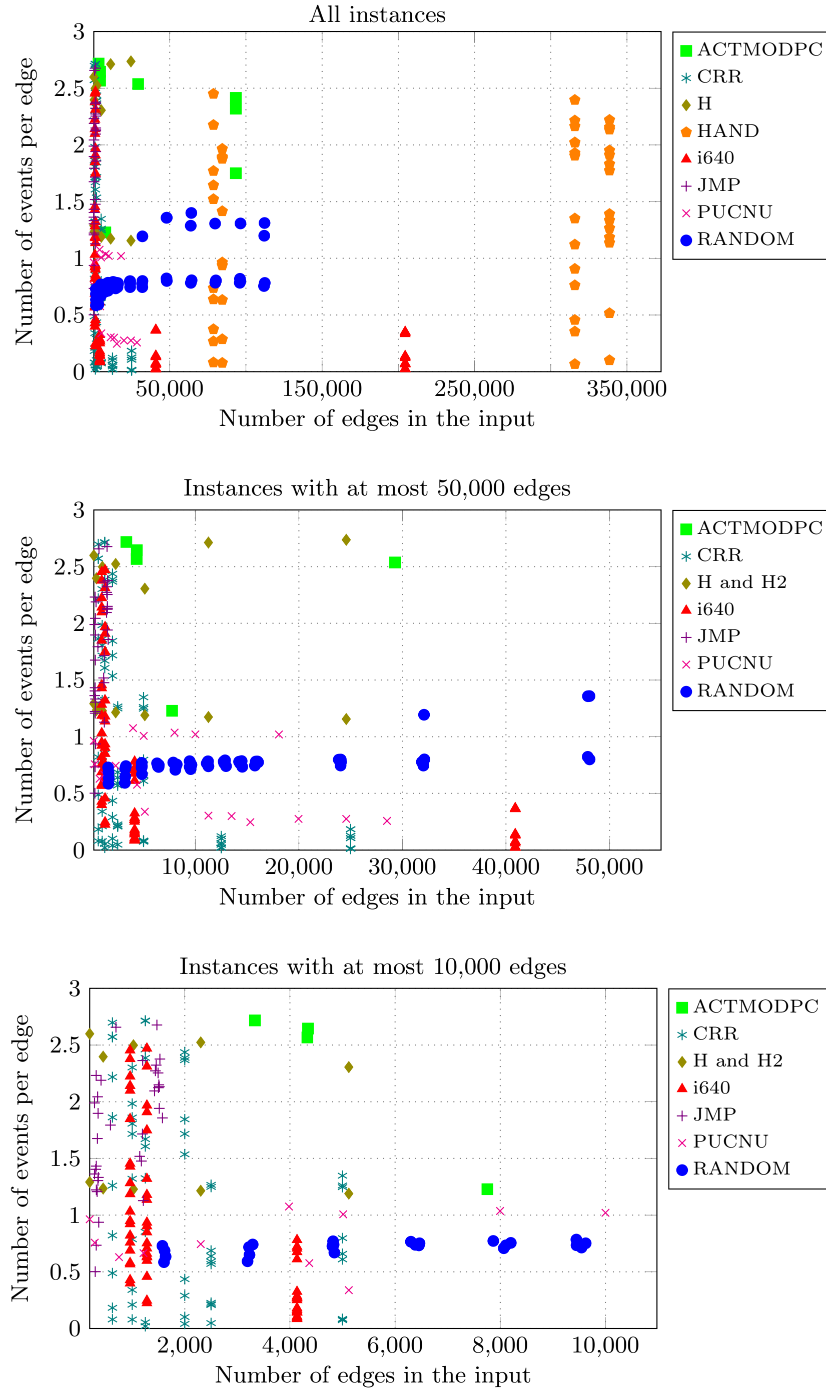

Fig. 2. Average number of edge events processed per edge for the PCSPG instances of the DIMACS challenge. Each color corresponds to one test case group.

References

ABHK11. Aaron Archer, MohammadHossein Bateni, MohammadTaghi Hajiaghayi, and Howard Karloff. Improved approximation algorithms for prize-collecting steiner tree and tsp. *SIAM J. Comput.*, 40(2):309–332, 2011.

AKR91. Ajit Agrawal, Philip Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized steiner problem on networks. In *STOC*, 1991.

BGSLW93. D. Bienstock, M. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 1993.

CC02. Miroslav Chlebík and Janka Chlebíková. Approximation hardness of the steiner tree problem on graphs. In *SWAT*, 2002.

CHLP01. R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. A faster implementation of the Goemans-Williamson clustering algorithm. In *SODA*, 2001.

CRR01. S. Canuto, M. Resende, and C. Ribeiro. Local search with perturbations for the prize-collecting steiner tree problem in graphs. *Networks*, 38(1):50–58, 2001.

CRW04. F. Chudak, T. Roughgarden, and D. Williamson. Approximate k-MSTs and k-Steiner trees via the primal-dual method and Lagrangean relaxation. *Mathematical Programming*, 2004.

DKR+08. Marcus T. Dittrich, Gunnar W. Klau, Andreas Rosenwald, Thomas Dandekar, and Tobias Müller. Identifying functional modules in protein–protein interaction networks: an integrated exact approach. *Bioinformatics*, 24(13):i223–i231, 2008.

FFFdP10. P. Feofiloff, C. G. Fernandes, C. E. Ferreira, and J. Coelho de Pina. A note on Johnson, Minkoff and Phillips' algorithm for the prize-collecting Steiner tree problem. *CoRR*, abs/1004.1437, 2010.

FSST86. Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

GGW98. Harold Gabow, Michel Goemans, and David Williamson. An efficient approximation algorithm for the survivable network design problem. *Mathematical Programming*, 82(1-2):13–40, 1998.

GW95. M. Goemans and D. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 1995.

HK13. M. Hauptmann and M. Karpinski. A compendium on Steiner tree problems. `http://theory.cs.uni-bonn.de/info5/steinerkompendium/netcompendium.pdf`, 2013.

