Better Bounds for Graph Bisection

Daniel Delling and Renato F. Werneck

Microsoft Research Silicon Valley
{dadellin,renatow}@microsoft.com

Abstract. We introduce new lower bounds for the minimum graph bisection problem. Within a branch-and-bound framework, they enable the solution of a wide variety of instances with tens of thousands of vertices to optimality. Our algorithm compares favorably with the best previous approaches, solving long-standing open instances in minutes.

1 Introduction

We study the minimum graph bisection problem: partition the vertices of a graph into two cells of equal vertex weight so as to minimize the number of edges between them. This classical NP-hard problem [10] has applications in areas as diverse as VLSI design, load-balancing, and compiler optimization. Fast and good heuristics exist [14, 19], but provide no quality guarantee. Our focus is on *exact* algorithms, which normally rely on the branch-and-bound framework [16]. Traditional approaches apply sophisticated techniques to find lower bounds, such as multicommodity flows [20, 21], or linear [3, 5, 9], semidefinite [1, 3, 13, 17], and quadratic programming [11]. The bounds they obtain tend to be very good, but quite expensive to compute. As a result, they can handle relatively small graphs, typically with hundreds of vertices. (See Armbruster [1] for a survey.)

Delling et al. [8] have recently proposed a branch-and-bound algorithm using only combinatorial bounds. Their *packing bound* involves building collections of disjoint paths and arguing that any bisection must cut a significant fraction of those. This approach offers a different trade-off: branch-and-bound trees tend to be bigger (since the bounds are weaker), but each node can be processed much faster. This pays off for very sparse graphs with small bisections, and allowed them to solve (for the first time) instances with tens of thousands of vertices.

In this work, we propose a new combinatorial bound that follows the same principle, but is much stronger. Section 3 explains our new *edge-based packing bound* in detail: it finds a much larger collection of paths by allowing them to overlap in nontrivial ways. Section 4 shows how to actually build this collection efficiently, which requires sophisticated algorithms and significant engineering effort. We then show, in Section 5, how to fix some vertices to one of the cells without actually branching on them. After explaining additional details of the branch-and-bound algorithm (Section 6), we present extensive experimental results in Section 7. It shows that our new algorithm outperforms any previous technique on a wide range of inputs, and is almost never much worse. In fact, it can solve several benchmark instances that have been open for decades, often in minutes or even seconds.

2 Preliminaries

We take as input a graph $G = (V, E)$, with $|V| = n$ and $|E| = m$. Each vertex $v \in V$ has an integral weight $w(v)$. By extension, for any set $S \subseteq V$, let $w(S) = \sum_{v \in S} w(v)$. Let $W = w(V)$ denote the total weight of all vertices. 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 number of edges between cells (*cut size*) is minimized. Conversely, $W_- = W - W_+$ is the *minimum allowed cell size*. Our algorithms assume edges are unweighted; it deals with small integral edge weights by creating parallel unweighted edges.

To find the optimum solution, we use the *branch-and-bound* technique [16]. It implicitly enumerates all solutions by dividing the original problem into 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. At all times, we keep a global *upper bound U* on the solution of the original problem, which is updated as better solutions are found. To process a node in the tree, we 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.

Concretely, for graph bisection each node of the branch-and-bound tree represents a *partial assignment* (A, B), where $A, B \subseteq V$ and $A \cap B = \emptyset$. The vertices in A or B are said to be *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 has the form $(A, B) = (\{v_0\}, \emptyset)$ and represents all valid bisections. (Note that we fix an arbitrary vertex v_0 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). If $L(A, B) \geq U$, we prune. Otherwise, we choose a free vertex v and *branch* on it, generating subproblems $(A \cup \{v\}, B)$ and $(A, B \cup \{v\})$.

The number of nodes in the branch-and-bound tree depends crucially on the quality of the lower bound. As a starting point, we use the well-known [7] *flow bound*: the minimum s–t cut between A and B. It is a valid lower bound because any extension (A^+, B^+) must separate A from B. It also functions as a *primal* heuristic: if the minimum cut happens to be balanced, we can update U. Unfortunately, the flow bound can only work well when A and B have similar sizes, and even in such cases the corresponding cuts are often far from balanced, with one side containing almost all vertices. Because the flow bound does not use the fact that the final solution must be balanced, it is rather weak by itself.

