Exact Combinatorial Branch-and-Bound for Graph Bisection*

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

[1] Microsoft Research Silicon Valley
{dadellin,goldberg,renatow}@microsoft.com
[2] Lomonosov Moscow State University
ilyaraz@gmail.com

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 present stronger lower bounds, improved branching rules, and a new decomposition technique that contracts entire regions of the graph without losing optimality guarantees. In practice, our algorithm works particularly well on instances with relatively small minimum bisections, solving large real-world graphs (with tens of thousands to millions of vertices) to optimality.

1 Introduction

We consider the *minimum graph bisection* problem. Its input is an undirected, unweighted graph $G = (V, E)$, and its goal is to partition V into two sets A and B such that $|A|, |B| \leq \lceil |V|/2 \rceil$ and the number of edges between A and B (the *cut size*) is minimized. This fundamental combinatorial optimization problem is a special case of *graph partitioning*, which asks for arbitrarily many cells. It has numerous applications, including image processing [43, 47], computer vision [32], divide-and-conquer algorithms [34], VLSI circuit layout [5], distributed computing [35], and route planning [13]. Unfortunately, the bisection problem is NP-hard [20] for general graphs, with a best known approximation ratio of $O(\log n)$ [38]. Only some restricted graph classes, such as grids without holes [17] and graphs with bounded treewidth [27], have known polynomial-time solutions.

In practice, there are numerous general-purpose heuristics for graph partitioning, such as METIS [30], SCOTCH [11, 37], Jostle [46], and KaFFPaE [40]. Successful heuristics tailored to particular graph classes, such as DibaP [36] (for meshes) and PUNCH [14] (for road networks), are also available. These algorithms are quite fast (often running in near-linear time) and can handle very

* This is a condensed version of a paper that appeared at ALENEX 2012 [15].
** This work was done while the third author was at Microsoft Research Silicon Valley.

large graphs, with tens of millions of vertices. They cannot, however, prove optimality or provide approximation guarantees. Moreover, 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 [33]. Most algorithms use sophisticated machinery to obtain lower bounds, such as multicommodity flows [41, 42] or linear [2, 6, 19], semidefinite [1, 2, 29], and quadratic programming [24]. Computing such bounds is quite expensive, however, in terms of time and space. 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 [18] can offer a different tradeoff: they provide weaker lower bounds, but compute them much faster (often in sublinear time). This works well for random graphs with up to 100 vertices, but does not scale to larger instances.

This paper introduces a new exact algorithm for graph bisection. We use novel combinatorial lower bounds that can be computed in near-linear time in practice. Even so, these bounds are quite strong, and can be used to find optimum solutions to real-world graphs with remarkably many vertices (more than a million for road networks, or tens of thousands for VLSI and mesh instances). To the best of our knowledge, our method is the first to find exact solutions for instances of such scale. In fact, it turns out that the running time of our algorithm depends more on the size of the bisection than on the size of the graph.

Our paper has four main contributions. First, we introduce (in Section 3) new and improved combinatorial lower bounds that significantly strengthen previous bounds. Second, we propose (in Section 4) careful branching rules that help to exploit the full potential of our bound. Third, Section 5 introduces a new decomposition technique that boosts performance substantially: it finds the optimum solution by solving a small number of (much easier) subproblems independently. Finally, Section 6 presents a careful experimental analysis of our techniques.

2 Preliminaries

Let $G = (V, E)$ denote the input graph, 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 integral cost $c(e)$. Let $W = \sum_{v \in V} w(v)$. 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. A bisection is ϵ-balanced if each cell has weight at most $(1 + \epsilon)\lceil W/2 \rceil$. If $\epsilon = 0$, we say the partition is *perfectly balanced* (or just *balanced*). The *minimum graph bisection problem* is that of finding the minimum-cost balanced bisection.

To simplify exposition, unless otherwise noted we consider the unweighted, balanced version of the problem, where $w(v) = 1$ for all $v \in V$, $c(e) = 1$ for all

$e \in E$, and $\epsilon = 0$. We must therefore partition G into two cells, each with weight at most $\lceil n/2 \rceil$, while minimizing the number of edges between cells.

A standard technique for finding exact solutions to NP-hard problems is *branch-and-bound* [21, 33]. It performs an implicit enumeration 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, which can be updated as the algorithm finds improved solutions. 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.