IOSS02. Trey Ideker, Owen Ozier, Benno Schwikowski, and Andrew F. Siegel. Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics*, 18(suppl 1):S233–S240, 2002.

JMP00. D. Johnson, M. Minkoff, and S. Phillips. The prize collecting Steiner tree problem: Theory and practice. In *SODA*, 2000.

JP95. M. Jünger and W. Pulleyblank. New primal and dual matching heuristics. *Algorithmica*, 13(4):357–380, 1995.

Kar72. Richard Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. 1972.

Kle94. Philip N. Klein. A data structure for bicategories, with application to speeding up an approximation algorithm. *Inf. Process. Lett.*, 52(6):303–307, 1994.

MS91. Bernard. Moret and Henry Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 1991.

MS08. Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

RAGT14. Polina Rozenshtein, Aris Anagnostopoulos, Aristides Gionis, and Nikolaj Tatti. Event detection in activity networks. In *KDD*, 2014.

SV87. John T. Stasko and Jeffrey Scott Vitter. Pairing heaps: Experiments and analysis. *Commun. ACM*, 30(3):234–249, 1987.

WS11. D. Williamson and D. Shmoys. *The Design of Approximation Algorithms*. 2011.

A Examples of HAND instances in the DIMACS challenge

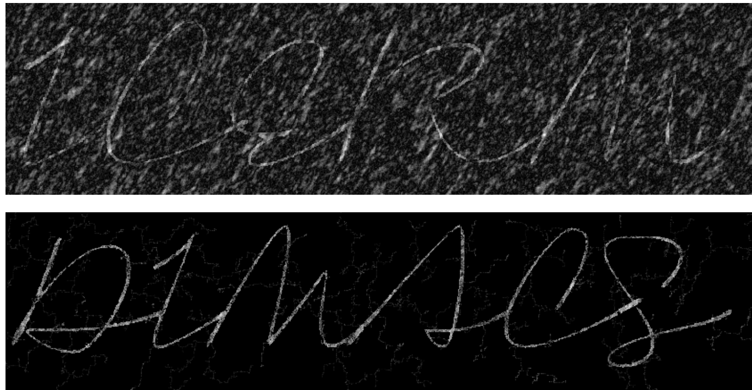

Fig. 3. Two input images for our PCST instances.

B Further experimental results

B.1 DIMACS test cases: PCSPG

Table 2: Results for the PCSPG-H test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
hc10p	1024	5120	6.589	73875	56356	31.09 %
hc10u	1024	5120	2.853	747	709	5.36 %
hc11p	2048	11264	17.170	147004	111661	31.65 %
hc11u	2048	11264	6.328	1524	1387	9.88 %
hc12p	4096	24576	54.965	293413	222606	31.81 %
hc12u	4096	24576	19.794	3027	2720	11.29 %
hc6p	64	192	0.637	4483	3723	20.41 %
hc6u	64	192	0.343	44	43	2.33 %
hc7p	128	448	0.545	9108	7696	18.35 %
hc7u	128	448	0.724	90	89	1.12 %
hc8p	256	1024	1.540	18452	17393	6.09 %
hc8u	256	1024	0.848	187	170	10.00 %
hc9p	512	2304	3.061	35759	26587	34.50 %
hc9u	512	2304	1.424	390	387	0.78 %

Table 3: Results for the PCSPG-H2 test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
hc10p2	1024	5120	6.596	74648	50588	47.56 %
hc10u2	1024	5120	2.215	505	468	7.91 %
hc11p2	2048	11264	16.983	149814	99983	49.84 %
hc11u2	2048	11264	4.718	1011	974	3.80 %
hc12p2	4096	24576	55.889	298874	205477	45.45 %
hc12u2	4096	24576	15.435	2040	1973	3.40 %
hc6p2	64	192	0.720	4468	3911	14.24 %
hc6u2	64	192	0.114	24	23	4.35 %
hc7p2	128	448	1.219	9163	6738	35.99 %
hc7u2	128	448	0.578	59	58	1.72 %
hc8p2	256	1024	1.273	18915	13719	37.87 %
hc8u2	256	1024	0.495	121	119	1.68 %
hc9p2	512	2304	3.034	37379	25529	46.42 %
hc9u2	512	2304	1.144	245	243	0.82 %

Table 4: Results for the PCSPG-HAND test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
handbd01	169800	338551	1072.303	729.3192	727.011784	0.32 %

