Exact Combinatorial Branch-and-Bound for Graph Bisection

Daniel Delling* Andrew V. Goldberg† Ilya Razenshteyn‡ Renato F. Werneck§

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 [45, 51], computer vision [33], divide-and-conquer algorithms [35], VLSI circuit layout [6], distributed computing [36], and route planning [14]. Unfortunately, the bisection problem is NP-hard [21] for general graphs, with a best known approximation ratio of $O(\log n)$ [40]. Only some restricted graph classes, such as grids without holes [18] and graphs with bounded treewidth [28], have known polynomial-time solutions.

In practice, there are numerous general-purpose heuristics for graph partitioning, such as METIS [31], SCOTCH [12, 38], Jostle [50], and KaFFPaE [42]. Successful heuristics tailored to particular graph classes, such as DibaP [37] (for meshes) and PUNCH [15] (for road networks), are also available. These algorithms are quite fast (often running in near-linear time) and can handle very 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 [34]. Most of these algorithms use sophisticated machinery to obtain lower bounds, such as multicommodity flows [43, 44], or linear [3, 7, 20], semidefinite [1, 3, 30], and quadratic programming [25]. 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 [19] 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

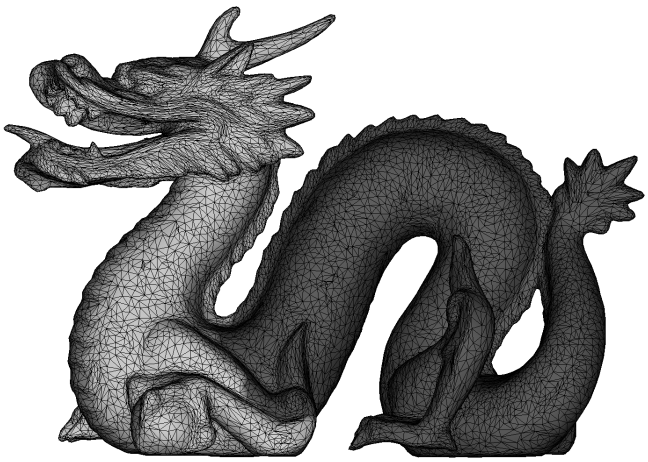

Figure 1: Example of a minimum bisection.

*Microsoft Research Silicon Valley. dadellin@microsoft.com

†Microsoft Research Silicon Valley. goldberg@microsoft.com

‡Lomonosov Moscow State University. ilyaraz@gmail.com. This work was done while the author was at Microsoft Research Silicon Valley.

§Microsoft Research Silicon Valley. renatow@microsoft.com

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). See Figure 1 for an example. 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) elaborate 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 to the input by solving a small number of (usually much easier) subproblems independently. Finally, we present (in Section 6) a careful experimental analysis of our techniques.

2 Preliminaries

We use $G = (V, E)$ to denote the input graph, with $n = |V|$ vertices and $m = |E|$ edges. Each vertex $v \in V$ may have an associated integral *weight*, denoted by $w(v)$, and each edge $e \in E$ has an associated 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. (Section 3.4 shows how we can handle less restrictive settings.)

A standard technique for finding exact solutions to NP-hard problems is *branch-and-bound* [22, 34]. 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, 3, 7, 20, 25, 30] 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 [9, 15] *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 possible). 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 have similar sizes; even in this case, the corresponding minimum cuts are 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*.

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 compute a lower bound for f^*.

It works as follows (see Figure 2). Let $G' = G \setminus (A \cup B)$ be the subgraph of G induced by the vertices that are

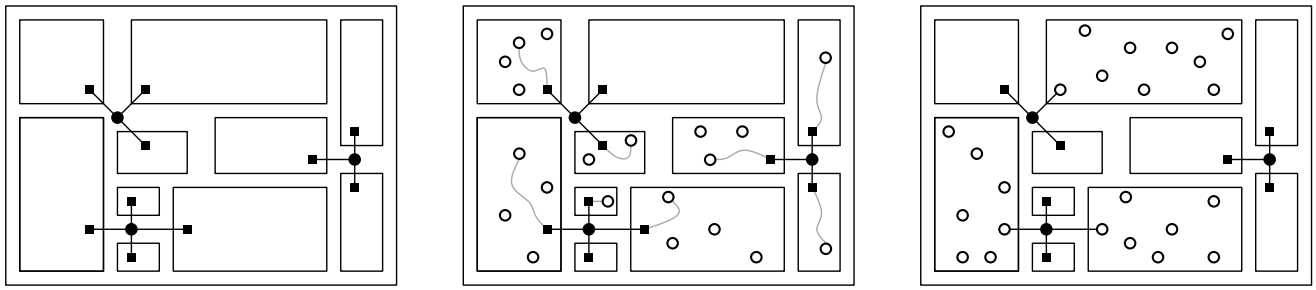

Figure 2: The packing bound. Filled circles are vertices in B; their free neighbors (filled squares) form a set R. **Left**: We first partition the free vertices into connected cells, each with at most one element of R. **Middle**: Given any extension A^+ (hollow circles), the number of nontrivial cells it touches (eight) is a lower bound on the minimum (B, A^+) cut. **Right**: The extension A^+ that hits the fewest cells (three) is a lower bound on any valid extension.

