An Exact Combinatorial Algorithm for Minimum Graph Bisection[⋆]

Daniel Delling[1], Daniel Fleischman[2], Andrew V. Goldberg[1], Ilya Razenshteyn[3], and Renato F. Werneck[1⋆⋆]

[1] Microsoft Research Silicon Valley
{dadellin,goldberg,renatow}@microsoft.com
[2] School of Operations Research and Information Engineering, Cornell University
df288@cornell.edu
[3] Computer Science and Artificial Intelligence Laboratory, MIT
ilyaraz@mit.edu

September 11, 2013

Abstract. We present a novel exact algorithm for the minimum graph bisection problem, whose goal is to partition a graph into two equally-sized cells while minimizing the number of edges between them. Our algorithm is based on the branch-and-bound framework and, unlike most previous approaches, it is fully combinatorial. We introduce novel lower bounds based on packing trees, as well as a new decomposition technique that contracts entire regions of the graph while preserving optimality guarantees. Our algorithm works particularly well on graphs with relatively small minimum bisections, solving to optimality several large real-world instances (with up to millions of vertices) for the first time.

1 Introduction

We consider the *minimum graph bisection* problem. It takes as input an undirected graph $G = (V, E)$, and its goal is to partition V into two sets A and B of roughly equal weight so as to minimize the total cost of all edges between A and B. This fundamental combinatorial optimization problem is a special case of *graph partitioning* [25], which may ask for more than two cells. It has numerous applications, including image processing [62, 66], computer vision [46], divide-and-conquer algorithms [50], VLSI circuit layout [8], distributed computing [51], compiler optimization [39], load balancing [33], and route planning [7, 15, 19, 35, 36, 40, 49]. In theory, the bisection problem is NP-hard [26] for general graphs, with a best known approximation ratio of $O(\log n)$ [55]. Only some restricted graph classes, such as grids without holes [22] and graphs with bounded treewidth [37], have known polynomial-time solutions.

In practice, there are numerous general-purpose heuristics for graph partitioning, including CHACO [34], METIS [44], SCOTCH [13, 54], Jostle [65], and KaHiP [58, 59], among others [6, 12, 31, 63, 48]. Successful heuristics tailored to particular graph classes, such as DibaP [53] (for meshes) and PUNCH [16] (for road networks), are also available. These algorithms can be quite fast (often running in near-linear time) and handle very large graphs, with tens of millions of vertices. They cannot, however, prove optimality or provide approximation guarantees. Moreover, with a few notable exceptions [59, 63], most of these algorithms only perform well if a certain degree of imbalance is allowed.

There is also a vast literature on practical exact algorithms for graph bisection (and partitioning), mostly using the branch-and-bound framework [47]. Most of these algorithms use sophisticated machinery to obtain lower bounds, such as multicommodity flows [60, 61] or linear [3, 9, 24], semidefinite [1, 3, 43], and quadratic programming [30]. Computing such bounds, however, can be quite expensive in terms of time and space.

[⋆] This article combines and unifies results originally described in two conference publications [17, 18] and incorporate several extensions that lead to a more robust algorithm. All work was done while all authors were at Microsoft Research Silicon Valley.

[⋆⋆] Corresponding author.

As a result, even though the branch-and-bound trees can be quite small for some graph classes, published algorithms can only solve instances of moderate size (with hundreds or a few thousand vertices) to optimality, even after a few hours of processing. (See Armbruster [1] for a survey.) Combinatorial algorithms can offer a different tradeoff: weaker lower bounds that are much faster to compute. An algorithm by Felner [23], for example, works reasonably well on random graphs with up to 100 vertices, but does not scale to larger instances.

This article introduces a new purely combinatorial exact algorithm for graph bisection that is practical on a wide range of graph classes. It is based on two theoretical insights: a *packing bound* and a *decomposition* strategy. The packing bound is a novel combinatorial lower bound in which we take a collection of edge-disjoint trees with certain properties and argue that any balanced bisection must cut a significant fraction of them. The decomposition strategy allows us to contract entire regions of the graph without losing optimality guarantees; we show that one can find the optimal solution to the input problem by independently solving a small number of (usually much easier) subproblems.

Translating our theoretical findings into a practical algorithm is not trivial. Both the packing bound and the decomposition technique make use of certain nontrivial combinatorial objects, such as collections of disjoint trees or edges. Although building feasible versions of these objects is straightforward, the quality of the bounds depends strongly on the properties of these structures, as does the size of the resulting branch-and-bound tree. This motivates another important contribution of this article: efficient algorithms to generate structures that are good enough to make our bounds effective in practice. While these algorithms are heuristics (in the sense that they may not necessarily lead to the best possible lower bound), the lower bounds they provide are provably valid. This is all we need to ensure that the entire branch-and-bound routine is correct: it is guaranteed to find the optimal solution when it finishes.

Finally, we present experimental evidence that combining our theoretical contributions with the appropriate implementation does pay off. Our algorithm works particularly well on instances with relatively small minimum bisections, solving large real-world graphs (with tens of thousands to more than a million vertices) to optimality. See Figure 1 for an example. In fact, our algorithm outperforms previous techniques on a wide range of inputs, often by orders of magnitude. We can solve several benchmark instances that have been open for decades, sometimes in a few minutes or even seconds. That said, our experiments also show that there are classes of inputs (such as high-expansion graphs with large cuts) in which our algorithm is asymptotically slower than existing approaches.

The remainder of this article is organized as follows. After establishing in Section 2 the notation we use, we explain our new packing bound in detail in Section 3. Section 4 then proposes sophisticated algorithms to build the combinatorial objects required by the packing bound computation. We then show, in Section 5, how we can fix some vertices to one of the cells without actually branching on them; this may significantly reduce the size of the branch-and-bound tree and is crucial in practice. Section 6 introduces our decomposition technique and proposes practical implementations of the theoretical concept. Section 7 explains how all ingredients are put together in our final branch-and-bound routine and discusses missing details, such as branching rules and upper bound computation. Section 8 then shows how our techniques can be extended to handle large edge costs efficiently. Finally, Section 9 presents extensive experiments showing that our approach outperforms previous algorithms on many (but by no means all) graph classes, including some corresponding to real-world applications.

2 Preliminaries

We take as input a graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges. Each vertex $v \in V$ has an integral *weight* $w(v)$, and each edge $e \in E$ has an associated integral *cost* $c(e)$. By extension, for any set $S \subseteq V$, let $w(S) = \sum_{v \in S} w(v)$ and let $W = w(V)$ denote the total weight of all vertices. A *partition* of G is a partition of V, i.e., a set of subsets of V which are disjoint and whose union is V. We say that each such subset is a *cell*, whose weight is defined as the sum of the weights of its vertices. The *cost* of a partition is the sum of the costs of all edges whose endpoints belong to different cells. A *bisection* is a partition into two cells. For a given input parameter $\epsilon \geq 0$, we are interested in computing a *minimum ϵ-balanced bisection* of G, i.e., a partition of V into exactly two sets (*cells*) such that (1) the weight of each cell is at most $W_+ = \lfloor(1+\epsilon)\lceil W/2\rceil\rfloor$ and (2) the total cost of all edges between cells (*cut size*) is minimized. Conversely, $W_- = W - W_+$ is the *minimum allowed cell size*. If $\epsilon = 0$, we say the partition is *perfectly balanced* (or just *balanced*).

To simplify exposition, we will describe our algorithms assuming that all edges have unit costs. Small integral edge costs can be dealt with by creating parallel unit edges; Section 8 shows how we can use a scaling technique to handle arbitrary edge costs.

A standard technique for finding exact solutions to NP-hard problems is *branch-and-bound* [27, 47]. It performs an implicit enumeration of all possible solutions by dividing the original problem into two or more slightly simpler subproblems, solving them recursively, and picking the best solution found. Each node of the branch-and-bound tree corresponds to a distinct subproblem. In a minimization context, the algorithm keeps a global *upper bound* U on the solution of the original problem; this bound is updated whenever better solutions are found. To process a node in the tree, we first compute a *lower bound* L on any solution to the corresponding subproblem. If $L \geq U$, we *prune* the node: it cannot lead to a better solution. Otherwise, we *branch*, creating two or more simpler subproblems.

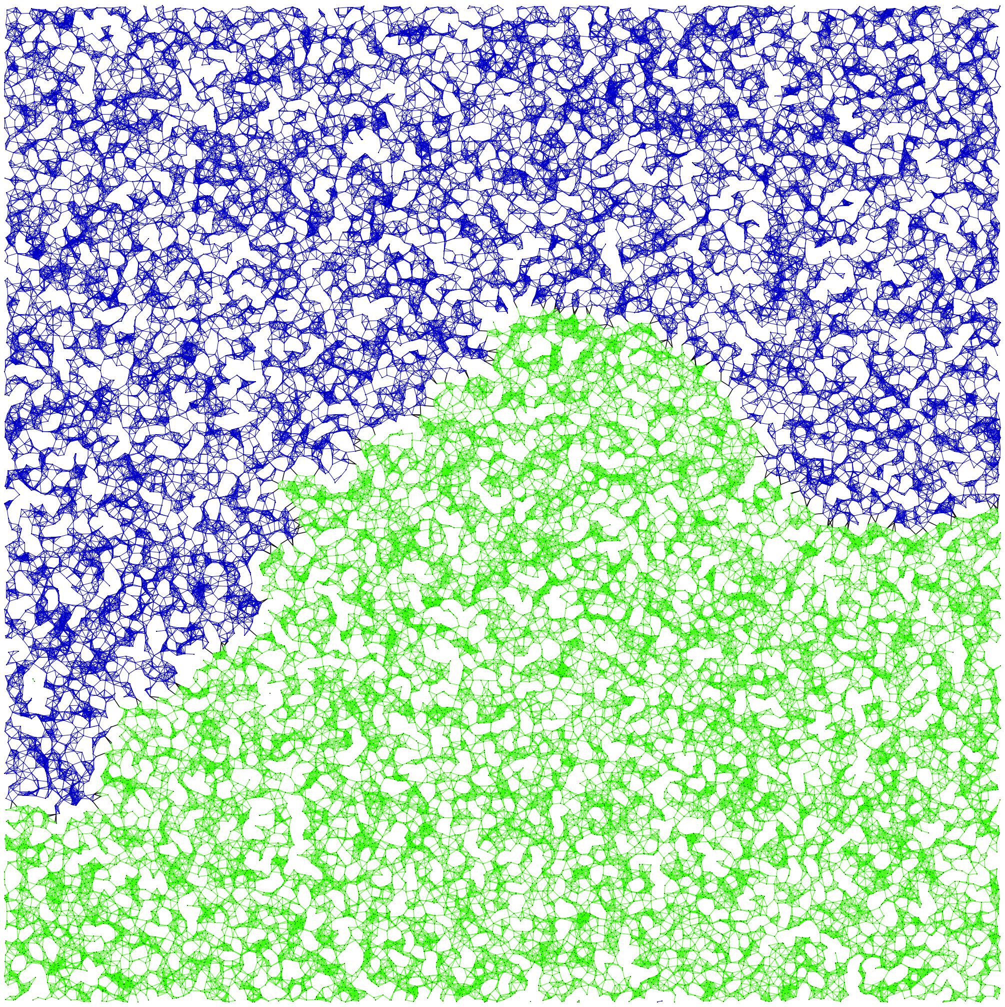


Fig. 1. Minimum bisection of rgg15, a random geometric graph with 32768 vertices and 160240 edges. Each cell (with a different color) has exactly 16384 vertices, and there are 181 cut edges.

In the concrete case of graph bisection, each node of the branch-and-bound tree corresponds to a *partial assignment* (A, B), where $A, B \subseteq V$ and $A \cap B = \emptyset$. We say the vertices in A or B are *assigned*, and all others are *free* (or *unassigned*). This node implicitly represents all valid bisections (A^+, B^+) that are *extensions* of (A, B), i.e., such that $A \subseteq A^+$ and $B \subseteq B^+$. In particular, the *root* node, which represents all valid bisections, has the form $(A, B) = (\{v\}, \emptyset)$. Note that we can fix an arbitrary vertex v to one cell to break symmetry.

To process an arbitrary node (A, B), we must compute a lower bound $L(A, B)$ on the value of any extension (A^+, B^+) of (A, B). The fastest exact algorithms [1, 3, 9, 24, 30, 43] usually apply mathematical programming techniques to find lower bounds. In this article, we use only combinatorial bounds that can be computed efficiently. In particular, our basic algorithm starts from the well-known [11, 16] *flow bound*: the minimum *s*–*t* cut between A and B, taking the edge costs as capacities. This is a valid lower bound because any extension (A^+, B^+) must separate A from B; moreover, it can be computed rather quickly in practice [29]. If the minimum cut happens to be balanced, we can prune (and update U, if applicable). Otherwise, we choose a free vertex v and branch on it, generating subproblems $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$.

Note that the flow lower bound can only work well when A and B are large enough. In particular, if either set is empty, the flow bound is zero. Even when A and B have approximately the same size, the corresponding minimum cut is often far from balanced, with one side containing many more vertices than the other. This makes the flow bound rather weak by itself. To overcome these issues, we introduce a new *packing lower bound*, which we describe next.

3 Packing Bound

The *packing bound* is a novel lower-bounding technique that takes into account the fact that the optimal solution must be balanced. Consider a partial assignment (A, B). Let f be the value of the maximum A–B flow, and G_f be the graph obtained by removing all flow edges from G (recall that we assume all edges have unit costs/capacities). Without loss of generality, assume that A is the *main side*, i.e., that the set of vertices reachable from A in G_f has higher total weight than those reachable from B. We will compute our new bound on G_f, since this allows us to simply add it to f to obtain a unified lower bound. (A detailed example illustrating the concepts introduced in this section can be found in Appendix A.1.)

To compute the bound, we need a *tree packing* $\mathcal{T}$. This is a collection of trees (acyclic connected subgraphs of G_f) such that: (1) the trees are edge-disjoint; (2) each tree contains exactly one edge incident to A; and (3) the trees are maximal (no edge can be added to $\mathcal{T}$ without violating the previous properties). See Figure 2(b) (in the appendix) for an example. Given a set $S \subseteq V$, let $\mathcal{T}(S)$ be the subset of $\mathcal{T}$ consisting of all trees that contain a vertex in S. By extension, let $\mathcal{T}(v) = \mathcal{T}(\{v\})$.

For now, assume a tree packing $\mathcal{T}$ is given; Section 4 will show how one can be built. With $\mathcal{T}$, we can reason about any extension (A^+, B^+) of (A, B). By definition, a tree $T_i \in \mathcal{T}$ contains a path from each of its vertices to A; if a vertex in B^+ is in T_i, at least one edge from T_i must be cut in (A^+, B^+). Since each tree $T_i \in \mathcal{T}(B^+)$ contains an edge-disjoint path from A to B^+ in G_f, the following holds:

Lemma 1. *If B^+ is an extension of B, then $f + |\mathcal{T}(B^+)|$ is a lower bound on the cost of the corresponding bisection $(V \setminus B^+, B^+)$.*

This applies to a fixed extension B^+ of B; we need a lower bound that applies to *all* (exponentially many) possible extensions. We must therefore reason about a *worst-case extension* B^*, i.e., a set $B^* \supseteq B$ that minimizes the bound given by Lemma 1.

First, note that $w(B^*) \geq W_-$, since $(V \setminus B^*, B^*)$ must be a valid bisection.

Second, let $D_f \subseteq V$ be the set of all vertices that are *unreachable* from A in G_f (in particular, $B \subseteq D_f$). Without loss of generality, we can assume that B^* contains D_f. After all, in Lemma 1 any vertex $v \in D_f$ is *deadweight*: since there is no path from v to A, it does not contribute to the lower bound.