Continued on next page

Table 4: Results for the PCSPG-HAND test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
handbd02	169800	338551	321.710	297.6341	152.068219	95.72 %
handbd03	169800	338551	1220.579	135.0745	135.03483	0.03 %
handbd04	169800	338551	595.249	1831.154	1295.9257	41.30 %
handbd05	169800	338551	1253.030	105.4757	105.4440025	0.03 %
handbd06	169800	338551	604.882	1541.342	1043.70125	47.68 %
handbd07	169800	338551	1311.182	77.86358	77.816032	0.06 %
handbd08	169800	338551	551.395	1378.553	908.45005	51.75 %
handbd09	169800	338551	1115.563	62.71914	62.683946	0.06 %
handbd10	169800	338551	524.504	1144.122	714.69975	60.08 %
handbd11	169800	338551	1265.249	46.77253	46.766642	0.01 %
handbd12	169800	338551	598.329	321.4391	192.87845	66.65 %
handbd13	169800	338551	1200.258	13.21217	13.1728175	0.30 %
handbd14	169800	338551	185.648	4379.104	4339.046755	0.92 %
handbi01	158400	315808	1019.333	1358.901	1356.314865	0.19 %
handbi02	158400	315808	282.622	533.3406	271.81917	96.21 %
handbi03	158400	315808	1199.880	243.1487	242.95564	0.08 %
handbi04	158400	315808	574.111	3240.382	2277.508	42.28 %
handbi05	158400	315808	1130.282	184.4673	184.4246975	0.02 %
handbi06	158400	315808	488.761	2951.265	2001.166	47.48 %
handbi07	158400	315808	1117.507	150.981	150.901424	0.05 %
handbi08	158400	315808	426.857	2284.602	1476.759	54.70 %
handbi09	158400	315808	1159.863	107.77	107.74168	0.03 %
handbi10	158400	315808	375.769	1881.135	1135.782	65.62 %
handbi11	158400	315808	1263.395	68.94471	68.931683	0.02 %
handbi12	158400	315808	246.903	138.3032	78.84144	75.42 %
handbi13	158400	315808	1090.012	4.343122	4.190887	3.63 %
handbi14	158400	315808	173.984	7881.839	3960.48994	99.01 %
handsd01	42500	84475	149.608	171.6532	171.2056625	0.26 %
handsd02	42500	84475	69.167	160.8896	84.348027	90.74 %
handsd03	42500	84475	195.518	31.30726	31.279994	0.09 %
handsd04	42500	84475	89.993	496.8705	324.9625	52.90 %
handsd05	42500	84475	196.203	21.93869	21.92235	0.07 %
handsd06	42500	84475	86.736	281.1465	172.54092	62.94 %
handsd07	42500	84475	198.438	11.80412	11.7978105	0.05 %
handsd08	42500	84475	51.532	143.3416	81.84537	75.14 %
handsd09	42500	84475	197.207	3.818683	3.817631	0.03 %
handsd10	42500	84475	39.534	1034.767	1025.31081	0.92 %
handsi01	39600	78704	143.628	295.5152	294.97325	0.18 %
handsi02	39600	78704	50.362	125.5538	64.263529	95.37 %
handsi03	39600	78704	210.602	56.15379	55.959944	0.35 %
handsi04	39600	78704	67.883	729.2789	468.91005	55.53 %
handsi05	39600	78704	194.393	35.04351	35.037619	0.02 %

Continued on next page

Table 4: Results for the PCSPG-HAND test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
handsi06	39600	78704	71.774	454.6696	277.26887	63.98 %
handsi07	39600	78704	166.329	18.41013	18.40239	0.04 %
handsi08	39600	78704	46.225	229.7032	129.04949	78.00 %
handsi09	39600	78704	162.554	5.962253	5.948498	0.23 %
handsi10	39600	78704	35.391	1805.427	966.4394	86.81 %

Table 5: Results for the PCSPG-CRR test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
C01-A	500	625	0.705	18	18	0.00 %
C01-B	500	625	0.969	88	44	100.00 %
C02-A	500	625	0.813	50	50	0.00 %
C02-B	500	625	1.116	141	84	67.86 %
C03-A	500	625	0.976	414	407	1.72 %
C03-B	500	625	1.418	763	395	93.16 %
C04-A	500	625	1.464	626	596	5.03 %
C04-B	500	625	1.173	1096	550	99.27 %
C05-A	500	625	1.472	1088	837	29.99 %
C05-B	500	625	1.614	1550	806	92.31 %
C06-A	500	1000	0.305	18	18	0.00 %
C06-B	500	1000	0.597	60	30	100.00 %
C07-A	500	1000	1.022	50	50	0.00 %
C07-B	500	1000	1.161	114	57	100.00 %
C08-A	500	1000	1.416	370	286	29.37 %
C08-B	500	1000	1.457	524	273	91.94 %
C09-A	500	1000	1.104	544	403	34.99 %
C09-B	500	1000	1.404	714	371	92.45 %
C10-A	500	1000	1.455	874	627	39.39 %
C10-B	500	1000	2.018	1091	559	95.17 %
C11-A	500	2500	0.430	18	18	0.00 %
C11-B	500	2500	0.672	37	19	94.74 %
C12-A	500	2500	0.637	39	34	14.71 %
C12-B	500	2500	0.635	48	24	100.00 %
C13-A	500	2500	1.338	250	151	65.56 %
C13-B	500	2500	1.235	278	139	100.00 %
C14-A	500	2500	1.320	309	171	80.70 %
C14-B	500	2500	1.160	333	176	89.20 %
C15-A	500	2500	1.817	517	304	70.07 %
C15-B	500	2500	1.800	569	288	97.57 %
C16-A	500	12500	1.378	12	6	100.00 %

Continued on next page

Table 5: Results for the PCSPG-CRR test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
C16-B	500	12500	2.043	12	6	100.00 %
C17-A	500	12500	1.515	19	10	90.00 %
C17-B	500	12500	1.535	19	10	90.00 %
C18-A	500	12500	1.516	124	68	82.35 %
C18-B	500	12500	2.103	128	64	100.00 %
C19-A	500	12500	1.593	157	85	84.71 %
C19-B	500	12500	1.636	159	80	98.75 %
C20-A	500	12500	2.107	267	145	84.14 %
C20-B	500	12500	2.019	268	135	98.52 %
D01-A	1000	1250	0.482	18	18	0.00 %
D01-B	1000	1250	1.380	107	54	98.15 %
D02-A	1000	1250	0.520	50	50	0.00 %
D02-B	1000	1250	1.124	237	132	79.55 %
D03-A	1000	1250	1.336	809	723	11.89 %
D03-B	1000	1250	2.193	1580	831	90.13 %
D04-A	1000	1250	1.586	1210	1081	11.93 %
D04-B	1000	1250	2.203	1941	1043	86.10 %
D05-A	1000	1250	2.101	2171	1767	22.86 %
D05-B	1000	1250	2.433	3196	1638	95.12 %
D06-A	1000	2000	1.100	18	18	0.00 %
D06-B	1000	2000	0.975	73	37	97.30 %
D07-A	1000	2000	1.000	50	50	0.00 %
D07-B	1000	2000	1.289	116	58	100.00 %
D08-A	1000	2000	2.129	776	642	20.87 %
D08-B	1000	2000	2.265	1100	566	94.35 %
D09-A	1000	2000	2.397	1118	867	28.95 %
D09-B	1000	2000	2.857	1516	769	97.14 %
D10-A	1000	2000	2.866	1707	1171	45.77 %
D10-B	1000	2000	2.933	2130	1091	95.23 %
D11-A	1000	5000	0.951	18	18	0.00 %
D11-B	1000	5000	1.023	31	16	93.75 %
D12-A	1000	5000	1.070	42	27	55.56 %
D12-B	1000	5000	1.046	44	22	100.00 %
D13-A	1000	5000	2.457	475	293	62.12 %
D13-B	1000	5000	2.522	518	265	95.47 %
D14-A	1000	5000	2.938	632	369	71.27 %
D14-B	1000	5000	3.991	702	359	95.54 %
D15-A	1000	5000	3.906	1081	630	71.59 %
D15-B	1000	5000	3.981	1146	579	97.93 %
D16-A	1000	25000	3.355	13	10	30.00 %
D16-B	1000	25000	3.532	15	8	87.50 %
D17-A	1000	25000	3.059	25	16	56.25 %