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 such partition is valid; as we shall see, we get better lower bounds if the cells containing elements in R are as large as possible and have roughly the same size.) 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 3.1. *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 3.1 gives the lowest possible bound (for a fixed partition into connected cells). We can build this extension with 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, it makes sense to 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 3.1. *The greedy packing algorithm finds a lower bound on the value of any bisection consistent with (A, B).*

3.1 Computing Packing Lower Bounds. Note that 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 [13, 11], but we propose two heuristics that work well in practice.

The first heuristic 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.

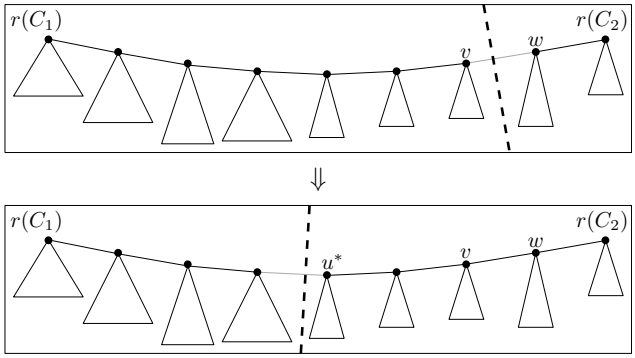

Figure 3: Packing local search. **Left**: The boundary edge (v, w) determines a path between cells C_1 and C_2 (triangles are subtrees). **Right:** Splitting the path at a different point leads to a more balanced partition.

The local search works by moving 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 $w \in C_2$. Without loss of generality, 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, the largest cell) from v to $r(C_1)$. (See Figure 3.) 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 corresponding to the most balanced final state (in which the sizes of C_1 and C_2 would be closest). If this is more balanced than the current state, we switch.

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. (Note that theoretical improvements would be possible using a dynamic-tree data structures [46], but in practice they are quite costly for graphs of moderate diameter [49], as in our case.)

3.2 Combining Packing and Flows. Although we have two lower bounds, based on packing and flows, we cannot simply add them 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.) Simply 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 must 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 algorithm provides valid lower bounds regardless of the flow decomposition used, but its packing portion is better if it has more edges to work with. We therefore favor flow decompositions with as few edges as possible: instead of using the standard push-relabel approach [24], we prefer an augmenting-path algorithm that greedily sends flows along shortest paths. Our implementation uses a simplified version of the IBFS (*incremental breadth first search*) algorithm [23], which is about as fast as push-relabel on our test instances.

3.3 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 $f' = f + x(v)$. (See Figure 4.) 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. In general, this test will succeed 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

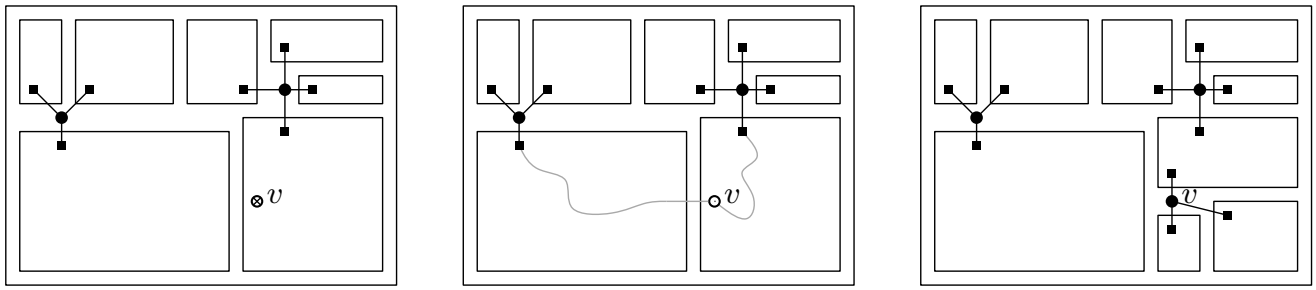

Figure 4: Illustration of forced assignments. **Left:** State after initial bounds have been computed (with flow edges already removed). Filled circles belong to B, and v is free. **Middle**: Assigning v to A would increase the flow bound. **Right:** Assigning v to B can increase the packing bound by splitting a cell into several ones.

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.[1] 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.

3.4 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.

Second, we prefer to branch on vertices that appear in large cells (from the packing bound). By branching on a large cell, we allow it to be split, thus improving 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.

4.1 Filtering. For some graph classes (notably road networks), high-degree vertices are often separated by a small cut from the rest of the graph [15]. This makes degrees poor proxies for connectivity. We could obtain a more robust measure of connectivity by reusing the packing algorithm described in Section 3. For each

[1] Note that, if the cell containing v is nontrivial, we could split it into $\deg_{G_f}(v) + 1$ cells by keeping the original root. For simplicity and performance, our implementation does not do this.

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. Unfortunately, computing this bound for every vertex in the graph would be quite expensive, particularly for large road networks.

Instead of explicitly computing the packing bound for every vertex in the graph, we propose a *filtering routine*. Its goal is more modest: determine if some of the most promising vertices (those with the highest degrees) are actually well-connected to the rest of the graph. This is done in two stages.