To reason about other vertices in B^*, we first establish a relationship between $\mathcal{T}$ and vertex weights by predefining a *vertex allocation*, a mapping from vertices to trees. We allocate each reachable free vertex v (i.e., $v \in V \setminus (D_f \cup A)$) to one of the trees in $\mathcal{T}(v)$, as shown in Figure 2(c), in the appendix. (Section 4 will discuss how to compute such an allocation.) The *weight* $w(T_i)$ of a tree $T_i \in \mathcal{T}$ is the sum of the weights of all vertices allocated to T_i.

Given a fixed allocation, we can assume without loss of generality that, if B^* contains a single vertex allocated to a tree T_i, it will contain *all* vertices allocated to T_i. To see why, note that, according to Lemma 1, the first vertex increases the lower bound by one unit, but the other vertices in the tree are free.

Moreover, to ensure that $w(B^*) \geq W_-$, B^* must contain a *feasible* set of trees $\mathcal{T}' \subseteq \mathcal{T}$, i.e., a set whose total weight $w(\mathcal{T}')$ (defined as $\sum_{T_i \in \mathcal{T}'} w(T_i)$) is at least as high as the *target weight* $W_f = W_- - w(D_f)$. Since B^* is the worst-case extension, it must correspond to a feasible set $\mathcal{T}'$ of minimum cardinality. Formally, given a partial assignment (A, B), a flow f, a tree packing $\mathcal{T}$, and an associated vertex allocation, we define the *packing bound* as $p(\mathcal{T}) = \min_{\mathcal{T}' \subseteq \mathcal{T}, w(\mathcal{T}') \geq W_f} |\mathcal{T}'|$.

Note that the exact value of this bound can be computed by a *greedy algorithm*: it suffices to pick trees in decreasing order of weight until their accumulated weight is at least W_f. The bound is the number of trees picked.

We can strengthen this bound further by allowing *fractional allocations*. Instead of allocating v's weight to a single tree, we can distribute $w(v)$ arbitrarily among all trees in $\mathcal{T}(v)$. For v's allocation to be *valid*, each tree must receive a nonnegative fraction of v's weight, and these fractions must add up to one. (Figure 2(d), in the appendix, has an example; Section 4 will discuss how such an allocation can be found.) The weight of a tree T is defined in the natural way, as the sum of all fractional weights allocated to T. By making trees more balanced, fractional allocations can improve the packing bound and are particularly useful when the average number of vertices per tree is small, or when some vertices have high degree. The fact that the packing bound is valid is our first important theoretical result.

Theorem 1. *Consider a partial assignment (A, B), a flow f, a tree packing $\mathcal{T}$, and a valid fractional allocation of weights. Then $f + p(\mathcal{T})$ is a lower bound on the cost of any valid extension of (A, B).*

Proof. Let (A^*, B^*) be a minimum-cost extension of (A, B). Let $\mathcal{T}^* = \mathcal{T}(B^*)$ be the set of trees in $\mathcal{T}$ that contain vertices in B^*. The cut size of (A^*, B^*) must be at least $f + |\mathcal{T}^*|$, since at least one edge in each tree must be cut. To prove that $f + p(\mathcal{T})$ is a valid lower bound, we show that $p(\mathcal{T}) \leq |\mathcal{T}^*|$ (the flow bound f is still valid because only edges in G_f are used in the packing bound). It suffices to show that $w(\mathcal{T}^*) \geq W_f$, i.e., that $\mathcal{T}^*$ is feasible. (This ensures that $\mathcal{T}^*$ will be one of the trees considered by the packing bound; since it minimizes over all feasible sets, the bound can be no higher than $|\mathcal{T}^*|$.) Let $R^* = B^* \setminus D_f$ be the set of vertices in B^* that are reachable from A in G_f; clearly, $w(R^*) = w(B^* \setminus D_f) \geq w(B^*) - w(D_f)$. Moreover, $w(\mathcal{T}^*) \geq w(R^*)$ must hold because (1) every vertex $v \in R^*$ must hit some tree in $\mathcal{T}^*$ (the trees are maximal); (2) although $w(v)$ may be arbitrarily split among several trees in $\mathcal{T}(v)$, all these must be in $\mathcal{T}^*$ (by definition); and (3) vertices of $\mathcal{T}^*$ that are in A^* (and therefore not in R^*) can only contribute nonnegative weights to the trees. Finally, since B^* is a valid bisection, we must have $w(B^*) \geq W_-$. Putting everything together, we have $w(\mathcal{T}^*) \geq w(R^*) \geq w(B^*) - w(D_f) \geq W_- - w(D_f) = W_f$. This completes the proof. $\square$

4 Bound Computation

Theorem 1 applies to any valid tree packing $\mathcal{T}$, but the quality of the bound it provides varies; intuitively, more balanced packings lead to better bounds. More precisely, we should pick $\mathcal{T}$ (and an associated weight allocation) so as to avoid heavy trees, which improves $p(\mathcal{T})$ by increasing the number of trees required to achieve the target weight W_f. For a fixed graph G_f, the number $|\mathcal{T}|$ of trees is the same for any tree packing $\mathcal{T}$, as is their total weight $w(\mathcal{T})$; therefore, in the ideal tree packing all trees would have the same weight.

Unfortunately, finding the best such tree packing is NP-hard. To prove this, we define a decision version of this problem as follows. Given a graph G with integral vertex weights and a partial assignment (A, B), decide whether there is a valid tree packing $\mathcal{T}$ rooted at A (with weight allocations) such that all trees have the same weight. For simplicity, consider the decision version of the 2-TREE PACKING special case, with exactly two trees ($|\mathcal{T}| = 2$); we must decide whether there is a packing in which both trees have exactly the same weight. Even this problem is NP-hard.

Theorem 2. *2-TREE PACKING is NP-hard.*

Proof. We prove it with a reduction from PARTITION, defined as follows [25]: given a set $S = \{s_1, s_2, \ldots, s_k\}$ of k integers, decide whether it can be partitioned into two subsets that add up to the same value. For the reduction, we define a graph H (with unweighted edges) as follows. For each $s_i \in S$, add vertices v_i and w_i to H, as well as a single edge (v_i, w_i). Also, for $1 \leq i < k$, we add two parallel edges between w_i and w_{i+1}. In addition, create a vertex u connected by two parallel edges to w_1. Assign vertex weight 0 to all w_i and u, and weight s_i to v_i. Finally, set $A = \{u\}$ and $B = \emptyset$. It is easy to see that 2-TREE PACKING is true if and only if PARTITION is true. □

Given this result, in practice we resort to (fast) heuristics to find a valid tree packing. Ideally, the trees and weight allocations should be computed simultaneously, to account for the interplay between them. Since it is unclear how to do so efficiently, we use a two-stage approach instead: we first compute a valid tree packing, then allocate vertex weights to these trees appropriately. We discuss each stage in turn.

4.1 Generating Trees

The goal of the first stage is to generate maximal edge-disjoint trees rooted at A that are as balanced and intertwined as possible, since this typically enables a more even distribution of vertex weights. We try to achieve this by growing these trees simultaneously, balancing their sizes (number of edges).

More precisely, each tree starts with a single edge (the one adjacent to A) and is marked *active*. In each step, we pick an active tree with minimum size (number of edges) and try to expand it by one edge in DFS fashion. A tree that cannot be expanded is marked as inactive. We stop when there are no active trees left. We call this algorithm *SDFS* (for *simultaneous depth-first search*).

An efficient implementation of SDFS requires a careful choice of data structures. In particular, a standard DFS implementation associates information (such as parent pointers and status within the search) with vertices, which are the entities added to and removed from the DFS stack. In our setting, however, the same vertex may be in several trees (and stacks) simultaneously, making an efficient implementation more challenging. We get around this by associating information with *edges* instead. Since each edge belongs to at most one tree, it has at most one parent and is contained in at most one stack. The combined size of all data structures we need is therefore $O(m)$.

Given this representation, we now describe our SDFS algorithm in more detail. We associated with each tree T_i a stack S_i, initially containing the root edge of T_i. (Each non-flow edge incident to A becomes the root edge of some T_i.) The basic step of the SDFS algorithm is as follows. First, pick an active tree T_i of minimum size. (This can be done efficiently by maintaining the current active trees in *buckets* according to their current number of edges.) Let (u, v) be the edge on top of S_i (the stack associated with T_i), and assume v is farther from T_i's root than u is (i.e., u is v's parent). Scan vertex v, looking for an *expansion edge*. This is an edge (v, w) such that (1) (v, w) is free (not assigned to any tree yet) and (2) no edge incident to w belongs to T_i. The first condition ensures that the final trees are disjoint, while the second makes sure they have no cycles. If no such expansion edge exists, we backtrack by popping (u, v) from S_i; if S_i becomes empty, T_i can no longer grow, so we mark it as inactive. If expansion edges do exist, we pick one such edge (v, w), push it onto S_i, and add it to T_i by setting *parent*$(v, w) \leftarrow (u, v)$. The algorithm repeats the basic step until there are no more active trees.

We must still define which expansion edge (v, w) to select when processing (u, v). We prefer an edge (v, w) such that w has several free incident edges (to help keep the tree growing) and is as far as possible from A (to minimize congestion around the roots, which is also why we do DFS). We use a preprocessing step to compute the distances from A to all vertices with a single breadth-first search (BFS).

To bound the running time of SDFS, note that a vertex v of degree $\deg(v)$ can be scanned $O(\deg(v))$ times, since each scan either eliminates a free edge or backtracks. When scanning v, we can process each outgoing edge (v, w) in $O(1)$ time using a hash table to determine whether w is already incident to v's tree. The worst-case time is therefore $\sum_{v \in V} (\deg(v))^2 = O(m\Delta)$, where Δ is the maximum degree.

4.2 Weight Allocation

Once a tree packing $\mathcal{T}$ is built, we must allocate the weight of each vertex v to the trees $\mathcal{T}(v)$ it is incident to. Our final goal is to have the weights as evenly distributed among the trees as possible. We work in two stages. First, an *initial allocation* splits the weight of each vertex evenly among all trees it is incident to. We then run a *local search* to rebalance the weight allocation among the trees. We process one vertex at a time (in arbitrary order) by reallocating v's weight among the trees in $\mathcal{T}(v)$ in a locally optimal way. More precisely, v is processed in two steps. First, we reset v's existing allocation by removing v's share from all trees it is currently allocated to, thus reducing their weights. We then distribute v's weight among the trees in $\mathcal{T}(v)$ (from lightest to heaviest), evening out their weights as much as possible. In other words, we add weight to the lightest tree until it is as heavy as the second lightest, then add weight to the first two trees (at the same rate) until each is as heavy as the third, and so on. We stop as soon as v's weight is fully allocated. The entire local search runs in $O(m \log m)$ time, since it must sort (by weight) the adjacency lists of each vertex in the graph once. In practice, we run the local search three times to further refine the weight distribution; additional runs typically have little effect.

5 Forced Assignments

We now propose a technique that uses logical implications to further reduce the size of the branch-and-bound tree. Consider a partial assignment (A, B). The quality of the bounds we use depends crucially on the sum of the degrees of all vertices already fixed to A or B, since they bound both the flow value and the number of trees created. If we could fix more vertices to A or B without explicitly branching on them, we would boost the effectiveness of both bounds, reducing the size of the branch-and-bound tree. This section explains how we can do this by exploiting some properties of the tree packing itself. These *forced assignments* work particularly well when the current lower bound for (A, B) is close enough to the upper bound U. Since most nodes of the branch-and-bound tree have this property, this technique can reduce the total running time significantly.

Our goal is to infer, without branching, that a certain free vertex v must be assigned to A (or B). Intuitively, if we show that assigning v to one side would increase the lower bound to at least match the upper bound, we can safely assign v to the other side. Sections 5.1–5.3 propose specific versions of such forced assignments. They all require computing the packing bound for a slightly different set of trees; Section 5.4 shows how this can be done quickly, without a full recomputation. Without loss of generality, we once again assume that the weight of the vertices reachable from A in G_f is at least as high as the weight of those reachable from B (i.e., that A is the main side). In what follows, let $\mathcal{T}$ be a tree packing with weight allocations and $f + p(\mathcal{T})$ be the current lower bound. All techniques we present are illustrated in Appendix A.2.

5.1 Flow-based Assignments

We first consider *flow-based forced assignments.* Let v be a free vertex reachable from A in G_f, and consider what would happen if it were assigned to B. The flow bound would immediately increase by $|\mathcal{T}(v)|$ units, since each tree in $\mathcal{T}(v)$ contains an independent path from v to A. We cannot, however, simply increase the overall lower bound to $f + p(\mathcal{T}) + |\mathcal{T}(v)|$, since the packing bound may already be "using" some trees in $\mathcal{T}(v)$. Instead, we must compute a new packing bound $p(\mathcal{T}')$, where $\mathcal{T}' = \mathcal{T} \setminus \mathcal{T}(v)$ but the weights originally assigned to the trees $\mathcal{T}(v)$ are treated as deadweight (unreachable). If the updated bound $f + p(\mathcal{T}') + |\mathcal{T}(v)|$ is U or higher, we have proven that no solution that extends $(A, B \cup \{v\})$ can improve the best known solution. Therefore, we can safely assign v to A.

We can make a symmetric argument for vertices w that are reachable from B in G_f, as long as we also compute an edge packing $\mathcal{T}_B$ on B's side. If we assigned such a vertex w to A, the overall flow would increase by $|\mathcal{T}_B(w)|$. Since the extra flow is on B's side, the original packing bound $p(\mathcal{T})$ (which uses only edges reachable from A) is still valid. We can obtain a slightly better bound $p'(\mathcal{T})$, however, by using the fact that vertex w itself can no longer be considered deadweight (i.e., assumed to be on B's side), since we are explicitly assigning it (tentatively) to A. If the new bound $f + |\mathcal{T}_B(w)| + p'(\mathcal{T})$ is U or higher, we can safely assign w to B.

5.2 Extended Flow-based Assignments

As described, flow-based forced assignments are weaker than they could be. When we take a vertex reachable from A in G_f and tentatively assign it to B, we argue that each tree containing v results in a new unit of flow, since following parent pointers within the tree leads to a vertex in A. This means that, even though a tree may have several edges incident to v, we only "send" new flow through the parent edge. This section shows how the bound can be strengthened by considering child edges as well. This is not trivial, however. Each child edge is the root of a different subtree, but no such subtree contains a vertex in A (by construction), so it cannot improve the flow bound by itself. To obtain new paths to A, we must use other trees as well.

For a precise description of the algorithm, we need some additional notation. For every edge e, let *tree*(e) be the tree (from $\mathcal{T}$) to which e is assigned. Each edge $e = (v, u)$ is either a *parent* or a *child* edge of v, depending on whether u is on the path from v to the root of *tree*(e) or not. Define *path*(e) as the path (including e itself) from e to the root of *tree*(e) following only parent edges. Moreover, let *sub*(e) be the subtree of *tree*(e) rooted at e. (Note that *tree*(e), *path*(e), and *sub*(e) are undefined if e belongs to no tree in $\mathcal{T}$.) Let $\sigma(e)$ be the set of all trees $t \in \mathcal{T} \setminus \{\mathit{tree}(e)\}$ that intersect *sub*(e), i.e., that have at least one vertex in common with *sub*(e).