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 the root can fix an arbitrary node 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, 2, 6, 19, 24, 29] usually apply mathematical programming techniques to find lower bounds. In this paper, we use only combinatorial bounds. In particular, our basic algorithm uses the well-known [8, 14] *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. 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 $(A, B \cup \{v\})$.

The flow lower bound can only work well when A and B have similar sizes; even in this case, the corresponding cuts are often far from balanced, with one side containing almost all vertices. This makes the flow bound rather weak by itself. To overcome these issues, we introduce a new *packing lower bound*.

3 The Packing Lower Bound

Let (A, B) be a partial assignment. To make it a balanced bisection, at least $\lfloor n/2 \rfloor - |A|$ free vertices must be assigned to A, obtaining an extended set A^+. (A similar argument can be made for B.) Suppose that, for each possible extension A^+ of A, we could compute the maximum flow $f(A^+)$ between B and A^+. Let f^* be the minimum such flow value (over all possible A^+); f^* is clearly a lower bound on the value of any bisection consistent with (A, B). Finding f^* exactly seems expensive; instead, we propose a fast algorithm to lower bound f^*.

It works as follows (see Fig. 1). Let $G' = G \setminus (A \cup B)$ be the subgraph of G induced by the vertices that are currently unassigned, and let R be the set of vertices of G' with at least one neighbor in B (in G). We partition the vertices in G' into connected cells, each containing at most one element of R. (Any

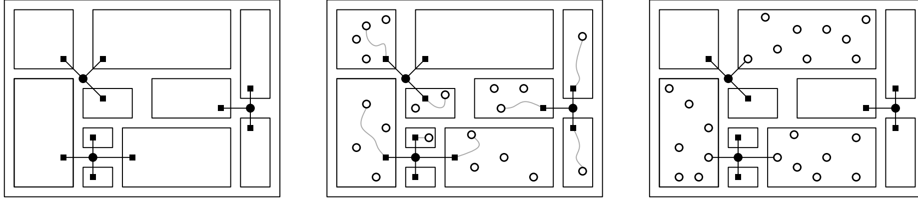

Fig. 1. The packing bound. Filled circles are vertices in B; their free neighbors (squares) form a set R. **Left**: We partition the free vertices into connected cells, each with at most one vertex of R. **Middle**: Given an extension A^+ (hollow circles), the number of nontrivial cells it touches (8) is a lower bound on the minimum (B, A^+) cut. **Right**: The extension that hits the fewest cells (3) is a lower bound on any valid extension.

such partition is valid; as we shall see, we get better lower bounds if the cells containing elements in R are large and have similar sizes.) We say that a cell C is *nontrivial* if it contains exactly one element from R; we call this element the *root* of the cell and denote it by $r(C)$. Cells with no element from R are *trivial*.

Lemma 1. *Let A^+ be a valid extension of A, and let $c(A^+)$ be the number of nontrivial cells hit by A^+. Then $c(A^+)$ is a lower bound on the maximum flow $f(B, A^+)$ from B to A^+.*

Proof. We claim we can find $c(A^+)$ disjoint paths between A^+ and B, each in a different nontrivial cell. Take a nontrivial cell C containing an element v from A^+. Because the cell is connected, there is a path P within C between v and its root $r(C)$. Because $r(C)$ belongs to R, there is an edge e (in the original graph G) between $r(C)$ and a vertex w in B. The concatenation of P and e is a path from A^+ to B. Since any valid extension must contain at least one edge from each of the $c(A^+)$ disjoint paths, the lemma follows.

Recall that we need a lower bound on *any possible* extension A^+ of A. We get one by finding the extension for which Lemma 1 gives the lowest possible bound (for a fixed partition into connected cells). To build this extension, we use a *greedy packing algorithm*. First, pick all vertices in trivial cells; because we cannot associate these cells with paths, they do not increase the lower bound. From this point on, we must pick vertices from nontrivial cells. Since the lower bound increases by one regardless of the number of vertices picked in a cell, we should pick entire cells at once (after one vertex is picked, others in the cell are free—they do not increase the bound). The optimal strategy is to pick cells in decreasing order of size, stopping when the sum of the sizes of all picked cells (trivial and nontrivial) is at least $\lfloor n/2 \rfloor - |A|$. We have thus shown the following:

Theorem 1. *The greedy packing algorithm finds a lower bound on the value of any bisection consistent with (A, B).*

Computing Packing Lower Bounds. The packing lower bound is valid for any partition, but its quality depends strongly on which one we pick. We should

choose the partition that forces the worst-case extension A^+ to hit as many nontrivial cells as possible. This means minimizing the total size of the trivial cells, and ensuring all nontrivial cells have the same number of vertices. This problem is hard [12, 10], but we propose two heuristics that work well in practice.