First, we determine whether each vertex is separated by a cut with exactly one or two edges from the remainder of the graph. We use the algorithm of Pritchard and Thurimella [39] to find all 1-cuts and 2-cuts in the graph in linear time. (These cuts are quite numerous in road networks [15].) For a vertex v, define $\text{cut}(v) = 1$ if it is inside a 1-cut of size at most $n/10$. Otherwise, if v is contained in a 2-cut of size at most $n/10$, let $\text{cut}(v) = 2$. For all other vertices v, let $\text{cut}(v) = \deg(v)$.

The second stage of filtering computes the packing bound for the set S containing the $2U$ vertices v with the highest $\text{cut}(v)$ values (recall that U is the best known upper bound), with ties broken at random. Let $\text{pack}(v)$ be the corresponding values. Let δ be the floor of the average value of $\text{pack}(v)$ over all vertices $v \in S$. For all vertices $w \notin S$, we set $\text{pack}(w) = \delta$.

The branch-and-bound algorithm then uses the standard criterion $(\text{dist}(v) \cdot \text{csize}(v) \cdot \text{conn}(v)^2)$ to choose the next vertex to branch on, but using a modified definition of connectivity: $\text{conn}(v) = \text{cut}(v) \cdot \text{pack}(v)$.

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 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 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 neighbors outside the region) is substantially higher, lower bounds should increase much faster as we traverse the branch-and-bound tree. The obvious problem with this approach 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 [29], 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. [9] 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 $(E_0, E_1, \ldots, E_U)$. 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. Then, use our standard algorithm to find the optimum bisection U_i of each graph G_i independently, and pick the best.

THEOREM 5.1. *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 $U + 1$ independent subproblems, but the high-degree vertices introduced by contraction 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 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 (the same partial assignment may be reached in different branches), but this does not affect correctness.

5.1 Finding a Decomposition. The 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, our edge partitioning algorithm should 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. 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 the remaining vertices.) Finally, the *shape* of each clump matters: all else being equal, we would like its expansion (number of neighbors outside the clump) to be as large as possible.

To achieve these goals, 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.

The goal of the generation routine is to build a set F of clumps (initially empty) that partition all the edges in the graph. It does so by maintaining a set C of *candidate* clumps, which are not necessarily disjoint and may not include all edges in the graph. The clumps in C are high-expansion subpaths extracted from BFS trees grown from random vertices. (Because they minimize the number of internal edges, such paths tend to have high expansion.) Once there are enough candidates in C, the algorithm transfers a few clumps from C to the final set F. The clumps are picked from C greedily, according to their expansion, and observing the constraint that clumps in F must be edge-disjoint. Once C no longer has suitable clumps with high enough expansion (higher than a certain threshold τ), a new iteration of the algorithm starts: it repopulates C by growing new BFS trees, then transfers some of the resulting candidate clumps to F. This algorithm stops when F is complete, i.e., when every edge in the graph belongs to a clump in F. To ensure convergence, the algorithm gradually decreases the threshold τ between iterations: initially only clumps with very high expansion are added to F, but eventually even single-edge clumps are allowed. (The interested reader will find additional details in Section A, in the appendix.)

The allocation phase distributes the clumps to the $U+1$ subproblems $(E_0, E_1, \ldots, E_U)$, which are initially empty. It allocates 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 ties broken arbitrarily. 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 of each E_i explicitly, updating it whenever a new clump is added.) This ensures clumps are well spread in each subproblem.

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 couple of particularly hard instances (clearly marked), we ran a distributed version of the code using the DryadOpt framework [8]. 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. We always report the total CPU time, the sum of the times spent by our C++ code on all 402 cores (400 on the cluster plus 2 on our standard machine). Note that this excludes the communication overhead, which is negligible. Unless otherwise mentioned, we find perfectly balanced partitions ($\epsilon = 0$).

6.1 Parameter Evaluation. We start by considering the effects of each improvement we propose on per-

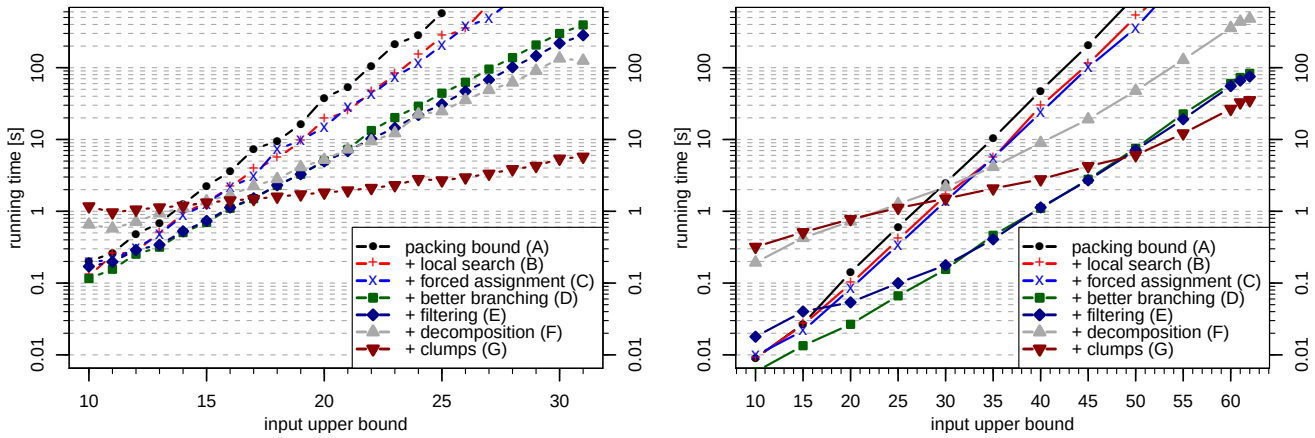

Figure 5: Running times of increasingly sophisticated versions of our algorithm as a function of the upper bound U on the inputs alue5067 (left) and mannequin (right).

