Graph Partitioning with Natural Cuts*

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 approach to graph partitioning based on the notion of *natural cuts*. Our algorithm, called *PUNCH*, has two phases. The first phase performs a series of minimum-cut computations to identify and contract dense regions of the graph. This reduces the graph size, but preserves its general structure. The second phase uses a combination of greedy and local search heuristics to assemble the final partition. The algorithm performs especially well on road networks, which have an abundance of natural cuts (such as bridges, mountain passes, and ferries). In a few minutes, it obtains excellent partitions for continental-sized networks.

1 Introduction

Partitioning a graph $G = (V, E)$ into many "well-separated" cells is a fundamental problem in computer science with applications in areas such as VLSI design [5], computer vision [23], image analysis [38], distributed computing [24], and route planning [10]. Most variants of this problem are known to be NP-hard [13] and focus on minimizing the *cut size*, i.e., the number of edges linking vertices from different cells. Given its importance, there is a rich literature on the problem, including a wealth of heuristic solutions (see [34, 37] for overviews).

A popular approach is *multilevel graph partitioning* (MGP), which generally works in three phases. During the first phase, the graph is iteratively shrunk by contracting edges. This is repeated until the number of remaining vertices is small enough to perform an expensive initial partitioning, the second stage of MGP. Finally, the graph is partially uncontracted, and local search is applied to improve the cut size. This approach can be found in many software libraries, such as SCOTCH [27], METIS [20], DiBaP [26], JOSTLE [37], CHACO [18], PARTY [28], and KaFFPaE [33].

Although MGP approaches can be used for road networks, they do not exploit the natural properties of such graphs in full. In particular, road networks are not uniform: there are densely populated regions (which should not be split)

* This is a condensed and slightly updated paper originally published at IPDPS'11 [8].
** This work was done while the third author was at Microsoft Research Silicon Valley.

close to natural separators like bridges, mountain passes, and ferries. Moreover, known MGPs focus on balancing the sizes of the cells while sacrificing either connectivity (METIS, SCOTCH, KaFFPaE) or cut size (DiBaP). This makes sense for more uniform graphs, such as meshes. However, many road network applications require cells to be connected and one does not want to sacrifice the cut size. Such applications include route planning [7, 9, 19, 25, 1], distribution of data [22], and computation of centrality measures [15].

This paper describes PUNCH (**P**artitioning **U**sing **N**atural **C**ut **H**euristics), a partitioning algorithm tailored to graphs containing natural cuts, such as road networks. Given a parameter U (the maximum size of any cell), PUNCH partitions the graph into cells of size at most U while minimizing the number of edges between cells. The algorithm runs in two phases: *filtering* and *assembly*.

The filtering phase aims to reduce the size of the graph significantly while preserving its overall structure. It keeps the edges that appear in *natural cuts*, relatively sparse cuts close to denser areas, and contracts other edges. The notion of natural cuts and efficient algorithms to compute them are the main contributions of our work. Note that to find a natural cut it is *not* enough to pick a random pair of vertices and run a minimum cut computation between them: because the average degree in road networks is small, this is likely to yield a trivial cut. We do better by finding minimum cuts between carefully chosen regions of the graph. Edges that never contributed to a natural cut are contracted, potentially reducing the graph size by orders of magnitude. Although smaller, the filtered (contracted) graph preserves the natural cuts of the input.

The second phase of our algorithm (assembly) is the one that actually builds a partition. Since the filtered graph is much smaller than the input, we can use more powerful (and time-consuming) techniques in this phase. Another important contribution of our work is a better local search algorithm for the second phase. Note that the assembly phase only tries to combine fragments (the contracted regions). Unlike existing partitioners, we do not disassemble fragments.

Note that we focus on finding partitions with small cells, but with no hard bound on the number of cells thus created. As already mentioned, previous work in this area has concentrated on finding *balanced* partitions, in which the total number of cells is bounded. We show how one can use simple heuristics to transform the solutions found by our algorithm into balanced ones.

We are not aware of any approach using min-cut computations to reduce the graph size in the context of graph partitioning. However, work on improving a partition is vast. For example, many of the algorithms within the MGP framework use local search based on vertex swapping, which improves the cut size by moving vertices from one cell to another. The most important ones are the FM [12] and KL [21] heuristics. The FM heuristic runs in worst-case linear time by allowing each vertex to be moved at most once. Local improvements based on minimum cuts often yield better results than greedy methods. For example, Andersen and Lang [2] run several minimum cut computations to improve the cut between two neighboring cells. Another common approach to optimize a cut between two cells is based on parametric minimum cut computation [6]. Be-

sides vertex swapping and minimum cuts, local search based on diffusion gives good results as well [26]. This approach has the nice side effect of optimizing the shape of the cells, but it requires an embedding of the graph. Most other methods, including ours, do not.