Given a vertex v reachable from A and an incident edge e, we can define an associated *expansion tree* $x_v(e)$, which is equal to *tree*(e) if e is a parent edge, and equal to any tree in $\sigma(e)$ if e is a child edge (if e belongs to no tree or if $\sigma(e) = \emptyset$, $x_v(e)$ is undefined). The *expansion set* for v, denoted by $X(v)$, is the union of $x_v(e)$ over all edges e incident to v. Note that $X(v) \subseteq \mathcal{T}$ is a set of trees with the following useful property.

Lemma 2. *Assigning v to B would increase the flow bound by at least $|X(v)|$ units.*

Proof. We must show that, for each tree $T \in X(v)$, we can create an independent path from v to A in G_f. If T is the expansion tree for a parent edge e of v, we simply take *path*(e). Otherwise, if T is the expansion tree for a child edge e of v, we take the concatenation of two paths, one within *sub*(e) and another within T itself. By construction (of $X(v)$), these paths are all disjoint. □

Note that we used a similar argument in Section 5.1 to justify the standard flow-based assignment; the difference here is that we (implicitly) send flow through potentially more trees. Accordingly, as long as we consider all affected trees as deadweight, a valid (updated) packing bound for $(A, B \cup \{v\})$ is given by $f + |X(v)| + p(\mathcal{T} \setminus X(v))$. If this is at least U, we can safely assign v to A.

A similar argument can be made if v is initially reachable from B in G_f. A valid lower bound for $(A \cup \{v\}, B)$ is $f + |X(v)| + p'(\mathcal{T})$, where $p'(\mathcal{T})$ is the standard packing bound, but with v no longer considered as deadweight.

Practical Issues Although extended flow-based forced assignments are conceptually simple, they require access to several pieces of information associated with each edge. We now explain how they can be computed efficiently.

We can compute $\sigma(\cdot)$ (as defined above) for all edges in a tree T by traversing the tree in bottom-up fashion, as a "preprocessing" step before actually trying the extended forced assignments. Let $e = (v, w) \in T$ be the parent edge of some vertex v. We can compute $\sigma(e)$ as the union of the $\sigma(f)$ values for all child edges f of e in T, together with all other trees incident to v itself. More precisely, $\sigma(e) = (\cup_{f=(u,v)\in(T\setminus\{e\})}\sigma(f)) \cup (\mathcal{T}(v) \setminus \{T\})$. Note, however, that each set $\sigma(\cdot)$ can have $\Theta(|\mathcal{T}|) = O(m)$ trees, so maintaining all such sets in memory during the algorithm can be expensive. In practice, for each edge e we keep only a subset $\tilde{\sigma}(e) \subseteq \sigma(e)$ (picked uniformly at random) of size at most κ, an input parameter; $\kappa = 3$ works well in practice. To process an edge e during the forced assignment routine, we pick a random unused element from $\tilde{\sigma}(e)$ as the expansion tree $x_v(e)$ of e. (We could maximize $|X(v)|$ by computing $x_v(e)$ for all edges e incident to v simultaneously using a maximum matching algorithm, but this would be expensive.) Although using $\tilde{\sigma}(e)$ instead of $\sigma(e)$ may make the algorithm slightly less effective (we may run out of unused elements in $\tilde{\sigma}(e)$ even though some would be available in $\sigma(e)$), it does not affect correctness.

We must be careful to compute the $\tilde{\sigma}(e)$ sets in a space-efficient way: we still need $\sigma(e)$ in order to compute $\tilde{\sigma}(e)$, which is a random subset. But $\sigma(e)$ is the union of up to $O(m)$ distinct $\sigma(e')$ values, one for each child edge $e' = (v, u)$ of v. Instead of first building all these child subsets and then computing their union, we build $\sigma(e)$ incrementally; as soon as each $\sigma(e')$ is computed, we determine $\tilde{\sigma}(e')$, then merge $\sigma(e')$ into $\sigma(e)$ (initially empty) and discard $\sigma(e')$. Traversing the tree in DFS post-order ensures that, at any time, all edges with nonempty $\sigma(\cdot)$ sets form a contiguous path in T. Moreover, if we make the DFS visit larger subtrees first, this path will have length $O(\log n)$. To see why, note that $\sigma(e)$ is empty while we visit the first (largest) child subtree of e; after that, each subtree is at most half as large as the subtree rooted at e itself. Since the maximum subtree size is $O(n)$, this situation cannot happen more than $O(\log n)$ times, limiting the total space used by the temporary $\sigma(\cdot)$ sets to $O(|\mathcal{T}| \log n) = O(m \log n)$. The worse-case time required by this precomputation step is also reasonable.

Lemma 3. *All $\tilde{\sigma}(\cdot)$ sets can be computed in $O(m\Delta \log n)$ total time.*

Proof. First, note that each vertex v is scanned $O(\deg(v))$ times (once for each tree it belongs to), thus bounding the total scanning time to $O(m\Delta)$, where Δ is the maximum degree. Maintaining the $\sigma(\cdot)$ sets during the execution is slightly more expensive. To bound the total time to process a single tree T, we note that each edge $e \notin T$ that is adjacent to T will create an entry in some set $\sigma(e)$, then "bubble up" the tree as sets are merged with parent sets. Eventually, each such entry will either be discarded (if the parent already has an entry corresponding to *tree*(e)) or will end up at the root. The total time (over all trees) spent on such original insertions and on deletions is $O(m\Delta)$, since any edge may participate in at most $2(\Delta - 1)$ original insertions (once for each tree incident to its endpoints). We still have to bound the time spent copying the elements of some (final) set $\sigma(e)$ to another (temporary) set $\sigma(p)$, where p is the parent edge of e. First, note that we only need to insert elements from a smaller set into a bigger one; otherwise, we just swap the entire sets (which costs $O(m)$ total time, since each edge is associated with at most one swap). So consider the case in we transfer elements from a set J into another set K (with $|J| \leq |K|$) while processing a tree T. Only elements in $J' = J \setminus K$ are actually inserted into K; the remaining are deleted (and the corresponding cost has already been accounted for). If $|J'| \leq |J|/2$, we can charge each insertion to a corresponding deletion in J. If $|J'| > |J|/2$, we have a *heavy transfer*. Note that $|J \cup K| = |J'| + |K| > |J|/2 + |J| = 3|J|/2$. Since the target set $(J \cup K)$ is bigger than the original set (J) by a constant factor, each entry (set element) can be involved in at most $O(\log |T|)$ heavy transfers. Considering that there are $O(m\Delta)$ entries overall (across all trees) and $|T| = O(n)$ for any tree T, the lemma follows. $\square$

5.3 Subdivision-based Assignments

A third strategy we use is the *subdivision-based forced assignment*, which works by implicitly subdividing heavy trees in $\mathcal{T}$. Let v be a free vertex reachable from A in G_f. If v were assigned to A, we could obtain a new tree packing $\mathcal{T}'$ by splitting each tree $T_i \in \mathcal{T}(v)$ into multiple trees, one for each edge of T_i that is incident to v. If $f + p(\mathcal{T}') \geq U$, we can safely assign v to B.

Some care is required to implement this test efficiently. In particular, to recompute the packing bound we need to compute the total weight allocated to each newly-created tree. To do so efficiently, we must precompute some information about the original packing $\mathcal{T}$. (This precomputation happens once for each branch-and-bound node, after $\mathcal{T}$ is known.) We define *size*(e) as the weight of the subtree of *tree*(e) rooted at e: this is the sum, over all vertices descending from e in *tree*(e), of the (fractional) weights allocated to *tree*(e). (If e belongs to no tree, *size*(e) is undefined.) The *size*(e) values can be computed with bottom-up traversals of all trees, which takes $O(m)$ total time.

These precomputed values are useful when the forced assignment routine processes a vertex v. Consider an edge e incident to v. If e is a child edge for v, it will generate a tree of size *size*(e). If e is a parent edge for v, the size of the new tree will be *size*$(r(e))$ − *size*(e), where $r(e)$ is the root edge of *tree*(e).

5.4 Recomputing Bounds

Note that all three forced-assignment techniques we consider need to compute a new packing bound $p(\mathcal{T}')$ for each vertex v they process. Although they need only $O(\deg(v))$ time to (implicitly) transform $\mathcal{T}$ into $\mathcal{T}'$, actually computing the packing bound from scratch can be costly. Our implementation uses an incremental algorithm instead. When determining the original $p(\mathcal{T})$ bound (with the greedy algorithm described in Section 3), we remember the entire state of the computation, including the sorted list of all original tree weights as well as relevant partial sums. To compute $p(\mathcal{T}')$, we can start from this initial state, discarding original trees that are no longer valid and considering new ones appropriately.

6 Decomposition

Both lower bounds we consider depend crucially on the degrees of the vertices already assigned. More precisely, let D_A and D_B be the sum of the degrees of all vertices already assigned to A and B, respectively, with $D_A \geq D_B$ (without loss of generality). It is easy to see that the flow bound cannot be larger than D_B, and the packing bound is usually bounded by roughly $D_A/2$, half the number of trees (unless a significant fraction of the vertices is already assigned).[4] If the maximum degree in the graph is a small constant (which is often the case on meshes, VLSI instances, and road networks, for example), our branch-and-bound algorithm cannot prune anything until deep in the tree. Arguably, the dependency on degrees should not be so strong. The fact that increasing the degrees of only a few vertices could make a large instance substantially easier to solve is counter-intuitive.

A natural approach to deal with this weakness is branching on entire regions (connected subgraphs) at once. We would like to pick a region and add *all* of its vertices to A in one branch, and all to B in the other. Since the "degree" of the region (i.e., the number of outgoing edges) is substantially larger, lower bounds should increase much faster as we go down the branch-and-bound tree. The obvious problem with this approach is that the optimal bisection may actually split the region itself; assigning the entire region to A or to B does not exhaust all possibilities.

One way to overcome this, is to make the algorithm probabilistic. Intuitively, if we contract a small number of random edges (merging both endpoints of each edge into a single, heavier vertex), with reasonable probability none of them will actually be cut in the minimum bisection. If this is the case, the optimum solution to the contracted problem is also the optimum solution to the original graph. We can boost the probability of success by repeating this procedure multiple times (with multiple randomly selected contracted sets) and picking the best result found. With high probability, it will be the optimum.

Probabilistic contractions are a natural approach for cut problems, and indeed known. For example, they feature prominently in Karger and Stein's randomized global minimum-cut algorithm [42], which uses the fact that contracting a random edge is unlikely to affect the solution. This idea has been used for the minimum bisection problem as well. Bui et al. [11] use contraction within a polynomial-time method which, for any input graph, either outputs the minimum bisection or halts without output. They show the algorithm has good average performance on a particular class of d-regular graphs with small enough bisections.

Since our goal is to find provably optimum bisections, probabilistic solutions are inadequate. Instead, we propose a contraction-based *decomposition algorithm*, which is *guaranteed* to output the optimum solution for *any input*. It is (of course) still exponential in the worst case, but for many inputs it has much better performance than our standard branch-and-bound algorithm.

The algorithm works as follows. Let U be an upper bound on the optimum bisection. First, partition the set E of edges into $U+1$ disjoint sets $(E_0, E_1, \ldots, E_U)$. For each subset E_i, create a corresponding (weighted) graph G_i by taking the input graph G and contracting all the edges in E_i. (We contract an edge (u, v) by combining u and v into a single vertex with weight $w(u) + w(v)$, redirecting edges incident to u or

[4] For an intuition on the $D_A/2$ bound, recall that the packing bound must accumulate enough trees to account for half the total weight reachable from A. Even if all D_A trees have exactly the same weight, roughly half of the trees will be enough for this. This is not a strict bound because it the number of trees needed also depends on how many vertices are already assigned to A and B.

v to the new vertex.) Then, we use our standard algorithm to find the optimum bisection U_i of each graph G_i independently, and return the best (lowest-value) such solution.

Theorem 3. *The decomposition algorithm finds the minimum bisection of G.*

Proof. Let $U^* \leq U$ be the cost of the minimum bisection. We have to prove that $\min_{0 \leq i \leq U}(U_i) = U^*$. First, note that $U_i \geq U^*$ for every i, since any bisection of G_i can be trivially converted into a valid bisection of G. Moreover, we argue that the solution of at least one G_i must correspond to the optimum solution of G itself. Let E^* be the set of cut edges in an optimum bisection of G. (If there is more than one optimum bisection, take one arbitrarily.) Because $|E^*| = U^*$ and the E_i sets are disjoint, $E^* \cap E_i$ can only be nonempty for at most U^* sets E_i. Therefore, there is at least one j such that $E^* \cap E_j = \emptyset$. Contracting the edges in E_j preserves the optimum bisection, proving our claim. □

The decomposition algorithm solves $U + 1$ subproblems, but the high-degree vertices introduced by the contraction routine should make each subproblem much easier for our branch-and-bound routine. Besides, the subproblems are not completely independent: they can all share the same best upper bound. In fact, we can think of the algorithm as traversing a single branch-and-bound tree whose root node has $U + 1$ children, each responsible for a distinct contraction pattern. The subproblems are not necessarily disjoint (the same partial assignment may be reached in different branches), but this does not affect correctness.

Finally, we note that solving $U + 1$ subproblems is necessary if we actually want to find a solution of value U if one exists. If we already know a solution with U edges and just want to prove it is optimal, it suffices to solve U subproblems: at least one of them would preserve a solution with fewer than U, if it existed. In particular, if we start running the algorithm with upper bound U and find an improving solution $U' < U$, we can stop after the first U' subproblems are solved. For the remainder of this section, we assume that only U initial subproblems are generated.

6.1 Finding a Decomposition

Our decomposition algorithm is correct regardless of how edges are partitioned among subproblems, but its performance may vary significantly. To make all subproblems have comparable degree of difficulty, a natural approach is to allocate roughly the same number of edges to each subproblem. Moreover, the choice of *which* edges to allocate to each subproblem G_i also matters. The effect on the branch-and-bound algorithm is more pronounced if we can create vertices with much higher degree than in the original graph. We can achieve this by making sure the edges assigned to E_i induce relatively large connected components (or *clumps*) in G. (In contrast, if all edges in E_i are disjoint, the degrees of the contracted vertices in G_i will not be much higher that those of other vertices.) Finally, the shape of each clump matters: all else being equal, we would like the expansion (number of incident edges) of each clump to be as large as possible.

To achieve these goals, we perform the decomposition in two stages: *clump generation* partitions all the edges in the graph into clumps, while *allocation* assigns the clumps to subproblems so as to balance the effort required to solve each subproblem. We discuss each stage separately.

Clump Generation Our clump generation routine must build a set F of clumps (initially empty) that partition all the edges in the graph. We combine two approaches to accomplish this. The *basic* clump generation routine is based on extracting paths from random BFS trees within the graph. This approach works reasonably well, and can be used to find a complete set of clumps, i.e., one that covers all edges in the graph. Our second approach, *flow-based* clump generation, is more focused: its aim is to find a small number (comparable to U) of good-quality clumps. We discuss each approach in turn, then explain how they can be combined into a robust clump-generation scheme.

Basic clump generation. Our basic clump generation routine maintains a set C of candidate clumps, consisting of high-expansion paths that are not necessarily disjoint and may not include all edges in the graph.

The algorithm adds new clumps to C until there are enough candidates, then transfers some of the clumps from C to F. This process is repeated until F is complete (its clumps contain all edges in the graph).

More precisely, our algorithm works in iterations, each consisting of two phases: generation and selection. Consider iteration i, with a certain *expansion threshold* τ_i (which decreases as the algorithm progresses).