3 Edge-Based Packing Bound

To take the balance constraint into account, we propose the *edge-based packing bound*, a novel lower bounding technique. 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. 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.

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). 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 shows how to build it). 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 a separate 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., one 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, regarding 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*, i.e., 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 $\mathcal{T}(v)$. (Section 4 will discuss how.) 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, 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 pick 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 this bound can be computed by a *greedy algorithm*, which picks trees in decreasing order of weight until their accumulated weight is at least W_f.

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. The weight of a tree T is defined in the natural way, as the sum of all fractional weights allocated to T. Fractional allocations can improve the packing bound by making trees more balanced. They 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 main 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. We must prove that $p(\mathcal{T}) \leq |\mathcal{T}^*|$. (Since we only consider G_f, the flow bound stays valid.) It suffices to show that $\mathcal{T}^*$ is feasible, i.e., that $w(\mathcal{T}^*) \geq W_f$ (since the packing bound minimizes over all feasible sets, it cannot be higher than any one of them). 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 $\mathcal{T}^*$; 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 $w(\mathcal{T}^*) \geq w(R^*) \geq w(B^*) - w(D_f) \geq W_- - w(D_f) = W_f$. $\square$

Comparison. Our packing bound is a generalization of the bound proposed by Delling et al. [8], which also creates a set of disjoint trees and uses a greedy packing algorithm to compute a lower bound. The crucial difference is that, while we only need the trees to be *edge-disjoint*, Delling et al. [8] also require them to be *vertex-disjoint*. We therefore refer to their method as VBB (for *vertex-based bound*). Dropping vertex-disjointness not only allows our method to balance the trees more effectively (since it can allocate vertex weights more flexibly), but also increases the number of available trees. This results in significantly better lower bounds, leading to much smaller branch-and-bound trees. As Section 7 will show, our method is particularly effective on instances with high-degree vertices, where it can be several orders of magnitude faster than VBB. The only drawback of our approach relative to VBB is that finding overlapping trees efficiently requires significantly more engineering effort, as the next section will show.

4 Bound Computation

Theorem 1 applies to any valid tree packing $\mathcal{T}$, but the quality of the bound it provides varies. Intuitively, we should pick $\mathcal{T}$ (and an associated weight allocation) so as to avoid heavy trees: this improves $p(\mathcal{T})$ by increasing the number of trees required to achieve the target weight W_f. Since $|\mathcal{T}|$ and $w(\mathcal{T})$ are both fixed (for any $\mathcal{T}$), in the ideal tree packing all trees would have the same weight. It is easy to show that finding the best such packing is NP-hard, so in practice we must resort to (fast) heuristics. Ideally, the trees and weight allocations should be computed simultaneously (to account for the interplay between them), but it is unclear how to do so efficiently. Instead, we use a two-stage approach: first compute a valid tree packing, then allocate vertex weights to these trees appropriately. We discuss each stage in turn.

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. We do so by growing these trees simultaneously, trying to balance their sizes.

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 and removed from the DFS stack. In our setting, however, the same vertex may be in several trees (and stacks) simultaneously. 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 inserted into at most one stack. This takes $O(m)$ total space regardless of the number of trees.

Given this representation, we now describe the basic step of SDFS in more detail. First, pick an active tree T_i of minimum size (using buckets). 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. 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 pop (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 $\mathit{parent}(v, w) \leftarrow (u, v)$. 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). Note that we can precompute the distances from A to all vertices with a single BFS.

To bound the running time of SDFS, note that a vertex v can be scanned $O(\deg(v))$ times (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.

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: *initial allocation* and *local search.* We discuss each in turn.

The first stage allocates each vertex to a single tree. We maintain, for each tree T_i, its maximum *potential weight* $\Pi(i)$, defined as the sum of the weights of all vertices that are adjacent to T_i and have not yet been allocated to another tree. To keep the trees balanced, we allocate weight to trees with smaller $\Pi(\cdot)$ values first. More precisely, initially all vertices in $\mathcal{T}$ are *available* (not allocated), all trees T_i are *active*, and $\Pi(i)$ is the sum of the weights of all available vertices incident to T_i. In each step, the algorithm picks an active tree T_i such that $\Pi(i)$ is minimum. If there is an available vertex v incident to T_i, we allocate it to T_i; otherwise, we mark the tree as inactive. We stop when no active tree remains.