formance. 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 [32]) with 3524 vertices, 5560 edges, and optimum bisection $opt = 30$; mannequin is a mesh (triangulation) used in computer graphics studies [41] with 689 vertices, 2043 edges, and $opt = 61$.

Figure 5 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 is the most basic: it 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 filters well-connected branching vertices as well (as explained in Section 4.1). 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. (Note that the vertical axis uses a logarithmic scale.) 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 $U \geq 3$—too slow to appear in the plot.

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 original 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), and for alue5067 (with $U = opt + 1$) 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). For the remaining experiments, we usually do not use filtering (except as noted, for large road networks).

Finally, we note that all versions of the algorithm have exponential dependence on U. This suggests an obvious algorithm for finding the optimum bisection opt of an instance (in case it is not known). We can simply run our algorithm repeatedly with increasing values of U; the algorithm will find the solution as soon as it gets an input $U > opt$. Since the algorithm is exponential in U, the total running time of this approach is not much higher than running directly from $opt + 1$. Since our focus is on lower bounds (and to minimize fluctuations), we actually use $U = opt + 1$ for all remaining experiments, unless otherwise noted. Section 6.5 will examine this issue in more detail.

6.2 Asymptotics. We now run our algorithm on synthetic graphs to get a better understanding of its asymptotic behavior. Here we test version D (without decomposition) on three graph classes. The first one

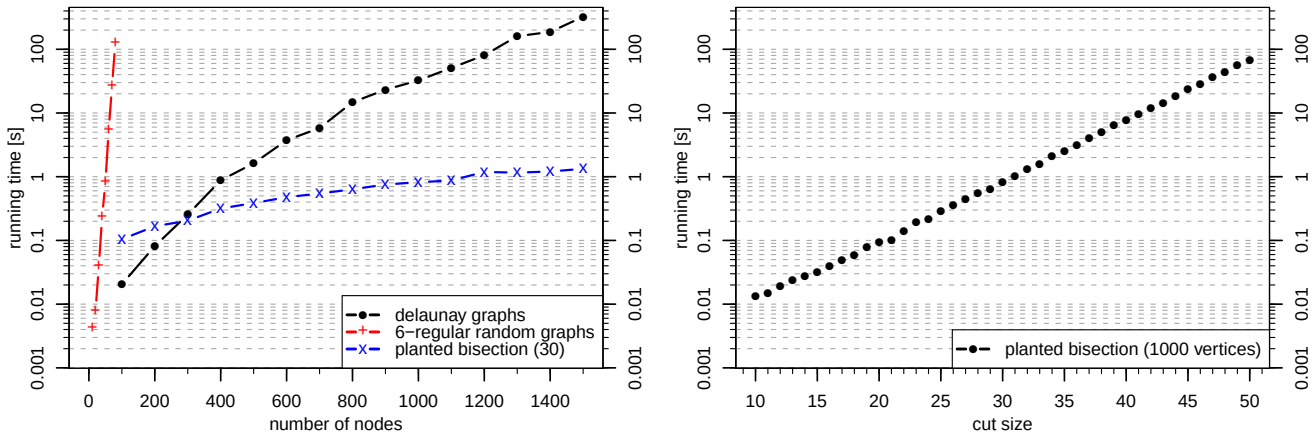

Figure 6: Running times on various synthetic graph classes.

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). Note that these three classes have roughly the same density ($m = 3n$), but differ significantly on the expected minimum bisection size: approximately $2\sqrt{n}$ for Delaunay, $\Theta(n)$ for random graphs, and exactly 30 for planted bisections. Figure 6 (left) shows the average running times (over 10 runs) of our algorithm as n varies. We always use $U = opt + 1$.

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 (with small fluctuations).

Figure 6 (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 size.

6.3 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 [4]. 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 to optimality. We consider instances from four classes: clustering, Delaunay triangulations, road networks, and instances from Walshaw's graph partitioning repository [48]. 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 the detailed performance of our algorithm. For each instance, we show the number of vertices (n), the number of edges (m), and the optimum bisection value (opt), followed by the total number of nodes in the branch-and-bound tree (BB) and the total running time of our algorithm 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.

We can also find the minimum bisections of reasonably large delaunay graphs, which are Delaunay triangulations of random points on the plane. Note that this version of the algorithm (with decomposition) is actually asymptotically better than the one tested in Figure 6 (without decomposition).

We can also deal with some inputs from the Walshaw repository [48]. For every instance in the table, we show (for the first time, to the best of our knowledge) that the best previously known bisections—found by heuristics [5, 10, 26, 27, 47]—are indeed optimal.

Our algorithm can also find exact solutions for some small clustering graphs. For these instances, the minimum bisection value is much larger relative to the graph size.