The *generation phase* creates new clumps to be added to the candidate set C. Our clumps are paths obtained by following parent pointers in BFS trees, which tend to have relatively high expansion because the subgraph of G induced by such a path is itself a path. (Note that this is not true for arbitrary non-BFS paths.) More precisely, we pick 5 vertices at random and grow BFS trees from them, breaking ties in favor of parents that lead to paths with higher expansion. From each tree T, we greedily extract a set of edge-disjoint paths to add to C (in decreasing order of expansion). When doing so, we restrict ourselves to paths that (1) have at most $\lceil m/(4U) \rceil$ edges (ensuring each subproblem has at least four clumps), (2) contain no edge that is already in F, and (3) have expansion at least $\tau_i/4$ (avoiding very small clumps to save edges for future iterations).

The *selection phase* extracts from C all clumps with expansion at least τ_i in decreasing order of expansion. A clump c is added to F if no edge in c is already in F; otherwise, it is simply discarded. To save space, we also discard from C clumps that have at least one edge that either (1) belongs to F or (2) appears in at least five other clumps with higher expansion.

For the next iteration, we set $\tau_{i+1} = \lfloor 0.9\tau_i \rfloor$. We stop when the threshold reaches 0, at which point we simply add all remaining unused edges to F as separate clumps. For speed, we grow smaller trees as the algorithm progresses, and only pick as roots vertices adjacent to at least one unused edge.

Flow-based clumps. We now consider an alternative clump generation scheme aimed at finding a small number of good-quality clumps. It is based on the observation that clumps containing a cut edge from the optimum bisection tend to lead to large packing bounds. To understand why, consider the alternative: if a clump is entirely contained within one side of the optimum bisection, the associated packing bound cannot be higher than *opt* (the optimum bisection value), since at most *opt* disjoint trees will reach the other side. This indicates that, to obtain large packing bounds, we should prefer clumps that actually cross the optimum bisection. Although the basic scheme described above can create such clumps, it does not actively look for them. We now describe an alternative that does; we call it the *flow-based* clump generation scheme.

It works in three stages. First, we quickly find an approximate minimum bisection, i.e., a small cut that roughly divides the graph in half. (The actual optimal solution would be ideal.) Second, we define two regions that are far from this cut, one on each side. Finally, we compute the maximum flow between these two regions, and use the paths in the corresponding flow decomposition as clumps. The remainder of this section describes how we implement each of these steps; other reasonable implementations are possible.

To find a quick approximate bisection, we first compute the *Voronoi diagram* with respect to two random vertices v_0 and v_1. In other words, we split V into two sets, R_0 and R_1, such that each element in R_i is closer to v_i than to v_{1-i}; this can be done with a single BFS [52]. Our tentative cut is the set S of *boundary vertices* in the Voronoi diagram, i.e., those with a neighbor in another region. Although this approach is very fast, we may be unlucky in our choice of v_0 and v_1 and end up with a very unbalanced cut, with $w(R_0) \ll w(R_1)$. (Without loss of generality, assume that $w(R_0) \leq w(R_1)$.) To avoid this, we actually try U random pairs (v_0, v_1), where U is the upper bound on the solution value. We then pick the cut with the highest *score* $s = r^2/b$, where $r = w(R_0)/w(R_1)$ and b is the number of boundary edges in the Voronoi diagram. Intuitively, the score measures how close we are to the optimal bisection: we want both regions to have roughly the same size and, among those, prefer smaller cuts.

After picking the highest-scored pair (R_0, R_1), we define subsets $Q_0 \subseteq R_0$ and $Q_1 \subseteq R_1$ to act as source and sink during our flow computation. We do so in a natural way. For a given parameter $0 < \alpha < 1$ and for each $i \in \{0, 1\}$, we run a BFS restricted to R_i starting from $S \cap R_i$ and stopping after the total weight of all scanned vertices is about to exceed $\alpha \cdot w(R_i)$; we take the remaining (unscanned) vertices as Q_i.

Initially, we pick α such that the distance from Q_0 to Q_1 is closest to $\lceil m/(4U) \rceil$, the target clump size. (To avoid pathological cases, we also enforce that $\alpha \leq 0.8$.) Once the initial flow is computed, we remove the edges in the corresponding flow from the graph and repeat the computation using the same Q_0 and Q_1, but with $\alpha' \leftarrow 0.8\alpha$. We stop this process when α falls below 0.2. This choice of parameters is somewhat arbitrary; the algorithm is not very sensitive to them.

Final approach. As already observed, when the sets Q_0 and Q_1 happen to be in opposite sides of the minimum bisection, the flow-based approach cannot create more than U clumps. In practice, the actual number of clumps is indeed close to U. We take these as the initial clumps in our set F, then apply the BFS-based clump generation scheme to obtain the remaining clumps. The initial threshold τ_0 for the basic clump generation scheme is set to the maximum expansion among all initial flow-based clumps.

Clump Allocation Once clumps are generated, we run an *allocation phase* to distribute them among the U subproblems $(E_0, E_1, \ldots, E_{U-1})$, which are initially empty. We allocate clumps one at a time, in decreasing order of expansion (high-expansion clumps are allocated first). In each step, a clump c is assigned to the set E_i whose distance to c is maximum (with random perturbations to handle approximate ties). The distance from E_i to c is defined as the distance between their vertex sets, or infinity if E_i is empty. For efficiency, we keep the Voronoi diagram associated with each E_i explicitly and update it whenever a new clump is added.

This unbiased distribution tends to allocate comparable number of edges to each subproblem. In some cases, however, this is not desirable: if a subproblem E_i already has much better clumps than E_j, it pays to add more clumps to E_j. To accomplish this, we propose a *biased distribution.* It works as before, but we associate with each subproblem an extra *bias* parameter, to be multiplied by the standard distance parameter. The bias depends on the quality of the first clump added to each subproblem; we refer to it as the *anchor clump.* The bias makes subproblems with worse anchor clumps more likely to receive additional clumps.

To maximize the effectiveness of this approach, we take special care when selecting the set of anchor clumps. We first compute the packing bound associated with the $\lceil 4U/3 \rceil$ clumps in F with highest expansion, then pick the U clumps with highest (packing) value as the anchors of the U subproblems. We define the *gap* associated with subproblem i as $g(i) = U - \phi(i)$, where $\phi(i)$ is the packing bound associated with i's anchor clump, computed by assigning the clump to A and making B empty in the partial assignment. Note that smaller gaps indicate better anchor clumps, and therefore should result in smaller biases. Simply making the bias proportional to the gap is too aggressive for some instances, resulting in very uneven distributions. Instead, we define $\mathit{bias}(i)$ as zero if $g(i) \leq 0$ (the anchor clump by itself is enough to prune the branch-and-bound tree at the root), and as $\mathit{bias}(i) = 1 + \log_2(U - \phi(i))$ otherwise. The log factor makes the distribution smoother, and the additive term ensures the expression is strictly positive.

7 The Algorithm in Full

Having described the main ingredients of our approach, we are now ready to explain how they fit together within our branch-and-bound routine. We first present an overview of the entire algorithm, then proceed to explain the missing details.

7.1 Overview

We take as input the graph G to be bisected, the maximum imbalance ϵ allowed, and an upper bound U on the solution value. The algorithm returns the optimal solution if its value is less than U; otherwise, it proves that U is a valid lower bound. (Section 7.4 explains how one can pick U if no good upper bound is known.)

We start by running a simple heuristic (detailed in Section 7.2) to decide whether to use the decomposition technique or not. If not, we simply call our core branch-and-bound routine (detailed below). Otherwise, we create a series of U subproblems (as described in Section 6.1), call the core branch-and-bound routine on each subproblem separately, and return the best solution found. As an optimization, if we find an improving solution U' for a subproblem, we use it as an upper bound for subsequent subproblems.

Our core branch-and-bound routine starts with an assignment $(A, B) = (\{v\}, \emptyset)$ at the root (Section 7.5 explains how the initial assigned vertex v is picked). It then traverses the branch-and-bound tree in DFS order (to keep space usage low). Each node of (A, B) of the tree is processed in four steps.

Step 1 computes the flow bound f; the exact algorithm is explained in Section 7.3. If $f \geq U$, we prune by discarding the node and backtracking. If $f < U$ and the corresponding minimum cut (A', B') is balanced enough (if $w(A') \geq W_-$ and $w(B') \geq W_-$), we update the current upper bound ($U \leftarrow f$), remember the corresponding bisection, and prune. Otherwise (if $f < U$ but the cut is not balanced), we remove the flow edges from G (creating G_f), and proceed to the second step.

Step 2 computes a tree packing $\mathcal{T}$ in G_f and the corresponding packing bound $p(\mathcal{T})$, as described in Section 4. If $f + p(\mathcal{T}) \geq U$, we prune. Otherwise, we proceed to the third step.

Step 3 applies the forced assignment rules introduced in Section 5, which may result in some vertices being added to A or B. If an inconsistency is detected (if assigning the same vertex to either A or B would push the lower bound above U, or if either side becomes too heavy), we prune. Otherwise, we proceed to the fourth step.

Step 4 picks a vertex $v \in V \setminus \{A, B\}$ to branch on (using rules explained in Section 7.5) and creates two subproblems, $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$, which are added to the branch-and-bound tree.

Steps 2 and 3 have already been discussed in detail. The remainder of this section focuses on the other aspects of the algorithm: criteria for using decomposition, flow computation, upper bound updates, and branching rules.

7.2 Decomposition

The first step of our algorithm is to decide whether to use decomposition or not. Intuitively, one should use decomposition when the upper bound U on the solution value is significantly higher than the degrees of the first few vertices we branch on, since they would lead to a very deep branch-and-bound tree.

We implement this idea by first computing a best-case estimate of the depth of the branch-and-bound tree without decomposition. We use the assumption that branching on a vertex of degree d increases the overall packing bound by $d/2$ (the intuition for this was explained in Section 6: in good cases, roughly half of the d new trees will be selected by the greedy algorithm). We sort all vertices in the graph in decreasing order of degree and count how many vertices x would be required for the accumulated degree to reach $2U$. If $x \leq \max\{5, \log_2 U\}$, we do not use decomposition, since there are potentially enough high-degree vertices to ensure we have a small branch-and-bound tree. Intuitively, decomposition would require at least one node in each of the U branch-and-bound trees, which is roughly equivalent to a single tree of height $\log_2 U$. Otherwise (if $x > \max\{5, \log_2 U\}$), we compute a rough estimate *cdeg* of the degree of all clumps that would be assigned to each subproblem, given by the average number of edges per subproblem (m/U) multiplied by the average degree of each vertex in the graph ($2m/n$). We only use decomposition if $cdeg \geq 2U$.

We stress that this is (once again) just a heuristic, and it is not hard to come up with instances for which it would make the wrong decision. That said, it is easy to compute and works well enough for the large (and diverse) set of instances considered in our experiments, allowing us to use the same settings for all inputs we test, with no manual tuning. In applications in which a more robust approach is needed, one could simply run both versions of the algorithm (with and without decomposition) in parallel and stop after one of them finishes. This method is guaranteed to be within a factor of two of the best choice in the worst case. Also note that, in general, using decomposition is a safer choice: although it can slow down the entire algorithm by a factor of at most U (even when it is not effective), it can lead to exponential speedups for some instances.

7.3 Flow Computation

Although the flow-based lower bound is valid regardless of the actual edges in the flow, the packing bound is usually better if it has more edges to work with, since fewer vertices tend to be unreachable and trees are more intertwined. We therefore prefer maximum flows that use relatively few edges: instead of using the standard push-relabel approach [29], we prefer an augmenting-path algorithm that greedily sends flows along shortest paths. Specifically, we use the IBFS (*incremental breadth first search*) algorithm [28], which is sometimes slower than push-relabel by a small constant factor on the instances we test, but still faster than computing the packing bound. Note that we could minimize the total number of edges using a minimum-cost maximum flow algorithm, but this would be somewhat slower.

7.4 Upper Bound

As already mentioned, we only update the best upper bound U when the minimum A–B cut happens to be balanced; we use no additional heuristics to find improved valid solutions. This approach works well enough in practice, at least if the initial upper bound U is relatively close to the actual solution. If the initial value is significantly higher, our pruning techniques are ineffective, and the DFS traversal of the branch-and-bound tree can go extremely deep.

In most experiments, we assume no good initial upper bound is known. We therefore simply call the branch-and-bound algorithm repeatedly with increasing values of U, and stop when the bound we prove is actually lower than the input bound. We use $U_1 = 1$ for the first call and $U_i = \lceil 1.05\, U_{i-1} \rceil$ for call $i > 1$. Since the algorithm has exponential dependence on U, solving the last problem (the only one where U_i is greater than the optimum) takes a significant fraction of the total time. The overhead of previous calls is a relatively modest constant factor.

That said, we use a couple of simple strategies that sometimes allow us to increase U_i faster between iterations, thus saving us a small factor. Consider a run of our branch-and-bound routine with upper bound U_i. If we run it without decomposition and compute a lower bound $L_i > U_i$ on the root of the branch-and-bound tree, we set $U_{i+1} = \max\{\lceil 1.05\, U_i \rceil, L_i + 1\}$. If we run the algorithm with decomposition, recall from Section 6.1 that we calculate packing bounds for $4U_i/3$ subproblems to guide the distribution process. Let h be the largest integer such that at least h such subproblems have a lower bound of h or higher. From the proof of Theorem 1, it follows that h is a lower bound on the solution of the original problem. We can then set $U_{i+1} = \max\{\lceil 1.05\, U_i \rceil, h + 1\}$.

7.5 Branching

If the lower bound for a given subproblem (A, B) is not high enough to prune it, we must branch on an unassigned vertex v, creating subproblems $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$. Our experiments show that the choice of branching vertices has a significant impact on the size of the branch-and-bound tree and on the total running time. Intuitively, we should branch on vertices that lead to higher lower bounds on the child subproblems. Given our lower-bounding algorithms, we can infer some properties the branching vertex v should have. Based on these properties, we compute a score for each vertex v, then branch on the vertex with the highest score. We discuss each property in turn (in roughly decreasing order of importance), then explain how they are combined into an overall score.

The first criterion we use is the degree $\deg(v)$ of each vertex v. Branching on high-degree vertices helps both the flow bound (by increasing the capacity out of the source or into the sink) and the packing bound (more trees can be created).

Second, all else being equal, we prefer to branch on vertices that are incident to heavier trees, since this would allow these trees to be split when the packing bound is computed. To measure this, we define a branching parameter $\mathrm{tweight}(v)$. If v is reachable from either A or B in G_f, we set $\mathrm{tweight}(v)$ to the average weight of the trees adjacent to v (the average is weighted by v's own allocation). If v is unreachable from neither A nor B, $\mathrm{tweight}(v)$ is set to the average weight of all trees in the packing (rooted at either A or B).

Third, we avoid branching on vertices reachable from B (the smaller side) in G_f, since splitting trees on this side will not (immediately) help improve the packing bound. We do so by associating a *side penalty* $\mathrm{sp}(v)$ to each vertex v, set to 1 if v is reachable from B, and 10 otherwise.

Fourth, we branch on vertices that are far from both A and B, since having assigned vertices spread around the graph helps maintain the trees balanced in the packing bound. Accordingly, we define $\mathrm{dist}(v)$

as the distance in G from v to the closest vertex in $A \cup B$. A single BFS is enough to find the distances between $A \cup B$ and all vertices. If G is not connected, for vertices v that are not reachable from $A \cup B$ we set $\text{dist}(v) = M + 1$, where M is the maximum finite distance from $A \cup B$.

Finally, it pays to treat disconnected graphs in a special way. Intuitively, it makes sense to branch on larger components first, since they tend to have more impact on the packing bound. We thus associate with each vertex v a value $\text{comp}(v)$, defined as the total vertex weight of the component that contains v.