To implement this, we maintain the active trees in a priority queue (according to $\Pi(i)$), and each tree T_i keeps a list of all available vertices it is incident to; these lists have combined size $O(m)$. When v is allocated to a tree T_i, we decrease $\Pi(j)$ for all trees $T_j \neq T_i$ that are incident to v ($\Pi(i)$ does not change), remove v from the associated lists, and update the priority queue. The total time is $O(m \log m)$ with a binary heap or $O(m+W)$ with buckets (with integral weights).

Given an initial allocation, we then run a *local search* to rebalance the trees. Unlike the constructive algorithm, it allows fractional allocations. 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 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.

5 Forced Assignments

Consider a partial assignment (A, B). As observed by Delling et al. [8], if the current lower bound for (A, B) is close enough to the upper bound U, one can often infer that certain free vertices v must be assigned to A (or B) with no need to branch, reducing the size of the branch-and-bound tree. This section studies how these *forced assignments* can be generalized to work with our stronger edge-based bounds. As usual, assume A is the main side, let $\mathcal{T}$ be a tree packing with weight allocations, and let $f + p(A)$ be the current lower bound.

First, we 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 a different 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.

Note that 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. Assigning such a vertex w to A would increase the overall bound by $|\mathcal{T}'_B(w)|$ (because the extra flow is on B's side, it does not affect $p(\mathcal{T})$). If the new bound $f + p(\mathcal{T}) + |\mathcal{T}'_B(w)|$ is U or higher, we can safely assign w to B.

Another strategy we use is *subdivision-based forced assignments*, which subdivides 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 of the newly-created trees. To do so efficiently, we use some precomputation. For each edge e, let $T(e) \in \mathcal{T}$ be the tree to which e belongs. Define $s(e)$ as the weight of the subtree of $T(e)$ rooted at e: this is the sum, over all vertices descending from e in $T(e)$, of the (fractional) weights allocated to $T(e)$. (If e belongs to no tree, $s(e)$ is undefined.) The $s(e)$ values can be computed with a bottom-up traversal of all trees, which takes $O(m)$ total time.

These precomputed values are useful when the forced assignment routine processes a vertex v. Each edge $e = (v, u)$ is either a *parent* or a *child* edge, depending on whether u is on the path from v to $T(e)$'s root or not. If e is a child edge, it will generate a tree of size $s(e)$. If e is a parent edge, the new tree will have size $s(r(e)) - s(e)$, where $r(e)$ is the root edge of $T(e)$.

Note that both forced-assignment techniques (flow-based and subdivision-based) must compute a new packing bound $p(\mathcal{T}')$ for each vertex v they process. Although they need only $O(\deg(v))$ time to transform $\mathcal{T}$ into $\mathcal{T}'$, actually computing the packing bound from scratch can be costly. Our implementation uses an *incremental algorithm* instead. When computing the original $p(\mathcal{T})$ bound, we remember the entire state of its computation (including the sorted list of all original tree weights). To compute $p(\mathcal{T}')$, we can start from this initial state, discarding trees that are no longer valid and considering new ones appropriately.

6 The Full Algorithm

We test our improved bounds by incorporating them into Delling et al.'s branch-and-bound routine [8]. We process each node of the branch-and-bound tree as follows. We first compute the flow bound, then add to it our new edge-based packing bound (which fully replaces their vertex-based bound). If the result is not smaller than the best known upper bound U, we prune. Otherwise, we try both types of forced assignment, then branch. The remainder of this section discusses branching rules, upper bounds, and an optional decomposition technique.

VBB branches on the free vertex v that maximizes a certain *score* based on three parameters: the degree of v, the *distance* from v to $A \cup B$, and the average *weight* of the trees $\mathcal{T}(v)$ that v belongs to. Together, these criteria aim to maximize the overall (flow and packing) bound. Besides these, we propose a fourth criterion: whenever the current minimum cut A–B is almost balanced, we prefer to branch on vertices that already carry some flow in order to increase the number of reachable vertices in G_f.