Continued on next page

Table 5: Results for the PCSPG-CRR test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
D17-B	1000	25000	3.162	25	13	92.31 %
D18-A	1000	25000	5.034	248	139	78.42 %
D18-B	1000	25000	5.481	255	130	96.15 %
D19-A	1000	25000	6.057	339	182	86.26 %
D19-B	1000	25000	6.221	347	174	99.43 %
D20-A	1000	25000	5.720	540	287	88.15 %
D20-B	1000	25000	5.438	541	271	99.63 %

Table 6: Results for the PCSPG-i640 test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
i640-001	640	960	0.537	3414	2417	41.25 %
i640-002	640	960	0.697	2801	2301	21.73 %
i640-003	640	960	0.502	2740	2740	0.00 %
i640-004	640	960	0.704	3925	2437	61.06 %
i640-005	640	960	0.706	3288	3288	0.00 %
i640-011	640	4135	0.745	2598	1675	55.10 %
i640-012	640	4135	0.774	2457	1385	77.40 %
i640-013	640	4135	0.975	2569	1383	85.76 %
i640-014	640	4135	1.102	2667	1334	99.93 %
i640-015	640	4135	0.885	2954	1477	100.00 %
i640-021	640	204480	22.889	2067	1182	74.87 %
i640-022	640	204480	29.455	2201	1173	87.64 %
i640-023	640	204480	22.389	2072	1188	74.41 %
i640-024	640	204480	28.852	1959	1082	81.05 %
i640-025	640	204480	25.747	2055	1028	99.90 %
i640-031	640	1280	0.679	2632	2632	0.00 %
i640-032	640	1280	0.480	2107	1506	39.91 %
i640-033	640	1280	0.609	2900	2900	0.00 %
i640-034	640	1280	0.827	3188	1946	63.82 %
i640-035	640	1280	0.875	2870	2019	42.15 %
i640-041	640	40896	5.612	1994	1073	85.83 %
i640-042	640	40896	6.457	2285	1221	87.14 %
i640-043	640	40896	6.285	1750	957	82.86 %
i640-044	640	40896	6.628	2130	1155	84.42 %
i640-045	640	40896	6.466	1932	1122	72.19 %
i640-101	640	960	0.744	9278	5141	80.47 %
i640-102	640	960	1.056	8575	6080	41.04 %
i640-103	640	960	1.131	8609	4935	74.45 %
i640-104	640	960	0.704	8068	4804	67.94 %

Continued on next page

Table 6: Results for the PCSPG-i640 test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
i640-105	640	960	0.831	10060	6636	51.60 %
i640-111	640	4135	0.978	6849	3999	71.27 %
i640-112	640	4135	1.182	8218	4439	85.13 %
i640-113	640	4135	1.045	7610	4307	76.69 %
i640-114	640	4135	1.000	7271	4362	66.69 %
i640-115	640	4135	0.992	7512	4402	70.65 %
i640-121	640	204480	35.409	6002	3239	85.30 %
i640-122	640	204480	36.399	6498	3439	88.95 %
i640-123	640	204480	36.208	6184	3286	88.19 %
i640-124	640	204480	35.688	6317	3559	77.49 %
i640-125	640	204480	36.076	6328	3416	85.25 %
i640-131	640	1280	0.823	7874	4521	74.17 %
i640-132	640	1280	1.143	8758	4813	81.97 %
i640-133	640	1280	1.011	8253	4859	69.85 %
i640-134	640	1280	1.359	7440	4322	72.14 %
i640-135	640	1280	1.172	7730	4627	67.06 %
i640-141	640	40896	7.253	7052	3654	92.99 %
i640-142	640	40896	7.195	6239	3442	81.26 %
i640-143	640	40896	7.029	5747	3074	86.96 %
i640-144	640	40896	7.476	6198	3455	79.39 %
i640-145	640	40896	7.155	6512	3408	91.08 %
i640-201	640	960	1.186	15953	9601	66.16 %
i640-202	640	960	1.102	17715	9940	78.22 %
i640-203	640	960	0.954	16230	9849	64.79 %
i640-204	640	960	0.905	14577	7955	83.24 %
i640-205	640	960	1.378	18241	10131	80.05 %
i640-211	640	4135	1.285	15124	8592	76.02 %
i640-212	640	4135	1.186	14377	8011	79.47 %
i640-213	640	4135	1.236	14522	7551	92.32 %
i640-214	640	4135	1.295	14571	7894	84.58 %
i640-215	640	4135	1.187	13509	7363	83.47 %
i640-221	640	204480	44.659	11630	6277	85.28 %
i640-222	640	204480	46.770	12692	6772	87.42 %
i640-223	640	204480	47.821	13065	7001	86.62 %
i640-224	640	204480	45.171	12492	6560	90.43 %
i640-225	640	204480	45.244	11845	6649	78.15 %
i640-231	640	1280	1.234	17317	9693	78.65 %
i640-232	640	1280	0.878	15708	9006	74.42 %
i640-233	640	1280	1.139	15172	8322	82.31 %
i640-234	640	1280	1.138	16859	9386	79.62 %
i640-235	640	1280	1.185	15660	8912	75.72 %
i640-241	640	40896	8.648	13460	7326	83.73 %