The first is a constructive algorithm that builds a reasonable initial partition from scratch. Starting from $|R|$ unit cells (each with one element of R), in each step it adds a vertex to a cell whose current size is minimum. This algorithm can be implemented in linear time by keeping with each cell C a list $E^+(C)$ of potential *expansion edges*, i.e., edges (v, w) such that $v \in C$ and $w \notin C$. Vertices that are not reachable from R are assigned to trivial cells. As the algorithm progresses, some cells will run out of expansion edges, as all neighboring vertices will already be taken. This may lead to very unbalanced solutions.

To improve the partition, we use our second heuristic: a *local search* routine that makes neighboring cells more balanced by moving vertices between them. To do so efficiently, it maintains a spanning tree for each nontrivial cell C, rooted at $r(C)$. Initially, this is the tree built by the constructive algorithm.

The local search moves entire subtrees between neighboring cells. It processes one *boundary edge* at a time. Consider one such edge (v, w), with $v \in C_1$ and $w \in C_2$, and assume cell C_1 has more vertices than C_2. To improve the solution, we attempt to move an entire subtree from C_1 to C_2. We find the best subtree to switch by traversing the path (in the spanning tree of C_1) from v to $r(C_1)$. (See Fig. 2.) Each vertex u on the path is associated with a possible move: removing the subtree rooted at u from C_1 and inserting it into C_2. Among these, let u^* be the vertex leading to the most balanced final state (in which the sizes of C_1 and C_2 are closest). If this is more balanced than the current state, we switch.

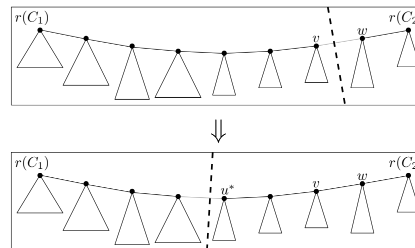

Fig. 2. Packing local search. **Top**: The boundary edge (v, w) determines a path between cells C_1 and C_2 (triangles are subtrees). **Bottom:** A different split of the path induces a more even partition.

The local search runs until a local optimum, when no improving switch exists. To implement it efficiently, we keep track of boundary edges and subtree sizes explicitly. This ensures the algorithm runs in polynomial (but superlinear) worst-case time. In practice, however, we reach a local optimum after very few moves, and the local search is about as fast as the constructive algorithm.

Combining Packing and Flows. We cannot simply add the packing- and flow-based bounds to obtain a unified lower bound, since they may interfere with one another. It is easy to see why: each method finds (implicitly) a set of edge-disjoint paths such that at least one edge from each such path must be in the solution. (For the flow bound, these are the paths in the flow decomposition.) Adding the bounds would require the sets of paths found by each algorithm to be mutually disjoint, which is usually not the case. To combine the bounds properly, we first compute the flow bound, then a packing bound that takes the flow into account.

More precisely, we first compute a flow bound f as usual. We then remove all flow edges from G, obtaining a new graph G_f. Finally, we compute the packing lower bound p on G_f. Now $f + p$ is a valid lower bound on the cost of the best bisection extending the current assignment (A, B), since there is no overlap between the paths considered by each method (flow and packing).

This approach finds valid lower bounds regardless of which edges are in the flow, but its packing portion is better if it has more edges to work with. We therefore favor flows with few edges: instead the standard push-relabel approach [23], we use an augmenting-path algorithm that greedily sends flows along shortest paths. We implemented a simplified version of IBFS (*incremental breadth first search*) [22], which is about as fast as push-relabel on our test instances.

Forced Assignments. Assume we have already computed the flow bound f followed by an additional packing lower bound p (using the neighbors of B as roots). For a free vertex v, let $N(v)$ be its set of neighbors in G_f (the graph without flow edges), let $\deg_{G_f}(v) = |N(v)|$, and let C be the cell (in the packing partition) containing v. We can often use logical implications to assign v to one of the sides (A or B) without actually branching on it. The idea is simple: if we can 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.