Like VBB, we only update the best *upper bound* U when the minimum A–B cut happens to be balanced. Moreover, we do not use any heuristics to try to find a good initial bound U. Instead, we just call the branch-and-bound algorithm repeatedly, with increasing values of U, and stop when the bound it proves is better than the input. We use $U_1 = 1$ for the first call, and set $U_i = \lceil 1.05 U_{i-1} \rceil$ for call $i > 1$. (Delling et al. suggest using 1.5 instead of 1.05, but this is too aggressive for nontrivial instances; we therefore run VBB with 1.05 in our experiments.)

Finally, we consider *decomposition.* As in VBB, the quality of our lower bounds depend on the degrees of the vertices already assigned to A or B (which limit both the A–B flow and the number of trees). If a graph with small degrees has a large bisection, the branch-and-bound tree can get quite deep. Delling et al. [8] propose a decomposition-based preprocessing technique to get around this. Let U be an upper bound on the optimum bisection of the input graph $G = (V, E)$. Partition E into $U + 1$ sets $E_0, E_1, \ldots, E_U$, and for each i create a new graph G_i by taking G and *contracting* all edges in E_i. To solve G, we simply solve each G_i to optimality (with our standard branch-and-bound algorithm) and return the best solution found. At least one subproblem must preserve the optimum solution, since none of the solution edges will be contracted. Delling et al. propose partitioning the edges into clumps (paths with many neighbors) to ensure that, after contraction, each graph G_i will have at least a few high-degree vertices. We generate clumps similarly (see [8] for details), adjusting a few parameters to better suit our stronger lower bounds: we allow clumps to be twice as long, and randomize the distribution of clumps among subproblems.

7 Experiments

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.

We test the effectiveness of our approach by comparing our branch-and-bound algorithm with other exact approaches proposed in the literature. We consider a set of benchmarks compiled by Armbruster [2] containing instances used in VLSI design (alue, alut, diw, dmxa, gap, taq), meshes (mesh), random graphs (G), random geometric graphs (U), and graphs deriving from sparse symmetric linear systems (KKT) and compiler design (cb). In each case, we use the

Table 1. Performance of our algorithm compared with the best available times obtained by Armbruster [1], Hager et al. [11], and Delling et al.'s VBB [8]. 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; "—" means "not tested" and *DNF* means "not finished in at least 5 hours".

NAME	n	m	ϵ	*opt*	BB	TIME	[Arm07]	[HPZ11]	VBB
G124.02	124	149	0.00	13	426	0.08	13.91	4.21	0.06
G124.04	124	318	0.00	63	204999	52.80	4387.67	953.63	768.20
G250.01	250	331	0.00	29	41754	10.58	1832.25	10106.14	16.34
KKT_capt09	2063	10936	0.05	6	33	0.16	1164.88	4658.59	0.10
KKT_skwz02	2117	14001	0.05	567	891	7.87	*DNF*	—	*DNF*
KKT_plnt01	2817	24999	0.05	46	12607	72.29	*DNF*	—	*DNF*
KKT_heat02	5150	19906	0.05	150	9089	120.13	*DNF*	—	614.04
U1000.05	1000	2394	0.00	1	456	0.33	53.62	—	0.27
U1000.10	1000	4696	0.00	39	2961	8.83	1660.63	—	17.38
U1000.20	1000	9339	0.00	222	38074	276.05	*DNF*	—	17469.18
U500.05	500	1282	0.00	2	138	0.12	19.81	—	0.30
U500.10	500	2355	0.00	26	967	1.32	495.91	—	2.42
U500.20	500	4549	0.00	178	66857	225.02	*DNF*	—	*DNF*
alut2292.6329	2292	13532	0.05	154	24018	251.05	391.76	—	3058.29
alue6112.16896	6112	36476	0.05	272	378320	13859.72	4774.15	—	*DNF*
cb.47.99	47	3906	0.00	765	270	1.24	5.28	0.29	*DNF*
cb.61.187	61	33281	0.00	2826	793	91.81	81.35	0.80	*DNF*
diw681.1494	681	3081	0.05	142	5362	12.38	*DNF*	—	*DNF*
diw681.3103	681	18705	0.05	1011	6673	162.85	*DNF*	—	*DNF*
diw681.6402	681	7717	0.05	330	2047	9.22	4579.12	—	*DNF*
dmxa1755.10867	1755	13502	0.05	150	2387	28.59	*DNF*	—	57.10
dmxa1755.3686	1755	7501	0.05	94	10390	63.42	1972.22	—	371.99
gap2669.24859	2669	29037	0.05	55	74	1.31	348.95	—	0.52
gap2669.6182	2669	12280	0.05	74	3225	31.26	651.03	—	105.87
mesh.138.232	138	232	0.00	8	124	0.05	10.22	6.91	0.04
mesh.274.469	274	469	0.00	7	79	0.05	8.52	24.62	0.38
taq170.424	170	4317	0.05	55	193	0.53	28.68	—	8.14
taq334.3763	334	8099	0.05	341	1379	4.88	*DNF*	—	*DNF*
taq1021.2253	1021	4510	0.05	118	3373	11.93	169.65	—	283.04