The remainder of this paper is organized as follows. We explain the two phases of our algorithm in Sections 2 (filtering) and 3 (assembly). Section 4 shows how to find balanced partitions with PUNCH. In Section 5 we present extensive experiments. Section 6 contains concluding remarks.

Preliminaries. The input to the partitioning problem is an undirected graph $G = (V, E)$. Each vertex $v \in V$ has a positive *size* $s(v)$, and each edge $e = \{u, v\} \in E$ has a positive *weight* $w(e)$ (or, equivalently, $w\{u, v\}$). We are also given a *cell size bound* U. We assume G is simple (without loss of generality) and connected (since we can process each connected component independently).

By extension, the size $s(C)$ of a set $C \subseteq V$ is the sum of the sizes of its vertices, and the weight of a set $F \subseteq E$ is the sum of the weights of its edges. A partition $P = \{V_1, V_2, \ldots, V_k\}$ of V is a set of disjoint subsets (also called *cells*) such that $\cup_{i=1}^{k} V_i = V$. Any edge $\{u, v\}$ with $u \in V_i$ and $v \notin V_i$ is called a *cut edge*. Given a set $S \subseteq V$, let $\delta(S) = \{\{u, v\} : \{u, v\} \in E, u \in S, v \notin S\}$ be the set of edges with exactly one endpoint in S. The set of edges between cells in a partition P is denoted by $\delta(P)$. The *cost* of P is the sum of the weights of its cut edges, i.e., $\mathit{cost}(P) = w(\delta(P))$.

The goal of the *graph partitioning problem* is to find a minimum-cost partition P such that the size of each cell is bounded by U. This problem is NP-hard [13].

2 Filtering Phase

The goal of the *filtering phase* of our algorithm is to reduce the size of the input graph while preserving its sparse cuts. The phase detects and contracts relatively dense areas separated by small cuts. The edges in these cuts are preserved, while all other edges are contracted.

To *contract* vertices u and v, we replace them by a new vertex x with $s(x) = s(u) + s(v)$. Also, for each edge $\{u, z\}$ or $\{v, z\}$ (with $z \notin \{u, v\}$) we create a new edge $\{x, z\}$ with the same weight. If multiple edges are created (which happens when u and v share a neighbor), we merge them and combine their weights. By extension, *contracting a set of vertices* means repeatedly contracting pairs of vertices in the set (in any order) until a single vertex remains. Similarly, contracting an edge means contracting its endpoints.

The filtering phase has two stages. The first finds *tiny cuts*, i.e., cuts with at most two edges. The second stage applies a randomized heuristic to find *natural cuts*, arbitrary cuts that are small relative to the neighboring areas of the graph. We discuss each stage in turn.

Detecting Tiny Cuts. The first stage starts with the original graph and gradually contracts some of its vertices. It consists of three passes.

The first pass uses depth-first search to identify all biconnected components of the graph. They form a tree T. We make T rooted by picking as a root the maximum-size edge-connected component, which on road networks typically corresponds to most of the graph. We then traverse T in top-down fashion. As soon as we enter a subtree S of total size at most U, we contract it into a single vertex. Note that this does not affect the optimum solution value: any solution that splits S in more than one cell can be converted (with no increase in cost) into one in which S defines a cell on its own.

To shrink the instance even further, we merge the newly-contracted vertex with its neighbor in the parent component, as long as (1) the subtree has size at most τ (a pre-determined threshold) and (2) the resulting merged vertex has size at most U. (We use $\tau = 5$ in our experiments.) Unlike the previous contraction rule, this one is heuristic: we may lose optimality. This is also the case with most of the reductions that follow.

During the second pass, we identify all vertices of degree 2. We contract each path they induce to a single vertex, unless its total size exceeds U.

The third pass, in which we process 2-cuts (cuts with exactly 2 edges), is more elaborate. In principle there could be $\Omega(m^2)$ such cuts, but it is easy to see that the following predicate $P \subseteq E \times E$ is an equivalence relation: $(e, f) \in P \leftrightarrow e = f$ or e and f form a 2-cut, but neither e nor f form a 1-cut. We identify these equivalence classes in linear time using the (quite elegant) algorithm of Pritchard and Thurimella [29]. We then process the equivalence classes one by one.

To process a class $S \subseteq E$, we first compute the connected components of the graph $G_S = (V, E \setminus S)$, then contract every component whose size is at most U.