First, consider what would happen if v were added to A. Let $x(v)$, the *expansion* of v, be the number of nontrivial cells (from the packing bound) that contain vertices from $N(v)$. Note that $0 \leq x(v) \leq \deg_{G_f}(v)$. Assigning v to A would create $x(v)$ disjoint paths from A to B, effectively increasing the flow bound to $f' = f + x(v)$. Note, however, that $f' + p$ may not be a valid lower bound, since the new paths may interfere with the "pure" packing bound. Instead, we compute a *restricted* packing lower bound p', taking as trivial the cells that intersect $N(v)$ (we just assume they belong to A^+). If $f' + p'$ is at least as high as the current upper bound, we have proven that v must be assigned to B. This test tends to succeeds only when the cells are unevenly balanced (otherwise the increase in flow is offset by a decrease in the packing bound).

Conversely, consider what would happen if v were added to B: we could split C into $\deg_{G_f}(v)$ cells, one rooted at each neighbor of v. The size of each new cell can be computed in constant time, since we know the subtree sizes within the original spanning tree of C. We then recompute the packing lower bound (using the original cells, with C replaced by the newly-created subcells) and add it to the original flow bound f. If this at least matches the current upper bound, then we have proven that v must actually be assigned to A. This works particularly well for trivial cells (the packing bound is unlikely to change for nontrivial ones).

Note that these *forced assignments* only work when lower and upper bounds are very close. Their main benefit is to eliminate vertices that are not good candidates for branching. Since the tests are very fast, they are still worth running.

Extensions. We can easily generalize the packing bound to handle ϵ-*balanced partitions*. In this case, cells must have size at most $M^+ = \lfloor (1+\epsilon)\lceil n/2 \rceil \rfloor$ and at least $M^- = n - M^+$; the packing bound must distribute M^- vertices instead of $\lfloor n/2 \rfloor$. Dealing with *weighted vertices* is also quite simple. The packing bound is

the minimum number of cells containing at least half of the total weight. When creating the packing partition, we should therefore strive to make cells balanced by weight instead of number of vertices; this can easily be incorporated into the local search. To handle small integral *edge weights*, we can simply use parallel edges. Additional extensions (such as arbitrary edge weights or partitions into more than two cells) are possible, but more complicated.

4 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 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.

First, the flow and packing bounds would both benefit from having the assigned vertices evenly distributed (on both sides of the optimum bisection). Since we do not know what the bisection is, a reasonable strategy is to spread vertices over the graph by branching on vertices that are far from both A and B. (Note that a single BFS can find the distances from $A \cup B$ to all vertices.) We call this the *distance* criterion. Moreover, we prefer to branch on vertices that appear in large cells (from the packing bound). By allowing such cells to be split, we can improve the packing bound. Finally, to help our flow bound, we would like to send a large amount of flow from a branching vertex v to A or B. This suggests branching on vertices that are *well-connected* to the rest of the graph. A proxy for connectivity is the *degree* of v, a trivial upper bound on any flow out of v.

In practice, connectivity tends to be more important than the other criteria, so we branch on the vertex v that maximizes $q(v) = \text{dist}(v) \cdot \text{csize}(v) \cdot \text{conn}(v)^2$, where $\text{dist}(v)$ indicates the distance from v to the closest assigned vertex, $\text{csize}(v)$ is the size of the cell containing v, and $\text{conn}(v)$ is the connectivity (degree) of v.

For some graphs (such as road networks), degrees are poor proxies for connectivity, since high-degree vertices are often separated by a small cut from most of the graph [14]. We could obtain a more robust measure of connectivity by reusing the packing algorithm described in Section 3. For each vertex v, we can run the algorithm with $A = \emptyset$ and $B = \{v\}$ to find a partition of $V \setminus \{v\}$ into $\deg(v)$ cells. If v is well-connected, all cells should have roughly the same size; if not, some cells will be much smaller than others. Computing this bound for every vertex in the graph would be quite expensive, so we actually only sample (in a preprocessing step) a few high-degree vertices to better estimate their connectivity. This *filtering routine* helps eliminate obviously bad branching vertices.

5 Contraction

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 $D_A \leq D_B$ (without loss of generality). It is easy to see that the flow bound cannot be larger than D_A, and that the packing bound is at most $D_B/2$ (when the regions are perfectly balanced). If all vertices have small constant degree (as in 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 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 (its number of outside neighbors) is substantially higher, lower bounds should increase much faster as we traverse the branch-and-bound tree. The obvious problem with this idea is that the optimal bisection may actually split the region in two. 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, 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 entire 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 [28], 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. [8] 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 the 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, but for many inputs it has much better performance than our standard branch-and-bound algorithm.