Putting it all together. The relative importance of these five criteria varies widely across instances. For most instances, using just the degree is enough. The remaining four criteria are useful for robustness: they never hurt much, and can be very helpful for some specific instances. The $\text{comp}(v)$ parameter is crucial for disconnected instances, for example.

We combine all parameters in the most natural way, essentially by taking their product. More precisely, we branch on the vertex v that maximizes $q(v) = (\deg(v)+1)^2 \cdot (\text{tweight}(v)+1) \cdot (\text{dist}(v)+1) \cdot \text{sp}(v) \cdot \text{comp}(v)$. The "+1" terms ensure all factors are strictly positive; a single zero in the expression would render all other factors irrelevant. We take the square of $\deg(v)$ term to reflect the fact that degrees are the most relevant criterion. We emphasize that there is nothing special about this particular expression; other approaches could be used as well.

8 Edge Costs

Our description so far has assumed that all edges in the input graph have unit cost, but our actual implementation supports arbitrary integral costs. This section shows how we accomplish this.

First, we note that all algorithms we described can handle parallel edges. Therefore, if the original edges have small costs, we can simply replace them by parallel (unit) edges. In fact, this is how we deal with potential high-cost edges after contraction. To handle arbitrarily large integral costs efficiently, however, we need something more elaborate.

Extending the flow bound is straightforward: we still perform a standard maximum-flow computation between A and B, using edge costs as capacities. The only difference is when we "remove" the flow to create G_f and compute the packing bound. If the flow through an edge e of capacity $c(e)$ is $f(e)$, removing the flow still leaves an edge of (residual) capacity $r(e) = c(e) - f(e)$. If $c(e) = f(e)$, we just remove the edge, as before.

Tree packing. Extending the packing bound is less straightforward. We generalize tree packings as follows. First, we associate to each tree in the packing an integral *thickness*. Each edge e may belong to more than one tree, but the sum of the thicknesses of these trees must not exceed $r(e)$, the residual capacity of e. Intuitively, each edge allocates portions of its cost to different trees, with the portion allocated to tree t equal to t's thickness. In practice, we split each edge into multiple edges with smaller (potentially different) costs; we then build trees as before, but ensure that every edge within each tree has exactly the same cost (corresponding to the thickness of the tree).

More precisely, we handle large costs by splitting each edge into a collection of parallel subedges, each with a cost taken from a discrete set of values. The exact parameters of this split depend on the average edge cost $\gamma = [\sum_{e \in E} cost(e)]/|E|$. If $\gamma \leq 10$, we simply split each original edge into unit-cost edges. If $\gamma > 10$, we use a scaling approach to compute disjoint sets of trees in decreasing order of thickness. More precisely, for each threshold θ_i (with $\theta_0 > \theta_1 > \ldots > \theta_k = 1$), we create a *partial graph* G_i in which every edge has cost θ_i. For each edge e in the original G, we create $\lfloor cost(e)/\theta_i \rfloor$ parallel edges in G_i. We then use our standard algorithm to compute a tree packing $\mathcal{T}_i$ in G_i. For each edge in $\mathcal{T}_i$, we subtract τ_e from the current cost of the corresponding edge e in G. Note that every tree in $\mathcal{T}_i$ has thickness exactly θ_i. To create our final tree packing $\mathcal{T}$, we simply take the union of the trees in all $\mathcal{T}_i$ packings, with their original thicknesses. We set the initial threshold to $\theta_0 = \lfloor \gamma/\sqrt{10} \rfloor$, then set $\theta_i = \eta \lfloor \theta_{i-1} \rfloor$ between iterations. The step $\eta < 1$ is chosen so as to make the total number of steps approximately equal to $\log_{10} \gamma$.

After the trees themselves are created, we use the algorithm described in Section 4.2 to allocate vertex weights to the trees, but taking thicknesses into account. Intuitively, our goal is to have the ratio between the weight and the thickness to be roughly the same across all trees. Both phases described in Section 4.2 (initial allocation and local search) can be trivially generalized to accomplish this, with no penalty in asymptotic performance. Once again, it suffices to interpret a thick tree as a collection of identical trees with unit thickness.

Calculating the Packing Bound. We know consider how to compute the packing bound associated with a tree packing $\mathcal{T}$ that is *heterogeneous*, i.e., contains trees with different thicknesses. First, note that correctness is not an issue: we could simply create a homogenous packing $\mathcal{T}'$ from $\mathcal{T}$ (by replacing each tree of thickness of k by a collection of k trees of unit thickness); the bound given by Theorem 1 ($f + p(\mathcal{T}')$) would is still valid. Since $\mathcal{T}'$ may be quite large, however, computing such bound explicitly would be expensive.

We therefore generalize the greedy algorithm to deal with heterogeneous tree packings directly. Instead of picking trees in decreasing order of weight, it suffices to process them in decreasing order of *profit*, defined as the ratio between weight and thickness. We take the trees in this order their cumulative weight is about to exceed the target weight W_f; the packing bound is the sum of their thicknesses, together with the fractional weight (rounded up) of the tree at the threshold. Note that this is essentially the fractional knapsack problem.

Forced Assignments. The forced assignment routines described in Section 5 enable us, in some situations, to assign a vertex v to one side of the bisection without the need for an explicitly branch. These routines can be generalized to handle heterogeneous tree packings.

For the plain flow-based assignments, we must add up the thicknesses of the parent edges of v. In the extended flow-based assignment, when considering a path through a child edge e, the increase in flow is equal to minimum between the thicknesses of $\mathit{tree}(e)$ (the tree containing e) and $x_v(e)$ (the expansion tree). Finally, the subdivision-based assignment remains essentially unchanged, except for the fact that newly-created trees may have thickness greater than one.

Decomposition. Our decomposition technique could be generalized, as in the packing bound. First, one splits each high-cost original edge into a small number of cheaper edges (with discrete edge costs). For each such cost, one partitions the corresponding edges into clumps, which are then distributed to different subproblems, as before. We then solve each such subproblem independently (contracting the clump edges as usual), and return the best solution found. Since we require all clumps assigned to a given subproblem to have the same thickness k, solving this subproblem is equivalent to solving k independent subproblems with thickness one. For correctness, therefore, it suffices to have the sum of the thicknesses of all subproblems be at least U, the upper bound on the solution value.

Branching. When branching, we define the degree $\deg(v)$ of a vertex v as the sum of the costs of its incident edges.

9 Experiments

We now evaluate the performance of our branch-and-bound algorithm on several benchmark instances from the literature. Section 9.1 starts our experimental analysis by comparing our method with state-of-the-art approaches from the literature. We show that, although our algorithm is outperformed on some graphs (notably those with high expansion), it is much faster than existing approaches on a wide variety of realistic graph classes, such as sparse graphs with relatively small bisections. This motivates our second set of experiments, reported in Section 9.2, which shows that our approach can solve, for the first time, quite large instances (with tens of thousands of vertices) to optimality. Finally, Section 9.3 studies the relative importance of each of the many elements of our algorithm, such as forced assignments and decomposition strategies.

We implemented our algorithm in C++ and compiled it with full optimization on Visual Studio 2010. All experiments were run on a single core of an Intel Core 2 Duo E8500 with 4 GB of RAM running Windows 7 Enterprise at 3.16 GHz.

9.1 Exact Benchmark Instances

Our first experiment compares our algorithm with the results reported by Hager et al. [30]. They use standard benchmark instances (originally considered by Brunetta et al. [9]) to compare their own quadratic-programming-based algorithm (CQB) with other state-of-the-art methods from the literature: BiqMac [56] (semidefinite programming), KRC [43] (also semidefinite programming), and SEN [61] (multicommodity flows).

Table 1 compares the performance of our algorithm against the results reported by Hager et al. [30]. For each instance, we show the number of vertices (n) and edges (m), the value of the optimum bisection (using $\epsilon = 0$), followed by the number of branch-and-bound nodes (BB) and the running time of our algorithm (TIME). The final four columns show the running times reported by Hager et al. [30] for all competing algorithms (when available). The times reported by Hager et al. [30] are already scaled to reflect the approximate execution time on their machine, a 2.66 GHz Xeon X5355. To make the resulting times consistent with our (slightly faster) machine, we further multiply those results by 0.788.[5]

Note that we do not give any upper bound U to our algorithm; it simply tries increasing values of U as needed, as described in Section 7.4. The statistics we report aggregate over all such runs. For each value of U, we use the automated approach described in Section 7.2 to decide whether to use decomposition or not.

The instances in this experiment are divided in classes according to their properties. Class grid corresponds to $h \times k$ rectangular grids with integral weights picked uniformly at random from the range $[1, 10]$. Class tori is similar, but with extra edges to make the grids toroidal. Each instance in class mixed consists of an $h \times k$ grid (with random edge weights in the range $[1, 100]$) to which extra edges (with random weights in the range $[1, 10]$) are added in order to create a complete graph. Class random contains random graphs of various densities and edge weights in the range $[1, 10]$. Class fem contains finite element meshes. Finally, debruijn contains de Bruijn graphs, which are useful in parallel computer architecture applications.

Table 1 shows that our method can solve all grid, fem, and tori instance in a few hundredths of a second; the difference to other approaches increases significantly with the size of the instances, indicating that our algorithm is asymptotically faster. Our algorithm also does well on debruijn instances, on which it is never far from the best algorithm.

For mixed and denser random instances, however, our method is clearly outperformed, notably by the methods based on semidefinite programming (BiqMac and KRC). For these instances, the ratio between the solution value and the average degree is quite large, so we can only start pruning very deep in the tree. Decomposition would not help, since it would contract very few edges per subproblem.

Intuitively, our method does well when the number of edges in the bisection is a small fraction of the total edge weight, which is not the case for mixed and random, but is definitely the case for grid-like graphs.

Our second experiment considers benchmark problems compiled by Armbruster [2] from previous studies. They include instances used in VLSI design (alue, alut, diw, dmxa, gap, taq) [24, 41], finite element meshes (mesh) [24], random graphs (G) [38], random geometric graphs (U) [38], as well as graphs derived from sparse symmetric linear systems (KKT) [32] and compiler design (cb) [39]. In each case, we use the same value of ϵ tested by Armbruster, which is either 0 or 0.05.

Table 2 reports the results obtained by our algorithm. For comparison, we also show the best running times obtained by Armbruster et al. [1, 3, 4] (using linear or semidefinite programming, depending on the instance) and by Hager et al. [30] (using CQB). As before, we multiply the times reported by Hager et al. by 0.788; similarly, we multiply the times reported by Armbruster [1] (on a 3.2 GHz Pentium 4 540) by 0.532 for consistency with our machine. Results for other algorithms (such as KRC and BiqMac) are only available for a tiny fraction of the instances in this table, and are thus discussed in the text only whenever appropriate.

The table includes all instances that can be solved by at least one method in less than 150 minutes in our machine (roughly corresponding to the 5-hour time limit set by Armbruster [1]), except those that can be solved in less than 5 seconds by both our method and Armbruster's. Note that we can solve every instance in the table to optimality. Although our method can be slightly slower than Armbruster's (notably on alue6112.16896), it is is usually much faster, often by orders of magnitude. We can solve in minutes (or even seconds) several instances no other method can handle in hours.

[5] The scaling factor was obtained from `http://www.cpubenchmark.net/singleThread.html`.

Table 1. Performance on standard benchmark instances. Columns indicate number of vertices (n) and edges (m), optimum bisection value (opt), number of branch-and-bound nodes (BB) for our algorithm, and running times in seconds for our method (TIME) and others (CQB, BiqMac, KRC, SEN); "—" means "not tested" and DNF means "not finished in 2.5 hours".

CLASS	NAME	n	m	opt	BB	TIME	CQB	BiqMac	KRC	SEN
grid	2x16	32	46	8	29	**0.01**	0.05	0.06	0.10	—
	18x2	36	52	6	32	**0.01**	0.06	0.04	0.05	—
	2x19	38	55	6	60	**0.02**	0.09	2.91	1.44	—
	5x8	40	67	18	36	**0.02**	0.08	0.06	0.05	—
	3x14	42	67	10	47	**0.02**	0.10	0.56	0.47	—
	5x10	50	85	22	47	**0.03**	0.22	0.25	0.24	—
	6x10	60	104	28	63	**0.03**	0.26	3.59	8.38	—
	7x10	70	123	23	83	**0.05**	0.41	17.05	14.63	—
fem	m4.i	32	50	6	31	**0.01**	0.03	0.03	0.02	0.11
	ma.i	54	72	2	17	**0.01**	0.13	0.04	0.08	0.22
	me.i	60	96	3	12	**0.01**	0.16	0.11	0.10	0.22
	m6.i	70	120	7	30	**0.01**	0.35	0.54	0.97	1.13
	mb.i	74	120	4	39	**0.01**	0.29	0.49	0.77	0.90
	mc.i	74	125	6	41	**0.01**	0.32	0.39	1.21	1.13
	md.i	80	129	4	38	**0.01**	0.42	0.69	0.76	1.01
	mf.i	90	146	4	36	**0.01**	0.53	1.12	0.63	1.46
	m1.i	100	155	4	57	**0.02**	0.78	14.77	28.76	2.36
	m8.i	148	265	7	46	**0.02**	3.43	2.41	8.43	3.26
mixed	2x17m	34	561	316	32971	22.07	1.21	0.91	**0.76**	—
	10x4m	40	780	436	180068	158.64	2.99	**0.04**	0.05	—
	5x10m	50	1225	670	6792445	8006.36	223.14	0.11	**0.05**	—
	13x4m	52	1326	721	DNF	DNF	853.07	**0.50**	0.89	—
	4x13m	52	1326	721	DNF	DNF	571.58	**0.52**	0.89	—
	9x6m	54	1431	792	DNF	DNF	2338.03	0.42	**0.32**	—
	10x6m	60	1770	954	DNF	DNF	2051.07	0.24	**0.20**	—
	10x7m	70	2415	1288	DNF	DNF	3975.26	**0.31**	0.36	—
random	q0.00	40	704	1606	DNF	DNF	294.11	0.19	**0.10**	—
	q0.10	40	647	1425	DNF	DNF	159.23	2.83	**1.07**	—
	q0.20	40	566	1238	4122814	5541.47	52.02	3.03	**0.65**	—
	q0.30	40	506	1056	949311	1261.00	29.48	4.42	**1.62**	—
	q0.80	40	145	199	1004	0.52	**0.16**	5.65	1.81	—
	q0.90	40	78	63	175	**0.06**	0.09	0.23	0.10	—
	c0.00	50	1108	2520	DNF	DNF	1892.41	10.98	**3.68**	—
	c0.10	50	1003	2226	DNF	DNF	1406.71	12.21	**4.18**	—
	c0.30	50	802	1658	DNF	DNF	1743.65	18.45	**4.30**	—
	c0.70	50	350	603	105712	107.94	12.49	13.31	**6.35**	—
	c0.80	50	235	368	11056	8.61	**1.46**	9.91	4.83	—
	c0.90	50	130	122	388	0.22	**0.18**	0.28	0.26	—
	c2.90	52	137	123	354	0.21	**0.20**	0.36	0.32	—
	c4.90	54	149	160	993	0.53	**0.32**	0.52	2.60	—
	c6.90	56	166	177	1340	0.79	0.52	**0.30**	0.79	—
	c8.90	58	179	226	4993	2.94	**1.56**	22.42	13.81	—
	s0.90	60	195	238	2630	1.76	**1.16**	4.38	7.80	—
tori	8x5	40	80	33	108	**0.03**	0.09	0.13	0.16	—
	21x2	42	63	9	24	**0.01**	0.02	0.17	0.13	—
	23x2	46	69	9	78	**0.02**	0.13	8.63	3.28	—
	4x12	48	96	24	39	**0.02**	0.16	0.23	0.44	—
	5x10	50	100	33	80	**0.03**	0.17	0.30	0.16	—
	6x10	60	120	35	77	**0.05**	0.34	4.75	9.19	—
	7x10	70	140	45	140	**0.08**	0.57	0.64	15.02	—
	10x8t	80	160	43	93	**0.05**	0.64	3.75	24.79	—
debruijn	debr5	32	61	10	145	0.02	0.06	0.06	0.16	**0.00**
	debr6	64	125	18	2583	0.60	**0.38**	0.66	12.32	0.79
	debr7	128	253	30	109039	19.67	2436.76	371.89	2204.00	**8.10**

Table 2. Performance of our algorithm compared with the best times obtained by Armbruster [1] and Hager et al. [30]. Columns indicate number of nodes (n), number of edges (m), allowed imbalance (ϵ), optimum bisection value (opt), number of branch-and-bound nodes (BB), and running times in seconds for our method (TIME) and others ([Arm07], CQB); "—" means "not tested" and DNF means "not finished in 2.5 hours".