Table 1: Instances from the 10th DIMACS Implementation Challenge with $\epsilon = 0$; BB is the number of branch-and-bound nodes and TIME is the total CPU time. All runs are sequential except for data, which uses DryadOpt.

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

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. All runs are sequential except for dragon-043571, which uses DryadOpt.

CLASS	NAME	n	m	opt	BB	TIME [S]
mesh	dolphin	284	846	26	386	0.14
	mannequin	689	2 043	61	41 702	31.34
	venus-711	711	2 127	43	1 370	1.17
	beethoven	2 521	7 545	72	18 779	57.77
	venus	2 838	8 508	83	5 180	19.95
	cow	2 903	8 706	79	19 911	66.58
	fandisk	5 051	14 976	137	820 604	5 183.86
	gargoyle	10 002	30 000	175	2 607 924	46 703.49
	feline	20 629	61 893	148	146 973	4 564.73
	dragon-043571	21 890	65 658	148	52 016 708	5 854 478.60
road	ny	264 346	365 050	18	1 584	437.58
	bay	321 270	397 415	18	1 702	555.41
	col	435 666	521 200	29	2 604	1 583.80
	fla	1 070 376	1 343 951	25	830	2 699.00
	nw	1 207 945	1 410 387	18	264	1 563.08
vlsi	alue5067	3 524	5 560	30	1 620	3.62
	alue7065	34 046	54 841	80	18 485	504.08
	alue7080	34 479	55 494	80	17 497	478.23

6.4 Assorted Large Instances. We now consider some natural classes of large instances with relatively small (but nontrivial) bisections. We take three classes of inputs: meshes (triangulations) representing various objects [41], road networks (from the 9th DIMACS Implementation Challenge [17], on shortest paths), and VLSI instances [32] (grid graphs with holes). The performance of our algorithm is summarized in Table 2. We use decomposition by clumps for all three classes, and filtering for road (but not mesh or vlsi). Once again, we use $\epsilon = 0$ and $U = opt + 1$.

As Table 2 shows, the performance of our algorithm again depends more strongly on the bisection than on graph size. In particular, we can solve fla and nw, with more than a million vertices, in less than an hour. These and other road networks need surprisingly few branch-and-bound nodes; for these instances, a large fraction of the total time is usually taken by clump decomposition and filtering. The other two classes considered (vlsi and mesh) have larger bisections, and need substantially more branch-and-bound nodes. Even so, we can handle VLSI instances with tens of thousands of vertices, as well as reasonably large meshes. As Figures 7 and 8 in the appendix illustrate, these instances are by no means trivial. In particular, to solve dragon-043571 (whose optimal solution is illustrated in Figure 1), we had to use our distributed implementation. In contrast, we can solve feline, which has similar size and solution value but is much more regular, in less than two CPU hours.

6.5 Comparison with Other Approaches. We now compare our algorithm to the best mathematical programming techniques we are aware of. Table 3 compares the running times of our algorithm (version D) with the best results reported by Armbruster [1] and Hager et al. [25] on a few publicly available [2] instances from the literature, including meshes (mesh), VLSI instances (gap, taq), and graphs derived from sparse symmetric linear systems (KKT). For each instance we use the most common value of ϵ in the literature (either 0.00 or 0.05). Some instances have edges with small integral weights, which are converted into parallel edges by our algorithm (this is accounted for in the value of m reported in the table). Note that the algorithms were run on different (but roughly comparable) Intel machines: Pentium 4 540 (3.2 GHz) for Armbruster [1], Xeon X5355 (2.66 GHz) for Hager et al. [25], and Core 2 Duo E8500 (3.16 GHz) for our algorithm. All runs are sequential.

Besides running our algorithm with $U = opt + 1$ (as in our standard experiments), we also consider a version in which no upper bound U is given. This version repeats our basic algorithm with increasing values of U: starting from $U_0 = 1$, it sets $U_i = \lceil 1.5U_i \rceil$ in each step i and stops when it finds a bisection that is strictly better than the current U_i.

The table shows that either version of our algorithm can be faster (sometimes substantially so) than the mathematical programming approaches for instances

Table 3: Comparison between the exact approaches of Armbruster [1] and Hager et al. [25] and two versions of our method, with different input upper bounds U. Note that some results by Hager et al. [25] are not available. Running times are in seconds.