The algorithm is as follows. Let U be an upper bound on the optimum bisection. First, partition E into $U+1$ disjoint sets $(E_0, E_1, \dots, E_U)$. 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. Then, use our standard algorithm to find the optimum bisection U_i of each graph G_i independently, and pick the best.

Theorem 2. *The decomposition algorithm finds the minimum bisection of G.*

Proof. Let $U^* \leq U$ be the minimum bisection cost. We must prove that $\min(U_i) = U^*$. 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 will 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, pick 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 does not change the optimum bisection, proving our claim.

The decomposition algorithm solves $U + 1$ subproblems, but the high-degree vertices introduced by contraction should make each subproblem much easier for our branch-and-bound. 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 a single branch-and-bound tree with a special root node that has $U + 1$ children, each responsible for a distinct contraction pattern. The subproblems are not necessarily disjoint (different branches may visit the same partial assignment), but this does not affect correctness.

Decomposition is correct regardless of how edges are partitioned among subproblems, but performance may vary significantly. To make all subproblems have comparable degree of difficulty, we 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 create vertices with much higher degree, which we achieve by assigning to E_i edges that induce relatively large connected components (or *clumps*) in G. (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 the remaining vertices.) Moreover, the *shape* of each clump matters: all else being equal, we would like its expansion (number of neighbors) to be as large as possible; we therefore make sure our clumps are *paths* in the graph. The maximum path length s is set to $\left\lceil \min\left\{4U, \frac{m}{10U}\right\}\right\rceil$ to balance two properties: clumps should not be much bigger than the optimum bisection, and each subproblem should have multiple clumps. We perform the decomposition in two stages: the *clump generation* partitions all the edges in the graph into clumps, while the *allocation* stage ensures that each subproblem is assigned a well-spread subset of the clumps of comparable total size. See the full paper [15] for more details.

6 Experiments

We implemented our algorithms in C++ using Visual Studio 2010. We ran most experiments on one core of an Intel Core 2 Duo E8500 running Windows 7 Enterprise at 3.16 GHz with 4 GB of RAM. For a few harder instances (clearly marked), we ran a distributed version of the code using the DryadOpt framework [7], which is written in C# and calls our native C++ code to solve individual nodes of the branch-and-bound tree. Distributed executions use up to 128 machines with two 2.6 GHz dual-core AMD Opteron processors (and 16 GB of RAM) each; we report the total CPU time across all machines. Unless otherwise mentioned, we find balanced partitions ($\epsilon = 0$).

Parameter Evaluation. We start by considering the effects of each improvement we propose on performance. For concreteness, we focus on two instances: alue5067 is a VLSI instance (a grid graph with holes used as a benchmark instance for the Steiner problem in graphs [31]) with 3524 vertices, 5560 edges, and optimum bisection $opt = 30$; mannequin is a mesh (triangulation) used in computer graphics studies [39] with 689 vertices, 2043 edges, and $opt = 61$.

Figure 3 shows the running times of several versions of our algorithm as the input bound U varies from 10 to $opt + 1$. (When $U \leq opt$, our algorithm simply proves that U is a valid lower bound.) Each version builds on the previous one. Version A, the most basic, uses the flow bound, the packing bound (using only the constructive algorithm to find cells), and branches on random vertices. Version B improves the packing partition using local search. Version C adds forced assignments. Versions D and E improve the branching criteria (from random): D uses distances, cell sizes, and degrees, while E also uses *filtering* to identify well-connected branching vertices. Versions F and G decompose the problem into $U + 1$ subproblems; F partitions the edges at random, while G uses clumps.

For alue5067, each version of the algorithm is faster than the previous one. The effect is minor for some features, such as forced assignments and random decomposition (since the subproblems it generates are not much easier). Other improvements (notably local search, sophisticated branching, and decomposition by clumps) clearly improve the asymptotic performance of the algorithm. Finally, we note that the packing bound itself leads to huge speedups: using only the flow bound, our algorithm would take more than 5 minutes for any $U \geq 3$.

The results for mannequin are similar, although decomposition is not as helpful (it even hurts if edges are distributed at random), since mannequin has higher degrees and much fewer fixed edges per subproblem (33) than alue5067 (179).