Continued on next page

Table 6: Results for the PCSPG-i640 test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
i640-242	640	40896	8.628	12789	6631	92.87 %
i640-243	640	40896	8.594	12843	6834	87.93 %
i640-244	640	40896	8.434	12343	6482	90.42 %
i640-245	640	40896	8.658	13221	6773	95.20 %
i640-301	640	960	1.646	49691	26620	86.67 %
i640-302	640	960	1.890	48225	27375	76.16 %
i640-303	640	960	1.605	49353	26045	89.49 %
i640-304	640	960	1.360	48917	27629	77.05 %
i640-305	640	960	1.297	49390	26417	86.96 %
i640-311	640	4135	1.907	43840	23732	84.73 %
i640-312	640	4135	1.905	43316	22876	89.35 %
i640-313	640	4135	1.892	42943	22944	87.16 %
i640-314	640	4135	1.686	43106	23129	86.37 %
i640-315	640	4135	2.246	43114	22521	91.44 %
i640-321	640	204480	99.990	41318	21811	89.44 %
i640-322	640	204480	101.721	40719	21770	87.04 %
i640-323	640	204480	99.632	40598	21511	88.73 %
i640-324	640	204480	103.789	41315	22084	87.08 %
i640-325	640	204480	95.049	40249	22023	82.76 %
i640-331	640	1280	1.724	47675	26028	83.17 %
i640-332	640	1280	1.491	48761	25955	87.87 %
i640-333	640	1280	1.708	48135	26606	80.92 %
i640-334	640	1280	1.808	49680	27376	81.47 %
i640-335	640	1280	1.831	47730	26625	79.27 %
i640-341	640	40896	14.047	41940	22498	86.42 %
i640-342	640	40896	14.301	41940	22208	88.85 %
i640-343	640	40896	13.074	42351	22920	84.78 %
i640-344	640	40896	14.382	42186	22769	85.28 %
i640-345	640	40896	14.075	42369	22757	86.18 %

Table 7: Results for the PCSPG-JMP test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
K100	100	351	0.369	135511	135511	0.00 %
K100.1	100	348	0.237	124108	124108	0.00 %
K100.10	100	319	0.610	133567	133567	0.00 %
K100.2	100	339	0.780	200262	135880	47.38 %
K100.3	100	407	0.682	115953	115953	0.00 %
K100.4	100	364	0.193	87498	78460	11.52 %
K100.5	100	358	0.667	119078	117395	1.43 %

Continued on next page

Table 7: Results for the PCSPG-JMP test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
K100.6	100	307	0.442	132886	127635	4.11 %
K100.7	100	315	0.865	172457	166381	3.65 %
K100.8	100	343	0.650	215616	201641	6.93 %
K100.9	100	333	0.612	122917	122917	0.00 %
K200	200	691	1.652	329211	307882	6.93 %
K400	400	1515	1.851	350093	333889	4.85 %
K400.1	400	1470	2.196	490771	456292	7.56 %
K400.10	400	1507	1.549	406658	314225	29.42 %
K400.2	400	1527	2.116	478238	430403	11.11 %
K400.3	400	1492	1.597	415328	405978	2.30 %
K400.4	400	1426	1.865	394046	347070	13.54 %
K400.5	400	1456	1.694	529581	503887	5.10 %
K400.6	400	1576	1.742	376830	358253	5.19 %
K400.7	400	1442	1.706	476299	352046	35.29 %
K400.8	400	1516	1.941	418614	411478	1.73 %
K400.9	400	1500	1.852	385622	342515	12.59 %
P100	100	317	0.752	823026	456106	80.45 %
P100.1	100	284	0.660	966020	556078	73.72 %
P100.2	100	297	0.274	429687	255201	68.37 %
P100.3	100	316	0.517	676158	393104	72.00 %
P100.4	100	284	0.673	841872	578481	45.53 %
P200	200	587	0.866	1325376	859880	54.13 %
P400	400	1200	1.487	2543645	1605683	58.42 %
P400.1	400	1212	0.967	2877459	1723565	66.95 %
P400.2	400	1196	1.772	2590034	1573583	64.59 %
P400.3	400	1175	1.146	3010150	1939555	55.20 %
P400.4	400	1144	1.104	2942801	1748831	68.27 %

Table 8: Results for the PCSPG-PUCNU test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
bip42nu	1200	3982	2.207	280	229	22.27 %
bip52nu	2200	7997	3.804	282	250	12.80 %
bip62nu	1200	10002	3.933	254	192	32.29 %
bipa2nu	3300	18073	8.524	399	315	26.67 %
bipe2nu	550	5013	1.711	57	42	35.71 %
cc10-2nu	1024	5120	1.485	188	147	27.89 %
cc11-2nu	2048	11263	3.072	352	311	13.18 %
cc12-2nu	4096	24574	7.881	651	517	25.92 %
cc3-10nu	1000	13500	2.891	65	64	1.56 %

Continued on next page

Table 8: Results for the PCSPG-PUCNU test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
cc3-11nu	1331	19965	4.610	91	87	4.60 %
cc3-12nu	1728	28512	6.416	111	109	1.83 %
cc3-4nu	64	288	0.348	11	11	0.00 %
cc3-5nu	125	750	0.667	19	18	5.56 %
cc5-3nu	243	1215	0.629	38	38	0.00 %
cc6-2nu	64	192	0.294	15	15	0.00 %
cc6-3nu	729	4368	1.598	104	82	26.83 %
cc7-3nu	2187	15308	3.336	322	264	21.97 %
cc9-2nu	512	2304	1.245	90	89	1.12 %