same value of ϵ tested by Armbruster, which is either 0 or 0.05 (but note that his definition of ϵ differs slightly).

Table 1 reports the results obtained by our algorithm (using decomposition for U, alue, alut, and KKT_heat02). For comparison, we also show the running times obtained by recent state-of-the-art algorithms by Armbruster et al. [3, 1] (using linear or semidefinite programming, depending on the instance), Hager et al. [11] (quadratic programming), and Delling et al.'s VBB approach. Our method and VBB were run on a 3.16 GHz Core 2 Duo, while Armbruster used a 3.2 GHz Pentium 4 540 and Hager et al. used a 2.66 GHz Xeon X5355. Since these machines are not identical, small differences in running times (a factor of two or so) should be disregarded.

The table includes all instances that can be solved by at least one method in less than 5 hours (the time limit set by Armbruster [1]), except those that can be solved in less than 10 seconds by both our method and Armbruster's. Note that we can solve every instance in the table to optimality. Although slightly slower than Armbruster's in a few cases (notably alue6112.16896), our method 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 5 hours.

Our approach is significantly faster than Hager et al.'s for mesh, KKT, and random graphs (G), but somewhat slower for the cb instances, which are small but have heavy edges. Since we convert them to parallel (unweighted) edges, we end up dealing with much denser graphs (as the m column indicates). On such dense instances, fractional allocations help the most: without them, we would need almost 1000 times as many branch-and-bound nodes to solve cb.47.99.

Compared to VBB, our method is not much better for graphs that are very sparse or have small bisections—it can be even slower, since it often takes 50% more time per branch-and-bound node due to its costlier packing computation. As in VBB, however, flow computations still dominate. For denser instances, however, our edge-based approach is vastly superior, easily handling several instances that are beyond the reach of VBB.

With longer runs, both Armbruster and Hager et al. [11] can solve random graphs G124.08 and G124.16 (not shown in the table) in a day or less. We would take about 3 days on G124.08, and a month or more for G124.16. Here the ratio between the solution value (449) and the average degree (22) is quite large, so we can only start pruning very deep in the tree. Decomposition would contract only about three edges per subproblem, which does not help. This shows that there are classes of instances in which our method is clearly outperformed.

For real-world instances, however, we can actually solve much larger instances than those shown on Table 1. In particular, we consider instances from the 10th DIMACS Implementation Challenge [4] representing social and communication networks (class clustering), road networks (streets), Delaunay triangulations (delaunay), random geometric graphs (rgg), and assorted graphs (walshaw) from the Walshaw benchmark [22] (mostly finite-element meshes). We also consider triangulations representing three-dimensional objects in computer graphics (mesh) [18] and grid graphs with holes (vlsi) representing VLSI circuits [15].

Table 2 compares our algorithm with VBB (results are not available for other exact methods). Both use decomposition for all classes but clustering. Since our focus is on lower bound quality, here we ran both algorithms directly with $U = opt + 1$ as an initial upper bound. For each instance, we report the number of branch-and-bound nodes and the running time of our algorithm, as well as its speedup (SPD) relative to VBB, i.e., the ratio between VBB's running time and ours. Note that VBB runs that would take more than a day were actually executed on a cluster using DryadOpt [6], which roughly doubles the total CPU time. The table includes most nontrivial instances tested by Delling et al. [8], and additional instances that could not be solved before. On instances marked *DNF*, VBB would be at least 200 times slower than our method.

Table 2. Performance on various large instances with $\epsilon = 0$; BB is the number of branch-and-bound nodes, TIME is the total CPU time, and SPD is the speedup over Delling et al.'s VBB algorithm [8]; *DNF* means the speedup would be at least 200.