For both instances, Version G spends half the time to process each node on the flow computation, with the other half split roughly evenly among the remaining routines: constructive, local search, forced assignments, and branching. This indicates that processing a branch-and-bound node takes essentially linear time. Recall that filtering is done in a preprocessing stage (separately for each subproblem). For alue5067 (with $U = opt + 1$), it is almost as expensive as traversing the actual branch-and-bound tree; for mannequin, it takes roughly 15% of the total time (but does not help as much).

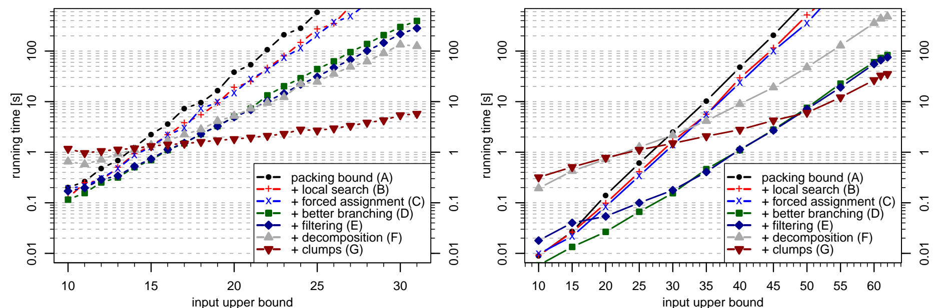

Fig. 3. Running times of increasingly sophisticated versions of our algorithm as a function of the upper bound U on the inputs alue5067 (left) and mannequin (right).

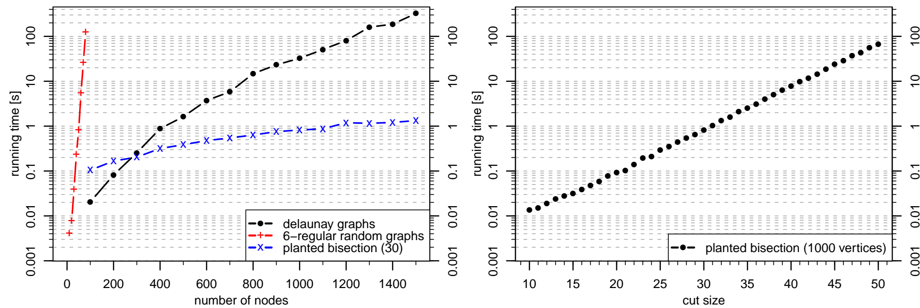

Fig. 4. Running times on various synthetic graph classes.

Finally, we note that all versions of the algorithm have exponential dependence on U. This suggests an obvious approach for finding the optimum bisection *opt* when it is not known: run the algorithm repeatedly with increasing values of U, and it will find the solution as soon as it gets an input $U > opt$. The total time should not be much higher than running only from $opt + 1$. Since our focus is on lower bounds, we use $U = opt + 1$ for all remaining experiments.

Asymptotics. For a better understanding of the asymptotic behavior or our method, we run it on synthetic graphs. Here we test version D (without decomposition) on three graph classes. The first consists of *Delaunay graphs*, each representing the Delaunay triangulation of n points picked at random in the unit square. The second class consists of 6-regular *random graphs*, built as the union of 6 random perfect matchings. Finally, we consider 6-regular graphs with *planted bisections* of size 30 (we take the union of two 6-regular random graphs with $n/2$ vertices each and add 30 random edges between them). These three classes have similar density ($m = 3n$), but differ significantly on the expected minimum bisection size: roughly $2\sqrt{n}$ for Delaunay, $\Theta(n)$ for random graphs, and exactly 30 for planted bisections. Figure 4 (left) shows the average running times (over 10 runs) of our algorithm as n varies (always with $U = opt + 1$).

It is clear that running times depend more strongly on the bisection than on graph size. Our method quickly becomes impractical for random graphs (it takes more than two minutes on graphs with 80 nodes), but is much more practical for Delaunay triangulations. For random graphs with small planted bisections, the running time is essentially linear in n: all branch-and-bound trees have roughly 1050 nodes. Figure 4 (right) also considers 6-regular random graphs with planted bisections, but now with $n = 1000$ and varying bisection size. As expected, running times increase exponentially with the bisection.

DIMACS Instances. This analysis indicates our algorithm should be able to handle fairly large real-world inputs, as long as their optimum bisection is not too large. To test this, we consider instances from the 10th DIMACS Implementation Challenge [3]. Since the challenge is meant to evaluate mainly heuristics, most instances are quite large (up to hundreds of millions of vertices) and have large bisections. Still, our algorithm can solve a wide variety of (smaller) instances

Table 1. Number of branch-and-bound nodes (BB) and total CPU time on DIMACS Challenge instances with $\epsilon = 0$; data uses DryadOpt, and all other runs are sequential.