Table 9: Results for the PCSPG-RANDOM test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
a0200RandGraph.1.2	200	1636	0.655	124.2196	121.2196	2.47 %
a0200RandGraph.1.5	200	1575	0.702	149.0266	103.02664	44.65 %
a0200RandGraph.2	200	1605	0.613	161.0155	95.01547	69.46 %
a0200RandGraph.3	200	1616	0.770	174.2793	95.2793	82.91 %
a0400RandGraph.1.2	400	3194	1.427	237.323	235.323	0.85 %
a0400RandGraph.1.5	400	3231	1.410	283.864	201.864	40.62 %
a0400RandGraph.2	400	3292	1.536	315.9484	193.44839	63.32 %
a0400RandGraph.3	400	3222	1.508	345.6945	195.69447	76.65 %
a0600RandGraph.1.2	600	4821	2.723	364.1584	361.6584	0.69 %
a0600RandGraph.1.5	600	4845	2.293	427.6309	297.6309	43.68 %
a0600RandGraph.2	600	4831	2.364	475.6437	291.6437	63.09 %
a0600RandGraph.3	600	4808	2.510	518.8675	298.36748	73.90 %
a0800RandGraph.1.2	800	6453	3.282	469.2574	468.2574	0.21 %
a0800RandGraph.1.5	800	6301	3.390	553.5262	392.0262	41.20 %
a0800RandGraph.2	800	6465	3.322	625.9556	385.4556	62.39 %
a0800RandGraph.3	800	6385	3.235	674.171	377.67099	78.51 %
a10000RandGraph.1.2	10000	80298	103.964	6019.544	5998.544	0.35 %
a10000RandGraph.1.5	10000	80288	103.732	7049.384	4993.884	41.16 %
a10000RandGraph.2	10000	79908	100.763	7869.94	4797.44	64.04 %
a10000RandGraph.3	10000	79778	141.979	8617.496	4860.996	77.28 %
a1000RandGraph.1.2	1000	8067	4.059	588.5942	580.5942	1.38 %
a1000RandGraph.1.5	1000	7868	4.294	700.5084	498.0084	40.66 %
a1000RandGraph.2	1000	8201	4.294	775.3939	482.8939	60.57 %
a1000RandGraph.3	1000	8107	4.271	850.9121	484.9121	75.48 %
a12000RandGraph.1.2	12000	96093	127.532	7187.638	7173.638	0.20 %
a12000RandGraph.1.5	12000	96391	129.887	8418.893	5960.393	41.25 %

Continued on next page

Table 9: Results for the PCSPG-RANDOM test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
a12000RandGraph.2	12000	95987	127.042	9367.894	5790.394	61.78 %
a12000RandGraph.3	12000	96449	182.075	10290.55	5819.551	76.83 %
a1200RandGraph.1.2	1200	9448	5.219	713.4155	711.4155	0.28 %
a1200RandGraph.1.5	1200	9625	5.257	846.3853	588.8853	43.73 %
a1200RandGraph.2	1200	9546	5.082	937.3539	584.3539	60.41 %
a1200RandGraph.3	1200	9451	4.967	1032.965	576.9649	79.03 %
a14000RandGraph.1.2	14000	112016	209.721	8401.319	8389.819	0.14 %
a14000RandGraph.1.5	14000	112228	223.811	9873.704	7003.704	40.98 %
a14000RandGraph.2	14000	112369	157.520	10999.81	6775.31	62.35 %
a14000RandGraph.3	14000	111869	153.388	12042.13	6813.632	76.74 %
a1400RandGraph.1.2	1400	11192	6.342	822.5676	812.0676	1.29 %
a1400RandGraph.1.5	1400	11226	6.350	974.4749	710.4749	37.16 %
a1400RandGraph.2	1400	11100	6.263	1091.35	667.3501	63.53 %
a1400RandGraph.3	1400	11263	6.141	1187.916	671.4155	76.93 %
a1600RandGraph.1.2	1600	12869	7.634	955.021	948.021	0.74 %
a1600RandGraph.1.5	1600	12739	7.617	1122.855	800.8553	40.21 %
a1600RandGraph.2	1600	12779	7.513	1259.276	784.2757	60.57 %
a1600RandGraph.3	1600	12963	7.185	1380.418	781.4181	76.66 %
a1800RandGraph.1.2	1800	14473	9.186	1077.301	1072.801	0.42 %
a1800RandGraph.1.5	1800	14222	8.568	1269.019	900.5192	40.92 %
a1800RandGraph.2	1800	14329	8.459	1415.957	872.9571	62.20 %
a1800RandGraph.3	1800	14531	8.611	1544.665	864.6654	78.64 %
a2000RandGraph.1.2	2000	16008	10.124	1165.768	1156.768	0.78 %
a2000RandGraph.1.5	2000	15835	9.955	1380.068	988.568	39.60 %
a2000RandGraph.2	2000	16062	9.897	1535.515	948.0154	61.97 %
a2000RandGraph.3	2000	15751	9.807	1709.208	971.7075	75.90 %
a3000RandGraph.1.2	3000	24045	19.692	1808.237	1794.737	0.75 %
a3000RandGraph.1.5	3000	23852	19.384	2115.289	1495.7889	41.42 %
a3000RandGraph.2	3000	24065	19.552	2359.534	1463.0341	61.28 %
a3000RandGraph.3	3000	24026	19.188	2602.898	1468.8977	77.20 %
a4000RandGraph.1.2	4000	32087	39.162	2434.258	2391.258	1.80 %
a4000RandGraph.1.5	4000	32119	30.076	2846.546	2007.046	41.83 %
a4000RandGraph.2	4000	31880	29.290	3184.475	1960.9746	62.39 %
a4000RandGraph.3	4000	32025	28.486	3485.598	1963.5977	77.51 %
a6000RandGraph.1.2	6000	47899	54.025	3594.402	3575.902	0.52 %
a6000RandGraph.1.5	6000	48077	53.019	4230.89	3021.39	40.03 %
a6000RandGraph.2	6000	48069	76.029	4701.917	2898.917	62.20 %
a6000RandGraph.3	6000	47915	73.581	5164.99	2912.49	77.34 %
a8000RandGraph.1.2	8000	64373	75.579	4789.461	4773.961	0.32 %
a8000RandGraph.1.5	8000	63812	105.115	5624.727	4012.727	40.17 %
a8000RandGraph.2	8000	63874	76.361	6267.877	3839.377	63.25 %