NAME	n	m	ϵ	opt	BB	TIME	[Arm07]	CQB
G124.02	124	149	0.00	13	376	**0.06**	7.40	3.32
G124.04	124	318	0.00	63	166799	**35.11**	2334.24	751.46
G250.01	250	331	0.00	29	42421	**10.86**	974.76	7963.64
KKT_capt09	2063	10936	0.05	6	30	**0.15**	619.72	3670.97
KKT_skwz02	2117	14001	0.05	567	902	**10.97**	DNF	—
KKT_plnt01	2817	24999	0.05	46	1564	**22.32**	DNF	—
KKT_heat02	5150	19906	0.05	150	4346	**83.63**	DNF	—
U1000.05	1000	2394	0.00	1	45	**0.03**	28.53	—
U1000.10	1000	4696	0.00	39	315	**1.09**	883.46	—
U1000.20	1000	9339	0.00	222	16502	**149.40**	DNF	—
U500.05	500	1282	0.00	2	34	**0.02**	10.54	—
U500.10	500	2355	0.00	26	95	**0.16**	263.82	—
U500.20	500	4549	0.00	178	27826	**104.34**	DNF	—
U500.40	500	8793	0.00	412	232593	**2159.34**	DNF	—
alut2292.6329	2292	6329	0.05	154	17918	**201.44**	208.42	—
alue6112.16896	6112	16896	0.05	272	239815	7778.05	**2539.85**	—
cb.47.99	47	99	0.00	765	141	**0.08**	2.81	0.23
cb.61.187	61	186	0.00	2826	193	**0.63**	43.28	**0.63**
diw681.1494	681	1494	0.05	142	3975	**8.25**	DNF	—
diw681.3103	681	3103	0.05	1011	5659	**129.82**	DNF	—
diw681.6402	681	6402	0.05	330	1234	**8.26**	2436.09	—
dmxa1755.10867	1755	10867	0.05	150	2417	**31.98**	DNF	—
dmxa1755.3686	1755	3686	0.05	94	8233	**45.93**	1049.22	—
gap2669.24859	2669	24859	0.05	55	11	**0.24**	185.64	—
gap2669.6182	2669	6182	0.05	74	2424	**24.93**	346.35	—
mesh.138.232	138	232	0.00	8	87	**0.02**	5.44	5.45
mesh.274.469	274	469	0.00	7	57	**0.03**	4.53	19.40
taq170.424	170	424	0.05	55	246	**0.51**	15.26	—
taq334.3763	334	3763	0.05	341	2816	**5.07**	DNF	—
taq1021.2253	1021	2253	0.05	118	3009	**9.88**	90.25	—

Our approach is significantly faster than Hager et al.'s for mesh, KKT, and sufficiently sparse random graphs (G). For the cb class, the algorithms have similar performance (KRC—not shown in the table—has comparable running times as well). These instances are small but its edges are costly, which means our algorithm is only effective because of the scaling strategy described in Section 8. Without scaling, it would be two orders of magnitude slower.

With longer runs, Armbruster [1], Hager et al. [30], and BiqMac [56] can solve denser random graphs G124.08 and G124.16 in a day or less (not shown in the table). We would take about 3 days on G124.08, and a month or more for G124.16. Once again, this shows that there are classes of instances in which our method is clearly outperformed.

The solutions we report were in most cases known; we just proved their optimality. We did find better solutions than those reported by Armbruster [1] in three cases: KKT_plnt01, diw681.6402, and taq334.3764. For the last two, however, Armbruster claims matching lower bounds for his solutions. This discrepancy could in principle be due to slightly different definitions of W_+, the maximum allowed cell size. We define it as in the 10th DIMACS Implementation challenge [5] ($W_+ \leq \lfloor(1+\epsilon)\lceil W/2\rceil\rfloor$) whereas Armbruster [1] uses $W_+ \leq \lceil(1+\epsilon)W/2\rceil$; these values can differ very slightly for some combinations of W and ϵ. The solutions we found, however, obey both definitions. For KKT_plnt01, we found a solution with cut size 46, $w(A) = 1479$, $w(B) = 1338$, and an imbalance (defined as $\max\{w(A), w(B)\}/\lceil W/2\rceil$) of 4.968%. For diw681.6402, we computed a solution of 330 with $w(A) = 1350$ and $w(B) = 1221$ (imbalance 4.977%), while for taq334.3764 our solution of 341 has $w(A) = 556$ and $w(B) = 503$ (imbalance 4.906%). We conjecture that Armbruster's code actually uses $W_+ \leq (1+\epsilon)W/2$ (without the ceiling). Indeed, with $\epsilon = 0.049$ our code finds the same solutions as Armbruster does (49, 331, and 342, respectively). Our running times are essentially the same with either value of ϵ.

9.2 Larger Benchmark Instances

We now show that we can actually solve real-world instances that are much larger than those shown in Table 2. In particular, we consider instances from the 10th DIMACS Implementation Challenge [5], on Graph Partitioning and Graph Clustering. They consist of social and communication networks (class clustering), road networks (streets), Delaunay triangulations (delaunay), random geometric graphs (rgg), planar maps representing adjacencies among census blocks in US states (redistrict), and assorted graphs (walshaw) from the Walshaw benchmark [64] (mostly finite-element meshes). We stress that these testbeds were created to evaluate heuristics and were believed to be beyond the reach of exact algorithms. To the best of our knowledge, no exact algorithm has been successfully applied to these instances.

Table 3 reports the detailed performance of our algorithm. For each case, we show the number of vertices (n) and edges (m), the optimum bisection value (opt), the total number of nodes in the branch-and-bound tree (BB), and the total running time of our algorithm in seconds. We use $\epsilon = 0$ for all classes but one: since vertices in redistrict instances are weighted (by population), we allow a small amount of imbalance ($\epsilon = 0.03$). We only report instances that our algorithm can solve within a few hours; most of the instances available in the repository are much larger (with up to tens of millions of vertices) and beyond the capabilities of our method. Since we deal with very large instances in this experiment, we save time by running our algorithm with $U = opt + 1$. Running times would increase by a small constant factor if we ran the algorithm repeatedly with increasing values of U. As usual, we use the heuristic described in Section 7.2 to decide whether to use decomposition in each case.

The table shows that our method can solve surprisingly large instances, with up to hundreds of thousands of vertices and edges. In particular, we can easily handle luxembourg, a road network with more than 100 thousand vertices; the fact that the bisection value is relatively small certainly helps in this case. Instances of comparable size from the redistrict class are harder, but can still be solved in a few minutes. We can also find the minimum bisections of rather large walshaw instances [64], which are mostly finite element meshes. For every such instance reported in the table, we show (for the first time, to the best of our knowledge) that the best previously known bisections, which were found by heuristics [6, 12, 31, 34, 48, 63] with no time limit, are indeed optimal.

Our algorithm can also deal with fairly large Delaunay triangulations (delaunay) and random geometric graphs (rgg). The sets of vertices in both classes correspond to points picked uniformly at random within a square, and differ only in how edges are added. Although random geometric graphs are somewhat denser than the triangulations, the actual cut sizes (bisections) are not much bigger. Our algorithm can then allocate more edges to each subproblem (during decomposition), which explains why it works better on rgg graphs. Our algorithm can also find exact solutions for some clustering graphs, even though the minimum bisection value is much larger relative to the graph size.

The rule described in Section 7.2 causes our algorithm to use decomposition for all instances in Table 3, except those from classes clustering (which can have very high-degree vertices) and redistrict (which are sparse but have a few high-degree vertices). For a small fraction of the redistrict instances, however, using decomposition would be slightly faster: solving nj2010, for example, would be four times quicker with decomposition. (The solution to this instance is shown in Figure 4(a), in Appendix B.) As anticipated, the rule is not perfect, but works well enough for most instances.

To further illustrate the usefulness of our approach, Table 4 considers additional natural classes of large instances with relatively small (but nontrivial) bisections. We test three classes of inputs: cgmesh (meshes representing various objects [57], commonly used in computer graphics applications), road (road networks from the 9th DIMACS Implementation Challenge [21]), and steinlib (sparse benchmark instances for the Steiner problem in graphs [45], mostly consisting of grid graphs with holes representing large-scale circuits).

Table 3. Performance of our algorithm on DIMACS Challenge instances starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds. We use $\epsilon = 0$ for all classes but redistrict, which uses $\epsilon = 0.03$.

CLASS	NAME	n	m	opt	BB	TIME
clustering	karate	34	78	10	4	0.02
	chesapeake	39	170	46	26	0.03
	dolphins	62	159	15	32	0.02
	lesmis	77	254	61	17	0.03
	polbooks	105	441	19	7	0.02
	adjnoun	112	425	110	12 488	4.13
	football	115	613	61	2 046	1.17
	jazz	198	2 742	434	70 787	160.37
	celegansneural	297	2 148	982	83	0.78
	celegans_metabolic	453	2 025	365	359	0.66
	polblogs	1 490	16 715	1 213	327 740	7 748.64
	netscience	1 589	2 742	0	363	0.20
	power	4 941	6 594	12	71	0.32
	PGPgiantcompo	10 680	24 316	344	49 381	1 100.62
	as-22july06	22 963	48 436	3 515	4 442	279.31
delaunay	delaunay_n10	1 024	3 056	63	1 422	3.76
	delaunay_n11	2 048	6 127	86	3 698	17.46
	delaunay_n12	4 096	12 264	118	9 322	100.49
	delaunay_n13	8 192	24 547	156	18 245	442.18
	delaunay_n14	16 384	49 122	225	599 012	31 281.53
redistrict	de2010	24 115	58 028	36	43	3.08
	hi2010	25 016	62 063	44	887	42.10
	ri2010	25 181	62 875	107	921	73.33
	vt2010	32 580	77 799	112	1 398	137.09
	ak2010	45 292	108 549	48	61	6.62
	nh2010	48 837	117 275	146	12 997	2 125.96
	ct2010	67 578	168 176	150	11 081	2 988.57
	me2010	69 518	167 738	140	12 855	3 215.64
	nv2010	84 538	208 499	126	883	259.03
	wy2010	86 204	213 793	190	6 113	2 098.37
	ut2010	115 406	286 033	198	12 112	5 639.60
	mt2010	132 288	319 334	209	3 728	2 129.32
	wv2010	135 218	331 461	222	76 644	48 675.42
	id2010	149 842	364 132	181	3 822	2 493.53
	nj2010	169 588	414 956	150	26 454	21 455.21
	la2010	204 447	490 317	125	5 116	4 299.09
rgg	rgg15	32 768	160 240	181	3 072	615.04
	rgg16	65 536	342 127	314	28 301	14 064.83
streets	luxembourg	114 599	119 666	17	318	38.22
walshaw	data	2 851	15 093	189	26 098	304.59
	3elt	4 720	13 722	90	860	10.45
	uk	4 824	6 837	19	1 429	6.25
	add32	4 960	9 462	11	22	0.15
	whitaker3	9 800	28 989	127	992	25.66
	crack	10 240	30 380	184	15 271	481.37
	fe_4elt2	11 143	32 818	130	1 189	37.40
	4elt	15 606	45 878	139	1 903	89.83
	fe_pwt	36 519	144 794	340	2 569	458.34
	fe_body	45 087	163 734	262	42 373	6 176.41
	brack2	62 631	366 559	731	34 308	20 327.60
	finan512	74 752	261 120	162	219	37.85

Table 4. Performance on additional large instances with $\epsilon = 0$, starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds.

CLASS	NAME	n	m	opt	BB	TIME
cgmesh	dolphin	284	846	26	101	0.16
	mannequin	689	2 043	61	3 217	5.26
	venus-711	711	2 127	43	182	0.43
	beethoven	2 521	7 545	72	388	2.36
	venus	2 838	8 508	83	443	3.35
	cow	2 903	8 706	79	1 305	7.85
	fandisk	5 051	14 976	137	5 254	68.10
	blob	8 036	24 102	205	756 280	17 127.04
	gargoyle	10 002	30 000	175	18 638	572.57
	face	12 530	36 647	174	48 036	1 704.89
	feline	20 629	61 893	148	13 758	443.24
	dragon-043571	21 890	65 658	148	64 678	4 111.25
	horse	48 485	145 449	355	71 616	12 552.44
road	ny	264 346	365 050	18	976	380.98
	bay	321 270	397 415	18	537	248.13
	col	435 666	521 200	29	3 672	2 164.13
	fla	1 070 376	1 343 951	25	901	1 640.38
	nw	1 207 945	1 410 387	18	166	463.35
	ne	1 524 453	1 934 010	24	206	751.48
	cal	1 890 815	2 315 222	32	733	2 658.27
steinlib	alue5067	3 524	5 560	30	574	2.33
	gap3128	10 393	18 043	52	942	10.89
	diw0779	11 821	22 516	49	150	3.14
	fnl4461fst	17 127	27 352	24	402	8.08
	es10000fst01	27 019	39 407	22	452	14.84
	alut2610	33 901	62 816	93	802	43.74
	alue7065	34 046	54 841	80	4 080	194.52
	alue7080	34 479	55 494	80	4 065	200.88
	alut2625	36 711	68 117	99	791	50.36
	lin37	38 418	71 657	131	14 989	1 184.92

(The solutions to some of these instances are shown in Figure 4, in Appendix B.) Once again, we use $\epsilon = 0$ and $U = opt + 1$. Our algorithm uses decomposition for all instances tested.

As the table shows, we can find the optimum bisection of some road networks with more than a million vertices in less than an hour while traversing very few branch-and-bound nodes. Decomposition is particularly effective on these instances, since a very small fraction of the edges belong to the minimum bisection. Since the graphs themselves are large, however, processing each node of the branch-and-bound tree can take a few seconds even with our almost-linear-time combinatorial algorithms. The main bottlenecks are finding the tree packing and the flow, which take comparable time. All other elements are relatively cheap: decomposition, allocating vertex weights to trees, the greedy computation of the packing bound (given the trees), and forced assignments. This relative breakdown holds for all instances tested, not just road networks.

The other two classes considered in Table 4 (steinlib and mesh) have larger bisections and need substantially more branch-and-bound nodes. Even so, we can handle instances with tens of thousands of vertices in a few minutes.

Finally, Table 5 shows the results of longer runs of our algorithm on some large benchmark instances. For most instances we use $\epsilon = 0$. The only exception is taq1021.5480, which is usually tested with $\epsilon = 0.05$ in the literature [1]; it is also the only one in the table that does not use decomposition. All runs use $U = opt + 1$.