					$U = opt + 1$		no U given		OTHER ALGO.	
NAME	n	m	ϵ	opt	BB	TIME	BB	TIME	[Arm07]	[HPZ11]
gap2669.6182.int	2 669	12 280	0.05	74	5 612	26.88	7 329	34.90	651.03	—
gap2669.24859.int	2 669	29 037	0.05	55	6	0.06	28	0.15	348.95	—
KKT_lowt01_m2	82	260	0.05	13	26	0.02	110	0.03	0.19	—
KKT_putt01_m2	115	433	0.05	28	4 161	0.31	10 704	0.81	1.67	1.51
mesh.35.54.int	35	54	0.00	3	10	0.02	20	0.02	0.00	—
mesh.69.112.int	69	112	0.00	4	20	0.02	36	0.02	0.03	0.36
mesh.70.120.int	70	120	0.00	7	17	0.02	27	0.02	0.54	—
mesh.74.129.int	74	129	0.00	8	39	0.02	232	0.05	1.41	0.49
mesh.137.231.int	137	231	0.00	7	43	0.03	62	0.03	2.67	5.35
mesh.274.231.int	137	231	0.00	8	56	0.03	268	0.07	2.63	7.88
mesh.138.232.int	138	232	0.00	8	75	0.03	508	0.10	10.22	6.91
mesh.148.265.int	148	265	0.00	7	21	0.02	34	0.02	0.77	4.30
mesh.274.469.int	274	469	0.00	7	37	0.03	50	0.03	8.52	24.62
taq170.424.int	170	4 317	0.05	55	4 807	2.67	5 291	3.00	28.68	—
taq228.692.int	228	5 759	0.05	63	519	0.41	1 072	0.82	4.20	—
taq278.396.int	278	5 158	0.05	37	714	0.38	1 050	0.57	1.56	—
taq1021.2253.int	1 021	4 510	0.05	118	78 862	125.92	83 286	134.61	169.65	—

with relatively small bisections. We stress, however, that the algorithms based on mathematical programming can handle graphs with large bisections (not shown in the table) much better. Our algorithm is not competitive in such cases. For example, Armbruster can solve the instance alue6112.16896 in 80 minutes, whereas our algorithm could not prove $opt = 272$ after a day, even with $U = opt + 1$ given.

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 [12, 15, 31, 42]. 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, 28] 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, and the referees 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. `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. `www.cc.gatech.edu/dimacs10/index.shtml`.

[5] 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.

[6] 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.

[7] L. Brunetta, M. Conforti, and G. Rinaldi. A Branch-and-Cut Algorithm for the Equicut Problem. *Mathematical Programming*, 78:243–263, 1997.

[8] M. Budiu, D. Delling, and R. F. Werneck. DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines. In *IPDPS*, pp. 1278–1289, 2011.

[9] T. N. Bui, S. Chaudhuri, F. Leighton, and M. Sipser. Graph Bisection Algorithms with Good Average Case Behavior. *Combinatorica*, 7(2):171–191, 1987.

[10] P. Chardaire, M. Barake, and G. P. McKeown. A PROBE-Based Heuristic for Graph Partitioning. *IEEE Transactions on Computers*, 56(12):1707–1720, 2007.

[11] 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.

[12] C. Chevalier and F. Pellegrini. PT-SCOTCH: A Tool for Efficient Parallel Graph Ordering. *Parallel Computing*, 34:318–331, 2008.

[13] J. Chlebíková. Approximating the Maximally Balanced Connected Partition Problem in Graphs. *Information Processing Letters*, 60(5):223–230, 1996.

[14] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *SEA*, LNCS 6630, pp. 376–387. Springer, 2011.

[15] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *IPDPS*, pp. 1135–1146. IEEE, 2011.

[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] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74. American Mathematical Society, 2009.

[18] 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.

[19] A. Felner. Finding Optimal Solutions to the Graph Partitioning Problem with Heuristic Search. *Annals of Mathematics and Artificial Intelligence*, 45(3–4):293–322, 2005.

[20] 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.

[21] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some Simplified $\mathcal{NP}$-Complete Graph Problems. *The-*

oretical Computer Science, 1:237–267, 1976.

[22] B. Gendron and T. G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[23] 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. Springer, 2011.

[24] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum-Flow Problem. *Journal of the ACM*, 35(4):921–940, 1988.

[25] W. W. Hager, D. T. Phan, and H. Zhang. An Exact Algorithm for Graph Partitioning. Submitted for publication. Available at `www.math.ufl.edu/~hager/papers/GP/cqb.pdf`, 2011.

[26] 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.

[27] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *SC*, p. 28. ACM Press, 1995.

[28] 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.

[29] D. R. Karger and C. Stein. A New Approach to the Minimum Cut Problem. *Journal the ACM*, 43(4):601–640, 1996.

[30] S. E. Karisch, F. Rendl, and J. Clausen. Solving Graph Bisection Problems with Semidefinite Programming *INFORMS Journal on Computing*, 12:177–191, 2000.

[31] G. Karypis and G. Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *Journal on Scientific Computing*, 20(1):359–392, 1999.

[32] T. Koch, A. Martin, and S. Voß. SteinLib: An Updated Library on Steiner Tree Problems in Graphs. TR 00-37, Konrad-Zuse-Zentrum Berlin, 2000.

[33] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut Textures: Image and Video Synthesis using Graph Cuts. *ACM Transactions on Graphics*, 22:277–286, 2003.

[34] A. H. Land and A. G. Doig. An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520, 1960.

[35] R. J. Lipton and R. Tarjan. Applications of a Planar Separator Theorem. *SIAM Journal on Computing*, 9:615–627, 1980.

[36] 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.

[37] 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.

[38] 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.

[39] D. Pritchard and R. Thurimella. Fast Computation of Small Cuts via Cycle Space Sampling. *ACM Transaction on Algorithms*, 7:46:1–46:30, 2011.

[40] H. Räcke. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *STOC*, pp. 255–263. ACM Press, 2008.

[41] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient Traversal of Mesh Edges Using Adjacency Primitives. *ACM Transactions on Graphics*, 27:144:1–144:9, 2008.

[42] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *ALENEX*. SIAM, 2012.