CLASS	NAME	n	m	opt	BB	TIME [S]
clustering	karate	34	78	10	4	0.00
	chesapeake	39	170	46	110 138	3.08
	dolphins	62	159	15	110	0.01
	lesmis	77	820	61	3 905 756	230.30
	polbooks	105	441	19	8	0.00
	football	115	613	61	7 301	1.08
	power	4 941	6 594	12	94	0.21
delaunay	delaunay_n10	1 024	3 056	63	14 361	18.25
	delaunay_n11	2 048	6 127	86	65 080	175.73
	delaunay_n12	4 096	12 264	118	474 844	2 711.73
	delaunay_n13	8 192	24 547	156	3 122 845	37 615.97
streets	luxembourg	114 599	119 666	17	786	91.17
walshaw	data	2 851	15 093	189	495 569 759	5 750 387.82
	3elt	4 720	13 722	90	12 707	82.10
	uk	4 824	6 837	19	1 624	3.81
	add32	4 960	9 462	11	225	2.80
	whitaker3	9 800	28 989	127	7 044	133.04
	fe_4elt2	11 143	32 818	130	10 391	224.26
	4elt	15 606	45 878	139	25 912	769.35

to optimality. We consider instances from four classes: clustering, Delaunay triangulations, road networks, and instances from Walshaw's graph partitioning repository [45]. For clustering instances, which are smaller, we use version D of our algorithm; for the three remaining series (delaunay, streets, and walshaw), which are larger and sparse, we also use decomposition by clumps.

Table 1 shows, for each instance, the number of nodes in the branch-and-bound tree (BB) and the total running time in seconds. As expected, running times depend more heavily on the size of the bisection than on the graph itself. In particular, our algorithm could easily solve luxembourg (a road network), even though it has more than 100 thousand vertices. It can also find the minimum bisections of reasonably large delaunay graphs, which are Delaunay triangulations of random points on the plane. Note that decomposition makes the algorithm asymptotically faster than the version tested in Figure 4. For several Walshaw instances, our algorithm proves (for the first time, to the best of our knowledge) that the best previously known bisections found by heuristics [4, 9, 25, 26, 44] are indeed optimal. Finally, we also find exact solutions for some small clustering graphs, whose solutions are much larger relative to the graph size.

We also tested instances from the redistricting class with arbitrary vertex weights and (unit edge costs), using version D and DryadOpt. These instances are hard for our method for $\epsilon = 0$ (it is not tuned to handle zero-weight vertices), but Table 2 shows[3] that our algorithm can still find optimal solutions for $\epsilon = 0.03$.

We omit detailed results on non-DIMACS instances due to space constraints, and refer the reader to the full paper [15]. It shows that we can solve VLSI

[3] An earlier version of this paper misstated the solution value for ct2010.

Table 2. Results on redistricting instances with $\epsilon = 0.03$ (using DryadOpt).

NAME	n	m	opt	BB	TIME [S]
de2010	24 115	58 028	36	216	7
ri2010	25 181	62 875	107	36 976	4 702
vt2010	32 580	77 799	112	31 483	4 896
nh2010	48 837	117 275	146	1 102 716	288 090
ct2010	67 578	168 176	150	347 130	180 509
me2010	69 518	167 738	140	1 321 389	565 727
nj2010	169 588	414 956	150	875 842	1 184 886

instances and computer graphics meshes with tens of thousands of vertices, and road networks with more than a million. In fact, for graphs with small bisections, our method often outperforms the best mathematical programming approaches, such as those of Armbruster et al. [1, 2] and Hager et al. [24]. These algorithms have much better performance on small graphs with large bisections, however.

7 Final Remarks

We presented a novel branch-and-bound algorithm that can find exact solutions to remarkably large real-world instances, particularly those with small bisections. The resulting algorithm is quite practical, and could conceivably be used within graph partitioning heuristics, which often need to find bisections of small subproblems [11, 14, 30, 40]. It may be possible to obtain further speedups: improved branching heuristics, primal algorithms, and strengthened versions of the packing bound (for weighted edges) should all help. A potential topic for future research is whether the techniques we propose (particularly decomposition, but also the packing lower bound) can be effectively integrated into mathematical programming methods. A combination of recent results [16, 27] suggests that the minimum bisection problem is fixed-parameter tractable (parameterized by minimum bisection size) for planar and almost planar graphs, such as road networks, VLSI, and meshes. It would be interesting to know whether similar ideas could give nontrivial performance guarantees to some variant of our algorithm.