We employ the standard parameter settings (used for all experiments so far) for all instances but t60k, for which we set the target number of edges per subproblem to $\lceil m/(150U) \rceil$ instead of the usual $\lceil m/(4U) \rceil$.

Table 5. Performance on harder instances with $\epsilon = 0$ (except for taq1021.5480), starting from $U = opt + 1$; BB is the number of branch-and-bound nodes, and TIME is the total CPU time in seconds. The setup varies depending on the instance; see text for details.

CLASS	NAME	n	m	opt	BB	TIME
delaunay	delaunay_n15	32 768	98 274	320	126 053 466	45 358 801
exact	taq1021.5480	1 021	5 480	1 650	3 102 902	105 973
ptv	bel	463 514	591 882	80	34 280	27 049
	nld	893 041	1 139 540	54	17 966	26 466
rgg	rgg17	131 072	728 753	517	34 800	64 072
	rgg18	262 144	1 547 283	823	192 966	671 626
streets	belgium	1 441 295	1 549 970	72	59 562	192 064
	netherlands	2 216 688	2 441 238	45	8 330	56 052
walshaw	t60k	60 005	89 440	79	56 681 055	14 378 268

This is a planar mesh in which most vertices have degree three. This is challenging for our algorithm because trees from the tree packing cannot "cross" the flow, which often causes large fractions of the vertices to be unreachable from either A and B (the assigned vertices). With so much deadweight, the packing bound becomes much less effective.

Note that two of the instances (delaunay_n15 and t60k) took months of CPU time. For them, we actually ran a distributed version of the code using the DryadOpt framework [10]. DraydOpt is written in C#, and calls our native C++ code to solve individual nodes of the branch-and-bound tree. The distributed version was run on a cluster where each machine has two 2.6 GHz dual-core AMD Opteron processors, 16 GB of RAM, and runs Windows Server 2003. We used 100 machines only, and report the total CPU time (the sum of the times spent by our C++ code on all cores). Note that this excludes the communication overhead, which is negligible. The remaining instances in the table were solved sequentially.

To the best of our knowledge, none of these instances has been provably solved to optimality before. Solutions matching the optimum were known for t60k [65], taq1021.5480 [1], and rgg17 [59]. We are not aware of any published solutions for belgium and netherlands. For the remaining three instances, we improve the best previously known solutions, all found by the state-of-the-art heuristic of Sanders and Schulz [59]: 867 for rgg18, 81 for bel, and 64 for nld. (These are the best results for multiple executions of their algorithm; the average results are 1151, 104, and 120, respectively.) This shows that, in some cases, our exact algorithm can be a viable alternative to heuristics. For nld, we improved the best known result by more than 18%, probably because the heuristics may have missed the fact that one of the cells in the optimum solution is disconnected. (See Figure 4(e), in Appendix B, for the solution of a similar instance.) As we have seen, for smaller or more regular instances, state-of-the-art heuristics can find solutions that are much closer to the optimum. Even in such cases, our algorithm can still be useful to calibrate the precise quality of the solutions provided.

9.3 Parameter Evaluation

We now discuss the relative importance of some of the techniques introduced in this article. One of our main contributions is the packing lower bound. Although we also use the (known) flow bound, it is extremely weak by itself, particularly when most the nodes are assigned to the same side. As a result, almost all instances we tested would not finish without the packing bound, even if we allowed a few hours of computation. The only exceptions are tiny instances, which can be solved but are orders of magnitude slower.

In the remainder of this section, we evaluate other important aspects of the algorithm: the decomposition technique, forced assignments, and the branching criterion. In every case, we pick a small set of instances for illustration (chosen to highlight the differences between the strategies), and always run the full algorithm with $U = opt + 1$ and the same value of ϵ as in our main experiments (usually 0). We set a time limit of 12 hours for the experiments in this section.

Table 6. Total running times (in seconds) of our algorithm on assorted instances with different decomposition strategies: no decomposition, random partition of the edges, using BFS-based clumps, and using both flow-based and BFS-based clumps.

CLASS	NONE	RANDOM	BFS	FLOW
G124.04	14.52	499.99	315.36	366.25
alue5067	441.22	225.46	3.12	2.61
cow	572.23	5537.97	12.41	8.53
delaunay_n10	25.94	191.46	4.22	4.02
delaunay_n11	658.88	4885.87	19.86	18.35
dragon-043571	DNF	DNF	21744.69	4128.44
gargoyle	DNF	DNF	1248.64	578.62
mannequin	57.17	198.18	5.40	6.18

We first consider the decomposition technique. Table 6 compares the total running time of our algorithm when different decomposition techniques (introduced in Section 6) are used. We consider four different approaches: (1) no decomposition; (2) random decomposition (each edge is independently assigned to each of the U subproblems at random); (3) decomposition with BFS-based clumps; and (4) decomposition with flow-based and BFS clumps. All remaining aspects of the algorithm remain unchanged. Note that the standard version of our algorithm (used in previous experiments) automatically picks either strategy (1) or strategy (4). Strategies (2) and (3) are shown here for comparison only.

As anticipated, decomposition can be quite useful, but only for a subset of the instances. Instances with relatively large bisections, such as G124.04 (a random graph) become significantly slower if decomposition is used: it has to solve many subproblems, each about as hard as the original ones. In contrast, decomposition is quite helpful for instances with smaller bisections, such as meshes (cow, mannequin, dragon-043571, and gargoyle), grid graphs with holes (alue5067), and Delaunay triangulations. The way we distribute edges to to subproblems is important, however. Random distribution is not enough to make the subproblems much easier; creating clumps is essential. Flow-based clumps have only minor effect in most cases, but never hurt much and are occasionally quite helpful (as in dragon-043571, which is a long and narrow mesh).

Another important contribution is the notion of *forced assignments*. We proposed three strategies: flow-based (Section 5.1), extended flow-based (Section 5.2), and packing-based (Section 5.3). Table 7 compares total running times on some instances using all possible combinations of these methods: (1) no forced assignments; (2) flow-based only; (3) extended flow-based only; (4) packing-based only; (5) flow-based + packing-based; (6) extended flow-based + packing-based (the default version used in our experiment). Note that other combinations are redundant, since extended flow-based assignments are strictly stronger than flow-based assignments.

The table shows that forced assignments are crucial to make our method robust. While it does not help much in some cases (such as random graphs), it introduces very little overhead. For other instances, forgoing forced assignments makes the overall algorithm orders of magnitude slower (this is the case for luxembourg, a road network). Forced assignments are especially helpful in guiding the algorithm towards

Table 7. Total running times (in seconds) on assorted instances using different combinations of forced-assignment techniques (based on flows, extended flows, and subdivisions).

CLASS	NONE	FLOW	EXTENDED	SUBDIV	FLOW+SUBDIV	FULL
G124.04	14.91	14.48	15.30	14.82	14.64	14.65
alue5067	10.63	3.10	2.65	17.44	3.09	2.62
cow	24.43	9.23	8.61	23.72	9.29	8.51
delaunay_n10	4.82	3.62	3.93	4.91	3.48	3.53
luxembourg	DNF	42.72	38.76	DNF	39.33	39.44

Table 8. Total running times (in seconds) on assorted instances using different branching techniques. All columns use degree as a branching criterion, by itself (column DEGREE) or in combination with one additional criterion (columns TREE, SIDE, DISTANCE, CONNECTED). The last column refers to our default branching criterion, which combines all five methods.

CLASS	DEGREE	TREE	SIDE	DISTANCE	CONNECTED	ALL
G124.04	24.69	27.14	24.59	15.85	24.44	15.12
ak2010	560.65	24.04	458.03	2088.31	78.46	6.76
alue5067	5.34	2.71	4.89	2.93	5.37	2.51
cow	8.99	9.32	8.74	8.85	8.91	8.62
delaunay_n10	4.10	4.05	4.06	4.01	4.13	4.07
hi2010	895.37	DNF	574.30	351.95	24.87	42.48

the optimum bisection (the actual balanced cut). Note that most gains come from the simpler flow-based strategy; extended flows and packing are generally helpful, but not crucial.

Finally, we evaluate our branching criteria. Recall, from Section 7.5, that we evaluate each potential branching vertex using a combination of five criteria: the *degree* of the vertex, *weight* of the tree it belongs to, which *side* it is reachable from in G_f, its *distance* to the closest assigned vertex, and the total weight of its *connected* component. Among those, the degree is absolutely crucial: our algorithm becomes orders of magnitude slower without this criterion. In Table 8 we consider the effect of each of the remaining methods (when applied in conjunction with degree) on the total running time of our algorithm.

Although branching based only on degrees is often good enough (as in cow and delaunay_n10), additional criteria usually have a significant positive effect. In particular, taking components into account is crucial for disconnected instances (such as ak2010 and hi2010, redistricting instances representing Alaska and Hawaii), and distance information can be helpful in instances with relatively flat degree distributions, such as G124.04 (a random graph). We stress, however, that these are heuristics, and sometimes they actually hurt, as when one uses only tree weights (in addition to degrees) on hi2010 or distances for ak2010. On balance, however, the combination of all five criteria leads to a fairly robust algorithm.

10 Conclusion

We have introduced new lower bounds for graph bisection that provide excellent results in practice. They outperform previous methods on a wide variety of instances, and help find provably optimum bisections for several long-standing open instances (such as U500.20 [38]). While most previous approaches keep the branch-and-bound tree small by computing very good (but costly) bounds at the root, our bounds are only useful if some vertices have already been assigned. This sometimes causes us to branch more, but we usually make up for it with a faster lower bound computation.

A notable characteristic of our approach is that it relies strongly on heuristics at various steps, such as generating clumps for decomposition, creating valid tree packings, and picking branching vertices. A natural direction for future research is to improve these heuristics so as to realize the full potential of the main theoretical techniques we introduce (packing bound and decomposition). For several classes of instances (such as Delaunay triangulations), it is quite likely that better methods for generating trees and clumps will lead to much tighter bounds (and smaller branch-and-bound trees) in practice.

For some other graph classes (such as high-expansion graphs), however, the tree packings generated by our heuristics are already perfectly balanced. For those, improvements would have to come from additional theoretical insights. Of course, it would be straightforward to create a hybrid algorithm that simply applies another technique (based on semidefinite programming or multicommodity flows, for example) when confronted with such instances. A more interesting question is whether bounds based on such techniques can be combined in a nontrivial way with the ones we propose here. Our decomposition technique, in particular, could be used in conjunction with any exact algorithm for graph bisection, though it is not very effective for such instances.

Another avenue for future research is proving nontrivial bounds for the running time of our algorithm (or a variant) for some graph classes. In particular, Demaine et al. [20] show that one can partition the edges of a minor-free graph into k classes (for any integer k) so that contracting all edges of any class leads to a graph with treewidth $O(k)$. This implies an exact algorithm for the minimum bisection problem on minor-free graphs with running time $O(e^{opt} n^{O(1)})$ [37]. Although far from practical, this theoretical algorithm has some parallels with our approach, and may potentially shed some light into the good empirical performance we observe.

Acknowledgements. We thank Diego Nehab and Dzung T. Phan for providing giving us access to some of the benchmark instances tested here. We also thank Haim Kaplan, Eduardo Uchoa, Manuel Sorge, Bob Tarjan, and Tony Wirth for interesting discussions, and anonymous referees for the conference versions of this article for their helpful comments.

References

1. M. Armbruster. *Branch-and-Cut for a Semidefinite Relaxation of Large-Scale Minimum Bisection Problems.* PhD thesis, Technische Universität Chemnitz, 2007.
2. M. Armbruster. Graph bisection and equipartition, 2007. `http://www.tu-chemnitz.de/mathematik/discrete/armbruster/diss/`.
3. M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin. A comparative study of linear and semidefinite branch-and-cut methods for solving the minimum graph bisection problem. In *Proc. Conf. Integer Programming and Combinatorial Optimization (IPCO)*, volume 5035 of *LNCS*, pages 112–124, 2008.
4. M. Armbruster, M. Fügenschuh, C. Helmberg, and A. Martin. LP and SDP branch-and-cut algorithms for the minimum graph bisection problem: A computational comparison. *Mathematical Programming Computation*, 4(3):275–306, 2012.
5. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *Graph Partitioning and Graph Clustering—10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013. `http://www.cc.gatech.edu/dimacs10/`.
6. S. T. Barnard and H. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency and Computation: Practice and Experience*, 6(2):101–117, 1994.
7. R. Bauer and D. Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009.
8. S. N. Bhatt and F. T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984.
9. L. Brunetta, M. Conforti, and G. Rinaldi. A branch-and-cut algorithm for the equicut problem. *Mathematical Programming*, 78:243–263, 1997.
10. M. Budiu, D. Delling, and R. F. Werneck. DryadOpt: Branch-and-bound on distributed data-parallel execution engines. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1278–1289, 2011.
11. T. N. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.
12. P. Chardaire, M. Barake, and G. P. McKeown. A PROBE-based heuristic for graph partitioning. *IEEE Transactions on Computers*, 56(12):1707–1720, 2007.
13. C. Chevalier and F. Pellegrini. PT-SCOTCH: A tool for efficient parallel graph ordering. *Parallel Computing*, 34:318–331, 2008.
14. T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.
15. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Proc. International Symposium on Experimental Algorithms (SEA)*, volume 6630 of *LNCS*, pages 376–387. Springer, 2011.
16. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph partitioning with natural cuts. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1135–1146. IEEE, 2011.
17. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Exact combinatorial branch-and-bound for graph bisection. In *Proc. Algorithm Engineering and Experiments (ALENEX)*, pages 30–44, 2012.

18. D. Delling and R. F. Werneck. Better bounds for graph bisection. In *Proc. European Symposium on Algorithms (ESA)*, pages 407–418, 2012.
19. D. Delling and R. F. Werneck. Faster customization of road networks. In *Proc. International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *LNCS*, pages 30–42. Springer, 2013.
20. E. D. Demaine, M. Hajiaghayi, and K. Kawarabayashi. Contraction decomposition in h-minor-free graphs and algorithmic applications. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 441–450, 2011.
21. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.
22. A. E. Feldmann and P. Widmayer. An $O(n^4)$ time algorithm to compute the bisection width of solid grid graphs. In *Proc. European Symposium on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 143–154, 2011.
23. A. Felner. Finding optimal solutions to the graph partitioning problem with heuristic search. *Annals of Math. and Artificial Intelligence*, 45(3–4):293–322, 2005.
24. C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. The node capacitated graph partitioning problem: A computational study. *Mathematical Programming*, 81:229–256, 1998.
25. M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.
26. M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified $\mathcal{NP}$-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.
27. B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.
28. A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck. Maximum flows by incremental breadth-first search. In *Proc. European Symposium on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 457–468, 2011.
29. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
30. W. W. Hager, D. T. Phan, and H. Zhang. An exact algorithm for graph partitioning. *Mathematical Programming*, 137:531–556, 2013.
31. M. Hein and T. Bühler. An inverse power method for nonlinear eigenproblems with applications in 1-spectral clustering and sparse PCA. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pages 847–855, 2010.
32. C. Helmberg. A cutting plane algorithm for large scale semidefinite relaxations. In M. Grötschel, editor, *The Sharpest Cut*. SIAM, Philadephia, PA, 2004.
33. B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.
34. B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing*, page 28. ACM Press, 1995.
35. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In Demetrescu et al. [21], pages 41–72.
36. M. Holzer, F. Schulz, and D. Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.
37. K. Jansen, M. Karpinski, A. Lingas, and E. Seidel. Polynomial time approximation schemes for MAX-BISECTION on planar and geometric graphs. *SIAM Journal on Computing*, 35:110–119, 2005.
38. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; Part I, Graph partitioning. *Operations Research*, 37(6):865–892, 1989.
39. E. Johnson, A. Mehrotra, and G. Nemhauser. Min-cut clustering. *Mathematical Programming*, 62:133–152, 1993.
40. S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.
41. M. Jünger, A. Martin, G. Reinelt, and R. Weismantel. Quadratic 0/1 optimization and a decomposition approach for the placement of electronic circuits. *Mathematical Programming*, 63:257–279, 1994.
42. D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal the ACM*, 43(4):601–640, 1996.
43. S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *INFORMS Journal on Computing*, 12:177–191, 2000.
44. G. Karypis and G. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1999.
45. T. Koch, A. Martin, and S. Voß. SteinLib: An updated library on Steiner tree problems in graphs. Technical Report 00-37, Konrad-Zuse-Zentrum Berlin, 2000.