[43] M. Sellmann, N. Sensen, and L. Timajev. Multicommodity Flow Approximation Used for Exact Graph Partitioning. In *ESA*, LNCS 2832, pp. 752–764, 2003.

[44] N. Sensen. Lower Bounds and Exact Algorithms for the Graph Partitioning Problem Using Multicommodity Flows. In *ESA*, LNCS 2161, pp. 391–403, 2001.

[45] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[46] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[47] 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.

[48] A. J. Soper, C. Walshaw, and M. Cross. The Graph Partitioning Archive, 2004. `staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.

[49] R. E. Tarjan and R. F. Werneck. Dynamic Trees in Practice. *ACM Journal of Experimental Algorithmics*, 14:4.5:1–4.5:23, 2009.

[50] 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.

[51] 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 Creating a Decomposition: Detailed Version

We now discuss the clump generation routine (sketched in Section 5.1) in more detail. Recall that the goal of this routine is to build a set F of clumps, initially empty, which partition all the edges in the graph. As already mentioned, it does so by maintaining a set C of candidate clumps (which may intersect and do not necessarily contain all edges), and gradually transferring clumps from C to F.

The basic algorithm works in iterations, each consisting of two phases: generation and selection. Consider iteration i, with a certain threshold τ_i. The *generation phase* creates new clumps to be added to the candidate set C. It does so by growing BFS trees from a constant number (5 in our implementation) of vertices picked at random. From each tree T, we repeatedly extract a series of disjoint paths and add them to C. We choose these paths greedily (from high to low expansion), but restrict ourselves to paths that (1) have at most s edges (where s is to be defined later) and (2) contain no edge that is already in F. The *selection phase* then extracts from C all clumps with expansion at least τ_i, and does so in greedy order (from higher to lower expansion). A clump c is added to F if no edge in c is already in F; otherwise, it is simply discarded. If F is not complete by the end of the iteration, we start iteration $i+1$ with $\tau_{i+1} = \lfloor 0.9\tau_i \rfloor$.

This is the basic algorithm, but there are still some details to specify.

First, we need to explain how the maximum clump size s is chosen. We must balance two desired properties: clumps should not be much smaller than the optimum cut size, and each subproblem should have multiple clumps. For graphs with m edges and upper bound U, we set $s = \left\lceil \min\left\{4U, \frac{m}{10U}\right\}\right\rceil$. This works well in practice, though individual instances could benefit from additional tuning.

A second issue we must worry about is space. Note that some low-expansion clumps (those with expansion lower than τ_i) are kept in C between iterations, since they may be useful once the threshold τ gets small enough. To keep the size of C manageable, we go over all clumps and discard those that have at least one edge in F—they will never be used. Moreover, we also avoid adding to C clumps that are too small to start with: when extracting paths from a tree T, we only consider those whose expansion is at least $x_T/4$, where x_T is the highest-expansion path in T that is valid (i.e., has at most s edges and no edge in common with F).

A third detail is related to the running time of the algorithm. For large graphs, growing full BFS trees can be rather expensive. We therefore adjust the algorithm to grow smaller trees as it progresses, and make sure these trees are grown in regions of the graph that still have unused edges. More precisely, we keep a set of *candidate roots* R: these are vertices with at least one incident unused edge (i.e., an edge that is not yet in F). We only grow trees from R. Moreover, each BFS tree is only allowed to scan $|R|$ vertices, ensuring tree generation gets faster as the algorithm progresses.

A final detail we consider is how to break ties when building the BFS trees. Whenever possible, we take parent edges that are not in F. In addition, to improve the quality of the paths we find, we pick as the parent of v the node that maximizes the expansion of the path up to v. (We can do so by looking at the adjacency lists of v, its candidate parents, and their parents.)