Continued on next page

Table 9: Results for the PCSPG-RANDOM test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
a8000RandGraph.3	8000	64177	110.962	6858.248	3869.7478	77.23 %

B.2 DIMACS test cases: RPCST

Table 10: Results for the RPCST-cologne test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
i101M1	748	6332	0.715	109271.5	109271.5	0.00 %
i101M2	748	6332	2.974	341795.5	230391.4	48.35 %
i101M3	748	6332	3.680	385703.6	192851.8	100.00 %
i102M1	749	6343	0.933	104065.8	104065.8	0.00 %
i102M2	749	6343	3.122	363303.1	293952.3	23.59 %
i102M3	749	6343	3.860	506798.4	319020.7	58.86 %
i103M1	751	6343	0.769	139749.4	139749.4	0.00 %
i103M2	751	6343	2.557	414413.9	300151.25	38.07 %
i103M3	751	6343	3.427	510785.8	255392.9	100.00 %
i104M2	741	6293	1.381	89920.84	78322.505	14.81 %
i104M3	741	6293	1.653	97148.79	48574.395	100.00 %
i105M1	741	6296	0.714	26717.2	26717.2	0.00 %
i105M2	741	6296	2.841	100269.6	83528.895	20.04 %
i105M3	741	6296	3.139	114983.2	57491.6	100.00 %
i201M2	1803	16743	11.066	355467.7	313219.67	13.49 %
i201M3	1803	16743	14.460	634950.9	490290.85	29.50 %
i201M4	1803	16743	14.282	819724.7	473862.95	72.99 %
i202M2	1804	16740	11.257	288946.8	227816.9	26.83 %
i202M3	1804	16740	12.507	430188.0	307986.75	39.68 %
i202M4	1804	16740	12.440	489456.1	244728.05	100.00 %
i203M2	1809	16762	12.272	459894.8	358362.05	28.33 %
i203M3	1809	16762	13.164	666414.9	395692.2	68.42 %
i203M4	1809	16762	12.847	707384.7	353692.35	100.00 %
i204M2	1801	16719	10.070	161700.5	161700.5	0.00 %
i204M3	1801	16719	13.795	344623.3	172311.65	100.00 %
i204M4	1801	16719	13.117	344623.3	172311.65	100.00 %
i205M2	1810	16794	13.817	571459.1	480621.3	18.90 %
i205M3	1810	16794	13.956	777936.3	417880.68	86.16 %
i205M4	1810	16794	14.023	844022.1	483966.55	74.40 %

B.3 DIMACS test cases: MWCS

Table 11: Results for the MWCS-ACTMOD test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
HCMV	3863	29293	43.111	3.384924	12.74904	73.45 %
drosophila001	5226	93394	113.456	11.93586	11.93586	0.00 %
drosophila005	5226	93394	164.101	89.45855	508.9217	82.42 %
drosophila0075	5226	93394	172.707	161.9906	694.0665	76.66 %
lymphoma	2034	7756	5.340	29.2803	134.6251	78.25 %
metabol_expr_mice_1	3523	4345	6.798	486.3809	1074.521	54.74 %
metabol_expr_mice_2	3514	4332	6.458	230.3167	470.3167	51.03 %
metabol_expr_mice_3	2853	3335	5.374	375.8796	1035.88	63.71 %

Table 12: Results for the MWCS-JMPALMK test instances.

Instance	n	m	Time (ms)	Cost	GW LB	Gap
n-1000-a-0.6-d-0.25-e-0.25	1000	4936	3.956	734.6279	2436.104	69.84 %
n-1000-a-0.6-d-0.25-e-0.5	1000	4936	1.785	1748.617	4338.253	59.69 %
n-1000-a-0.6-d-0.25-e-0.75	1000	4936	1.836	2736.778	6038.393	54.68 %
n-1000-a-0.6-d-0.5-e-0.25	1000	4936	4.459	400.2112	2056.78	80.54 %
n-1000-a-0.6-d-0.5-e-0.5	1000	4936	4.463	1111.557	3316.989	66.49 %
n-1000-a-0.6-d-0.5-e-0.75	1000	4936	4.387	1725.57	4445.405	61.18 %
n-1000-a-0.6-d-0.75-e-0.25	1000	4936	4.120	290.8025	1457.18	80.04 %
n-1000-a-0.6-d-0.75-e-0.5	1000	4936	4.224	712.9581	2420.693	70.55 %
n-1000-a-0.6-d-0.75-e-0.75	1000	4936	4.120	975.4331	2921.894	66.62 %
n-1000-a-1-d-0.25-e-0.25	1000	13279	2.155	823.6556	2240.721	63.24 %
n-1000-a-1-d-0.25-e-0.5	1000	13279	2.231	1809.199	4114.425	56.03 %
n-1000-a-1-d-0.25-e-0.75	1000	13279	2.140	2761.493	5809.901	52.47 %
n-1000-a-1-d-0.5-e-0.25	1000	13279	2.245	470.849	1548.617	69.60 %
n-1000-a-1-d-0.5-e-0.5	1000	13279	2.149	1160.829	2897.232	59.93 %
n-1000-a-1-d-0.5-e-0.75	1000	13279	2.183	1745.709	4045.207	56.85 %
n-1000-a-1-d-0.75-e-0.25	1000	13279	2.259	320.082	1151.803	72.21 %
n-1000-a-1-d-0.75-e-0.5	1000	13279	2.224	740.4528	1906.83	61.17 %
n-1000-a-1-d-0.75-e-0.75	1000	13279	2.184	997.1745	2343.634	57.45 %
n-1500-a-0.6-d-0.25-e-0.25	1500	7662	7.953	1007.326	3725.782	72.96 %
n-1500-a-0.6-d-0.25-e-0.5	1500	7662	7.727	2589.487	6562.229	60.54 %
n-1500-a-0.6-d-0.25-e-0.75	1500	7662	2.726	4133.526	9197.658	55.06 %
n-1500-a-0.6-d-0.5-e-0.25	1500	7662	7.983	657.121	2977.315	77.93 %
n-1500-a-0.6-d-0.5-e-0.5	1500	7662	7.415	1724.295	4957.598	65.22 %
n-1500-a-0.6-d-0.5-e-0.75	1500	7662	6.991	2633.488	6759.007	61.04 %
n-1500-a-0.6-d-0.75-e-0.25	1500	7662	7.051	434.9262	2341.859	81.43 %