46. V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Tr. on Graphics*, 22:277–286, 2003.
47. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
48. K. J. Lang and S. Rao. A flow-based method for improving the expansion or conductance of graph cuts. In *Proc. Conf. Integer Programming and Combinatorial Optimization (IPCO)*, pages 325–337, 2004.
49. U. Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In Demetrescu et al. [21], pages 19–40.
50. R. J. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.
51. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *PODC*, page 6. ACM, 2009.
52. K. Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Information Processing Letters*, 27:125–128, 1988.
53. H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009.
54. F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, volume 1067 of *LNCS*, pages 493–498. Springer, 1996.
55. H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 255–263. ACM Press, 2008.
56. F. Rendl, G. Rinaldi, and A. Wiegele. Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations. *Mathematical Programming*, 121:307–335, 2010.
57. P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. on Graphics*, 27:144:1–144:9, 2008.
58. P. Sanders and C. Schulz. Distributed evolutionary graph partitioning. In *Proc. Algorithm Engineering and Experiments (ALENEX)*, pages 16–29. SIAM, 2012.
59. P. Sanders and C. Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proc. International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.
60. M. Sellmann, N. Sensen, and L. Timajev. Multicommodity flow approximation used for exact graph partitioning. In *Proc. European Symposium on Algorithms (ESA)*, volume 2832 of *LNCS*, pages 752–764, 2003.
61. N. Sensen. Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows. In *Proc. European Symposium on Algorithms (ESA)*, volume 2161 of *LNCS*, pages 391–403, 2001.
62. J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.
63. A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.
64. A. J. Soper, C. Walshaw, and M. Cross. The graph partitioning archive, 2004.
65. C. Walshaw and M. Cross. JOSTLE: Parallel multilevel graph-partitioning software – an overview. In F. Magoulès, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.
66. Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993.

A Algorithm Examples

A.1 Packing Bound

Figure 2 illustrates our lower bounds. Assume that $\epsilon = 0$ and that all vertices have unit weight. Note that $W = 24$ and $W_- = W_+ = 12$, i.e., we must find a solution with 12 vertices on each side. For simplicity, we refer to vertices by their rows and columns in the picture: vertex (1,1) is at the top left, and vertex (4,6) at the bottom right.

Assume some vertices have already been assigned to A (red boxes) and B (blue disks). First, we compute a flow f (indicated in bold) from A to B (Figure 2(a)). By removing these edges, we obtain the graph G_f. We then compute a tree packing on G_f, as shown in Figure 2(b). For each of the 11 edges (u, v) with $u \in A$ and $v \notin A$ we grow a tree (indicated by different colors and labeled $a, \ldots, k$) in G_f. Note that the deadweight (number of vertices unreachable from A) is 6, and that 15 free vertices are reachable from A. Finally, we allocate the weights of these 15 vertices to the trees.

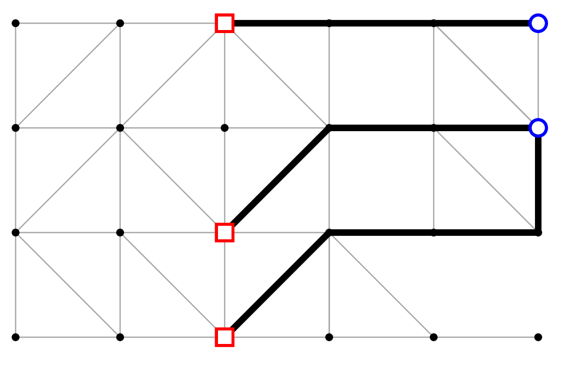
Figure 2(c) shows an allocation where each vertex is assigned in full to a single tree. This results in 8 trees of weight 1 (b, c, e, f, h, i, j, k), and 3 of weight 2 (a, d, g). Together, the three heaviest trees have weight 6; with 6 units of deadweight, these trees are enough to reach the target weight of 12. Therefore, the packing bound is 3. Together with the flow bound, this gives a total lower bound of 6.

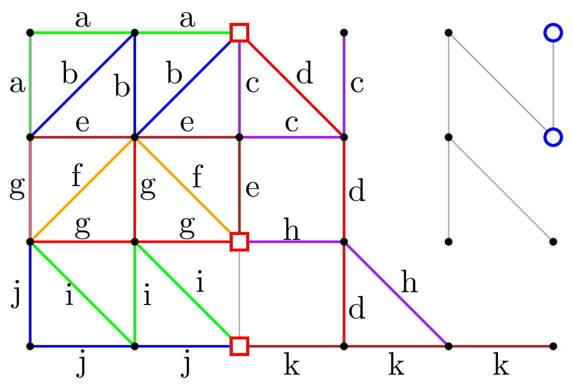
Figure 2(d) shows an alternative allocation in which some vertices are split equally among their incident trees. This results in 3 trees of weight 1 (e, f, j), and 8 trees of weight 1.5 (a, b, c, d, g, h, i, k). Now, we must add at least 4 trees to B ensure its weight is at least 12. The packing bound is thus 4 and the total lower bound is 7, matching the optimum solution.

A.2 Forced Assignment

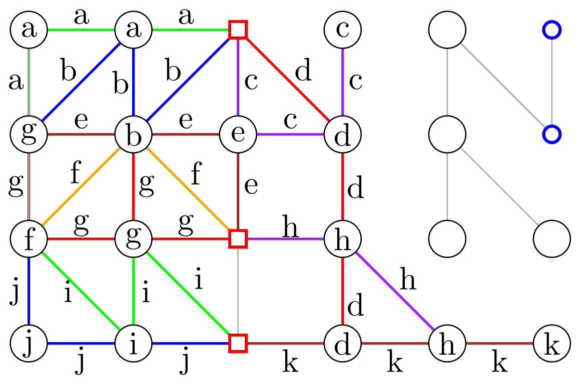
Figure 3 gives an example for our forced assignment techniques. We start from the tree packing in Figure 2(b). Figure 3(a) shows how the flow-based forced assignment applies to vertex $(2, 2)$. It is incident to four trees (b, e, f, g). If it were assigned to B (blue circles), the flow bound would increase by 4 units, to 7. Using the extended flow-based forced assignment, we can increase the flow bound by another 2 units, sending flow along $f + j$ and $b + a$ to A (red squares). If the total lower bound, including the recomputed packing bound, is at least as high as the best solution seen so far, we can safely assign vertex $(2, 2)$ to A.

Figure 3(b) illustrates our subdivision-based forced assignment. Consider what would happen if we were to assign vertex $(4, 2)$ to A. We implicitly split all trees incident to this vertex (i and j) into new trees i', i'', and j'. Tree i' is rooted at vertex $(4, 3)$, and the others at vertex $(4, 2)$. The vertex assignment remains consistent with the original one (as in Figure 2(c) or 2(d), for example). We then recompute the packing bound for this set of trees. If the new lower bound is at least as high as the best solution seen so far, we can safely assign vertex $(4, 2)$ to B.

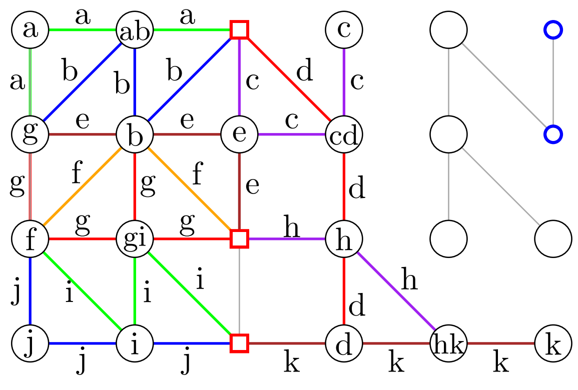

(a) Flow Bound



(b) Tree Packing

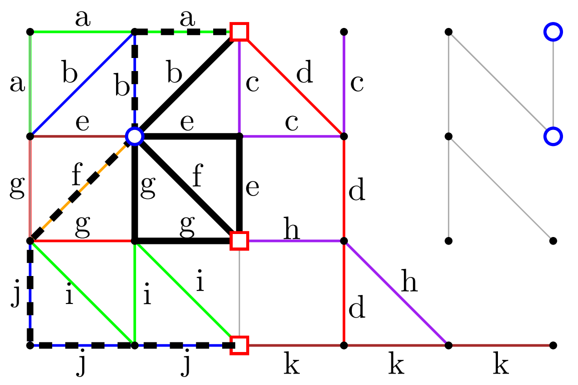


(c) Integral Vertex Allocation

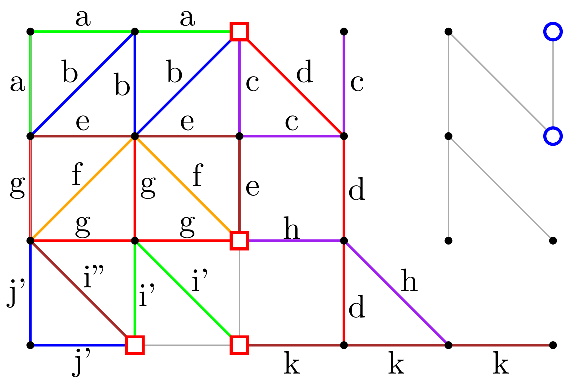


(d) Fractional Vertex Allocation

Fig. 2. Example for lower bounds. Red boxes and blue circles are already assigned to A and B, respectively. The figures show (a) the maximum A-B flow; (b) a set of maximal edge-disjoint trees rooted at A; (c) an integral vertex allocation; and (d) a fractional allocation where vertices with two labels have their weights equally split among the corresponding trees.



(a) Flow-based

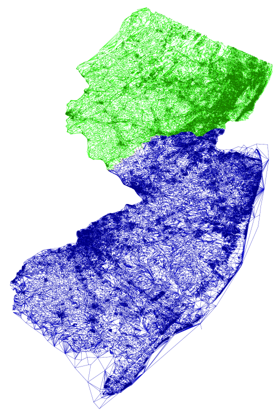


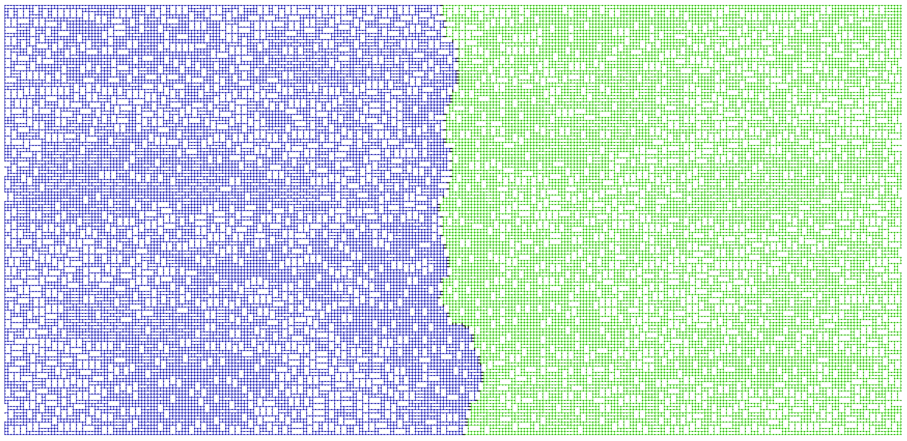
(b) Subdivision-based

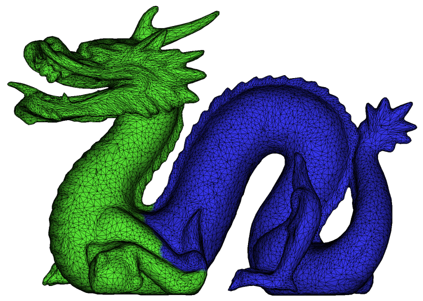
Fig. 3. Examples of forced assignments. Figure (a) shows the additional flow that would be created if vertex $(2, 2)$ were assigned to B (blue circles); solid edges correspond to the standard flow-based forced assignment and dashed edges to the extended version. Figure (b) shows (with primed labels) new trees that would be created if vertex $(4, 2)$ were assigned to A (red squares).

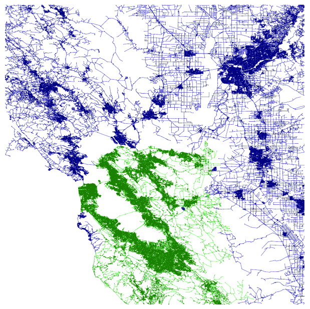
B Additional Solutions

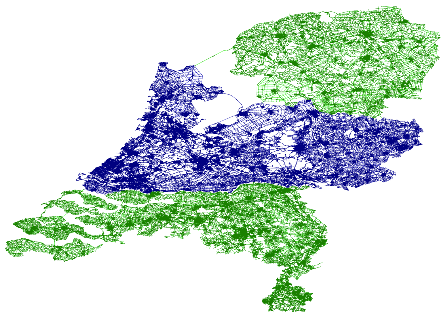
Figure 4 presents optimal bisections of large benchmark instances from various classes. Coordinates for the walshaw instances were obtained from the University of Florida Sparse Matrix Collection [14]. High-resolution images (and the actual solutions) are available upon request.

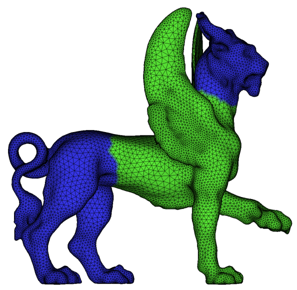

(a) nj2010 (redistrict), with $n = 169\,588$ and solution 150

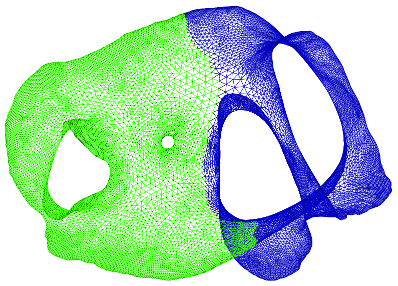

(b) alue7065 (vlsi), with $n = 34\,046$ and solution 80


(c) dragon-043571 (cgmesh), with $n = 21\,890$ and solution 148


(d) bay (road), with $n = 321\,270$ and solution 18


(e) netherlands (streets), with $n = 2\,216\,688$ and solution 45


(f) feline (cgmesh), with $n = 20\,629$ and solution 148


(g) 4elt (walshaw), with $n = 15\,606$ and solution 139

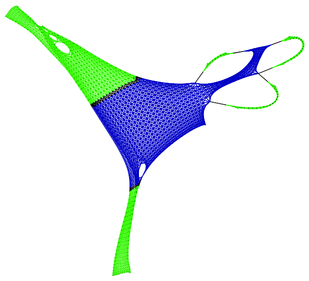

(h) data (walshaw), with $n = 2\,851$ and solution 189

Fig. 4. Optimal solutions for various benchmark instances (using $\epsilon = 0.03$ for nj2010 and $\epsilon = 0$ for the others).