CLASS	NAME	n	m	opt	BB	TIME [S]	SPD
clustering	lesmis	77	820	61	21	0.02	12975.4
	as-22july06	22963	48436	3515	7677	417.27	*DNF*
delaunay	delaunay_n11	2048	6127	86	4540	18.87	9.3
	delaunay_n12	4096	12264	118	13972	140.64	19.3
	delaunay_n13	8192	24547	156	34549	759.67	49.5
	delaunay_n14	16384	49122	225	635308	30986.82	*DNF*
mesh	cow	2903	8706	79	2652	13.04	5.1
	fandisk	5051	14976	137	6812	81.81	63.4
	blob	8036	24102	205	623992	12475.18	*DNF*
	gargoyle	10002	30000	175	46623	1413.62	33.0
	feline	20629	61893	148	43944	1474.22	3.1
	dragon-043571	21890	65658	148	1223289	53352.66	109.7
	horse	48485	145449	355	121720	21527.24	*DNF*
rgg	rgg15	32768	160240	181	5863	1111.35	148.1
	rgg16	65536	342127	314	43966	24661.47	32.2
streets	luxembourg	114599	119666	17	844	101.21	0.9
vlsi	alue7065	34046	54841	80	9650	350.05	1.4
walshaw	data	2851	15093	189	29095	265.12	21689.9
	crack	10240	30380	184	19645	605.12	479.2
	fe_4elt2	11143	32818	130	1324	42.89	5.2
	4elt	15606	45878	139	4121	187.51	4.1
	fe_pwt	36519	144794	340	2310	394.50	39.7
	fe_body	45087	163734	262	147424	17495.68	*DNF*
	finan512	74752	261120	162	1108	339.70	2.2

Our new algorithm is almost always faster than VBB, which is only competitive for very sparse inputs, such as road networks. For denser graphs, our algorithm can be orders of magnitude faster, and can solve a much greater range of instances. Note that several instances could not be solved with VBB even after days of computation. In contrast, in a few hours (or minutes) we can solve to optimality a wide variety of graphs with tens of thousands of vertices.

8 Conclusion

We have introduced new lower bounds 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 [12]). 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 causes us to branch more, but we usually make up for it with a faster lower bound computation.

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 *IPCO*, LNCS 5035, pp. 112–124, 2008.
4. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 10th DIMACS Implementation Challenge: Graph Partitioning and Graph Clustering, 2011. `http://www.cc.gatech.edu/dimacs10/index.shtml`.
5. L. Brunetta, M. Conforti, and G. Rinaldi. A branch-and-cut algorithm for the equicut problem. *Mathematical Programming*, 78:243–263, 1997.
6. M. Budiu, D. Delling, and R. F. Werneck. DryadOpt: Branch-and-bound on distributed data-parallel execution engines. In *IPDPS*, pp. 1278–1289, 2011.
7. T. N. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.
8. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Exact combinatorial branch-and-bound for graph bisection. In *ALENEX*, pp. 30–44, 2012.
9. 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.
10. 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.
11. W. W. Hager, D. T. Phan, and H. Zhang. An exact algorithm for graph partitioning. *Mathematical Programming*, pp. 1–26, 2011.
12. 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.
13. S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *INFORMS Journal on Computing*, 12:177–191, 2000.
14. G. Karypis and G. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1999.
15. 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.
16. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
17. F. Rendl, G. Rinaldi, and A. Wiegele. Solving max-cut to optimality by intersecting semidefinite and polyhedral relaxations. *Math. Programming*, 121:307–335, 2010.
18. 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.
19. P. Sanders and C. Schulz. Distributed evolutionary graph partitioning. In *ALENEX*, pp. 16–29. SIAM, 2012.
20. M. Sellmann, N. Sensen, and L. Timajev. Multicommodity flow approximation used for exact graph partitioning. In *ESA*, LNCS 2832, pp. 752–764, 2003.
21. N. Sensen. Lower bounds and exact algorithms for the graph partitioning problem using multicommodity flows. In *ESA*, LNCS 2161, pp. 391–403, 2001.
22. A. J. Soper, C. Walshaw, and M. Cross. The Graph Partitioning Archive, 2004. `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.