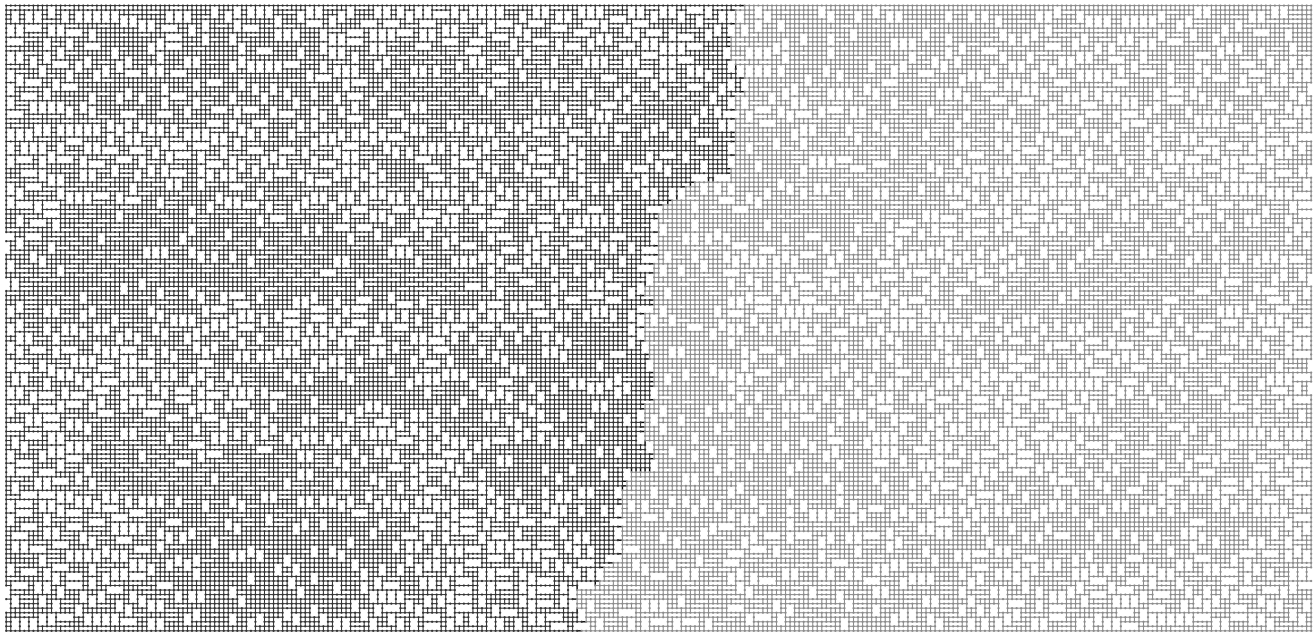

Figure 7: Minimum bisection of alue7065.

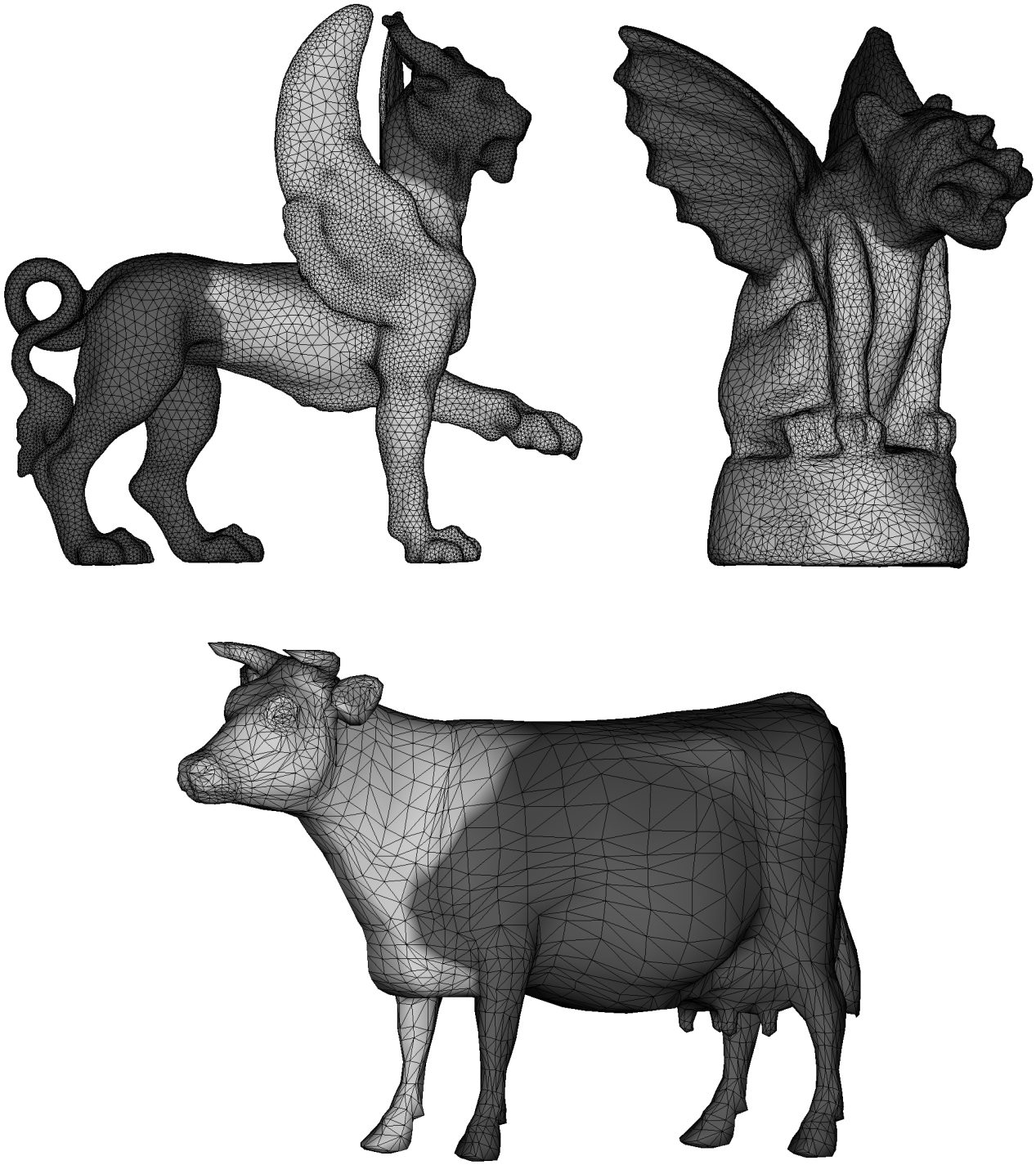

Figure 8: Minimum bisections of feline, gargoyle, and cow. Note that one of the cells in feline is not connected.