Acknowledgments. We thank Diego Nehab for the benchmark meshes and visualization tools, Tony Wirth for discussions on the hardness of various subproblems.

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, 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.
3. 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`.

4. 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.
5. 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.
6. L. Brunetta, M. Conforti, and G. Rinaldi. A branch-and-cut algorithm for the equicut problem. *Mathematical Programming*, 78:243–263, 1997.
7. M. Budiu, D. Delling, and R. F. Werneck. DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines. In *IPDPS*, pp. 1278–1289, 2011.
8. T. N. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.
9. P. Chardaire, M. Barake, and G. P. McKeown. A PROBE-Based Heuristic for Graph Partitioning. *IEEE Transactions on Computers*, 56(12):1707–1720, 2007.
10. F. Chataigner, L. B. Salgado, and Y. Wakabayashi. Approximation and Inapproximability Results on Balanced Connected Partitions of Graphs. *Discrete Mathematics and Theoretical Computer Science*, 9(1):177–192, 2007.
11. C. Chevalier and F. Pellegrini. PT-SCOTCH: A Tool for Efficient Parallel Graph Ordering. *Parallel Computing*, 34:318–331, 2008.
12. J. Chlebíková. Approximating the Maximally Balanced Connected Partition Problem in Graphs. *Information Processing Letters*, 60(5):223–230, 1996.
13. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *SEA*, LNCS 6630, pp. 376–387. Springer, 2011.
14. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *IPDPS*, pp. 1135–1146. IEEE, 2011.
15. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Exact Combinatorial Branch-and-Bound for Graph Bisection. In *ALENEX*, pp. 30–44. SIAM, 2012.
16. E. D. Demaine, M. Hajiaghayi, and K. Kawarabayashi. Contraction Decomposition in H-Minor-Free Graphs and Algorithmic Applications. In *STOC*, pp. 441–450, 2011.
17. A. E. Feldmann and P. Widmayer. An $O(n^4)$ time algorithm to compute the bisection width of solid grid graphs. In *ESA*, LNCS 6942, pp. 143–154, 2011.
18. 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.
19. 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.
20. M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some Simplified $\mathcal{NP}$-Complete Graph Problems. *Theoretical Computer Science*, 1:237–267, 1976.
21. B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.
22. A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck. Maximum flows by incremental breadth-first search. In *ESA*, LNCS 6942, pp. 457–468, 2011.
23. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
24. W. W. Hager, D. T. Phan, and H. Zhang. An exact algorithm for graph partitioning. Submitted for publication. Available at `http://www.math.ufl.edu/~hager/papers/GP/cqb.pdf`, 2011.
25. M. Hein and T. Bühler. An Inverse Power Method for Nonlinear Eigenproblems with Applications in 1-Spectral Clustering and Sparse PCA. In *NIPS*, pp. 847–855, 2010.

26. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *SC*, p. 28. ACM Press, 1995.
27. 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.
28. D. R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal the ACM*, 43(4):601–640, 1996.
29. S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite. *INFORMS Journal on Computing*, 12:177–191, 2000.
30. 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.
31. 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.
32. 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.
33. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
34. R. J. Lipton and R. Tarjan. Applications of a Planar Separator Theorem. *SIAM Journal on Computing*, 9:615–627, 1980.
35. 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*, p. 6. ACM, 2009.
36. 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.
37. 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*, LNCS 1067, pp. 493–498. Springer, 1996.
38. H. Räcke. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *STOC*, pp. 255–263. ACM Press, 2008.
39. 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.
40. P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *ALENEX*, pp. 16–29. SIAM, 2012.
41. M. Sellmann, N. Sensen, and L. Timajev. Multicommodity Flow Approximation Used for Exact Graph Partitioning. In *ESA*, LNCS 2832, pp. 752–764, 2003.
42. N. Sensen. Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows. In *ESA*, LNCS 2161, pp. 391–403, 2001.
43. J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.
44. 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.
45. A. J. Soper, C. Walshaw, and M. Cross. The Graph Partitioning Archive, 2004. `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.
46. C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoulès, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pp. 27–58. Civil-Comp Ltd., 2007.
47. 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.