Continued on next page

Table 12: Results for the MWCS-JMPALMK test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
n-1500-a-0.6-d-0.75-e-0.5	1500	7662	2.699	1045.44	3747.08	72.10 %
n-1500-a-0.6-d-0.75-e-0.75	1500	7662	2.898	1407.977	4341.327	67.57 %
n-1500-a-1-d-0.25-e-0.25	1500	20527	3.524	1205.373	3264.203	63.07 %
n-1500-a-1-d-0.25-e-0.5	1500	20527	3.669	2707.081	6080.164	55.48 %
n-1500-a-1-d-0.25-e-0.75	1500	20527	4.060	4175.813	8783.18	52.46 %
n-1500-a-1-d-0.5-e-0.25	1500	20527	3.649	740.277	2461.711	69.93 %
n-1500-a-1-d-0.5-e-0.5	1500	20527	3.973	1782.566	4422.099	59.69 %
n-1500-a-1-d-0.5-e-0.75	1500	20527	3.923	2674.675	6091.821	56.09 %
n-1500-a-1-d-0.75-e-0.25	1500	20527	3.863	477.7948	1648.632	71.02 %
n-1500-a-1-d-0.75-e-0.5	1500	20527	3.924	1071.441	2861.031	62.55 %
n-1500-a-1-d-0.75-e-0.75	1500	20527	3.872	1410.238	3559.718	60.38 %
n-500-a-0.62-d-0.25-e-0.25	500	2597	1.930	356.401	1199.654	70.29 %
n-500-a-0.62-d-0.25-e-0.5	500	2597	0.802	898.4498	2205.742	59.27 %
n-500-a-0.62-d-0.25-e-0.75	500	2597	0.767	1402.885	3024.709	53.62 %
n-500-a-0.62-d-0.5-e-0.25	500	2597	2.375	226.201	987.4253	77.09 %
n-500-a-0.62-d-0.5-e-0.5	500	2597	2.315	624.9131	1623.962	61.52 %
n-500-a-0.62-d-0.5-e-0.75	500	2597	0.836	947.6905	2162.381	56.17 %
n-500-a-0.62-d-0.75-e-0.25	500	2597	0.939	134.4324	828.3533	83.77 %
n-500-a-0.62-d-0.75-e-0.5	500	2597	0.803	345.6002	1216.692	71.60 %
n-500-a-0.62-d-0.75-e-0.75	500	2597	0.914	482.1382	1445.92	66.66 %
n-500-a-1-d-0.25-e-0.25	500	6519	0.997	409.6342	1063.28	61.47 %
n-500-a-1-d-0.25-e-0.5	500	6519	1.054	949.8127	2122.384	55.25 %
n-500-a-1-d-0.25-e-0.75	500	6519	0.993	1427.803	2929.67	51.26 %
n-500-a-1-d-0.5-e-0.25	500	6519	1.171	270.9102	819.5848	66.95 %
n-500-a-1-d-0.5-e-0.5	500	6519	0.976	653.8947	1441.323	54.63 %
n-500-a-1-d-0.5-e-0.75	500	6519	1.105	958.3406	2004.07	52.18 %
n-500-a-1-d-0.75-e-0.25	500	6519	0.998	150.0016	528.9513	71.64 %
n-500-a-1-d-0.75-e-0.5	500	6519	0.978	349.0255	924.8322	62.26 %
n-500-a-1-d-0.75-e-0.75	500	6519	0.989	482.0102	1167.051	58.70 %
n-750-a-0.647-d-0.25-e-0.25	750	4219	3.546	586.4515	1798.94	67.40 %
n-750-a-0.647-d-0.25-e-0.5	750	4219	3.455	1324.763	3280.711	59.62 %
n-750-a-0.647-d-0.25-e-0.75	750	4219	1.331	2059.922	4567.116	54.90 %
n-750-a-0.647-d-0.5-e-0.25	750	4219	1.382	301.4397	1324.321	77.24 %
n-750-a-0.647-d-0.5-e-0.5	750	4219	1.255	888.0125	2404.871	63.07 %
n-750-a-0.647-d-0.5-e-0.75	750	4219	1.275	1357.369	3276.089	58.57 %
n-750-a-0.647-d-0.75-e-0.25	750	4219	3.624	228.8896	1070.453	78.62 %
n-750-a-0.647-d-0.75-e-0.5	750	4219	3.471	554.0301	1784.386	68.95 %
n-750-a-0.647-d-0.75-e-0.75	750	4219	3.541	751.8793	2088.89	64.01 %
n-750-a-1-d-0.25-e-0.25	750	9822	1.581	631.9982	1699.787	62.82 %
n-750-a-1-d-0.25-e-0.5	750	9822	1.542	1380.98	3077.466	55.13 %
n-750-a-1-d-0.25-e-0.75	750	9822	1.557	2092.577	4416.8	52.62 %

Continued on next page

Table 12: Results for the MWCS-JMPALMK test instances (continued).

Instance	n	m	Time (ms)	Cost	GW LB	Gap
n-750-a-1-d-0.5-e-0.25	750	9822	1.517	351.1403	1144.497	69.32 %
n-750-a-1-d-0.5-e-0.5	750	9822	1.588	918.4944	2205.828	58.36 %
n-750-a-1-d-0.5-e-0.75	750	9822	1.632	1370.38	3076.459	55.46 %
n-750-a-1-d-0.75-e-0.25	750	9822	1.593	252.662	843.233	70.04 %
n-750-a-1-d-0.75-e-0.5	750	9822	1.573	564.8708	1455.649	61.19 %
n-750-a-1-d-0.75-e-0.75	750	9822	1.652	754.9382	1775.413	57.48 %