Note that we cannot afford to look at $\Theta(|V|)$ vertices to process each equivalence class, since there are too many of them. We get around this by always traversing two components at a time. Initially, we take an arbitrary edge of the equivalence class and start traversing the two components containing its endpoints. Whenever we finish traversing one component, we start visiting the next one in the cycle. After $k-1$ components are visited in full (where k is the number of components), we stop. At this point, only the largest component, which typically contains almost the entire graph, has not been visited in full. In total, to process the equivalence class we visit no more than twice the number of vertices in the smaller components.

Detecting Natural Cuts. The second stage of the filtering phase of PUNCH detects *natural cuts* in the graph. Unlike the cuts in the previous section, they do not have a preset number of edges. Intuitively, a natural cut is a sparse cut separating a local region from the rest of the graph. Our algorithm finds natural cuts throughout the graph, ensuring that every vertex is inside some such cut.

It is tempting to look for a good cut by picking two vertices (s and t) within a local region and computing the minimum cut between them. Unfortunately, since the average degree on a road network is very small (lower than 3), such s–t cuts are usually trivial, with either s or t alone in its component.

Alternatively, one could try a more complicated procedure, such as determining the sparsest cut of some region R, i.e., the cut $C \subseteq R$ minimizing

$w(\delta(C))/(s(C) \cdot s(R \setminus C)))$. This could be useful, but finding such a cut is NP-hard, and practical approximation algorithms are not known [3].

By computing a minimum cut between *sets* of vertices, we get a notion of natural cuts that is both useful and tractable. These cuts are nontrivial and can be computed by a standard s–t cut algorithm, such as the push-relabel method [16].

Our algorithm works in iterations. Each iteration picks a vertex v as a *center* and grows a breadth-first search (BFS) tree T from v, stopping when $s(T)$ (the sum of its vertex sizes) reaches αU, for some parameter $0 < \alpha \leq 1$. We call the set of neighbors of T in $V \setminus T$ the *ring of* v. The *core of* v is the union of all vertices added to T before its size reached $\alpha U/f$, where $f > 1$ is a second parameter. (In our experiments, we use $\alpha = 1$ and $f = 10$ as default.) We temporarily contract the core to a single vertex s and the ring into a single vertex t and compute the minimum s–t cut between them (using $w(.)$ as capacities), as shown in Figure 1.

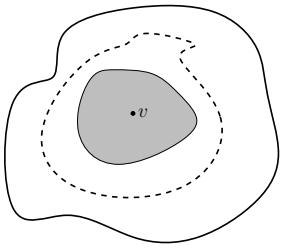

Fig. 1. Computing a natural cut. A BFS tree of size at most αU is grown from a center vertex v. The external neighboring vertices of this tree are the *ring* (outer solid line). The set of all vertices visited by the BFS while the tree had size less than $\alpha U/f$ is the *core* (gray region). The natural cut (dashed line) is the minimum cut between the contracted versions of the core and the ring.

To pick the centers in each iteration, we need a rule that ensures that every vertex eventually belongs to at least one core, and is therefore inside at least one cut. We accomplish this by picking v uniformly at random among all vertices that have not yet been part of any core. The process stops when there are no such vertices left. Note that we can repeat this entire procedure $\mathcal{C}$ times in order to increase the number of marked edges, where $\mathcal{C}$ (the *coverage*) is a user-defined parameter.

When these iterations finish, we contract each connected component of the graph $G_C = (V, E \setminus C)$, where C is the union of all edges cut by the process above. We call each contracted component a *fragment*. Figure 2 gives an example.

Note that setting $\alpha \leq 1$ ensures that no fragment in the contracted graph has size greater than U. The transformed problem can therefore still be partitioned

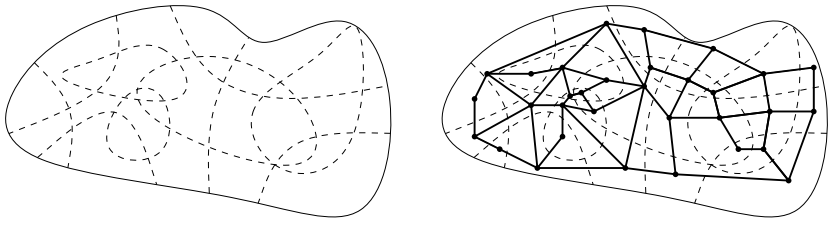

Fig. 2. During the filtering phase, several natural cuts are detected in the input graph (left). At the end of this phase, any edge not contributing to a cut is contracted. Each vertex of the resulting graph (right) represents a *fragment*.

into cells of size at most U, and any such partition can be transformed into a feasible solution to the original instance.

Finally, we note that the generation of natural cuts can be easily parallelized. Our implementation first picks all centers sequentially, then runs each minimum-cut computation (including the creation of the relevant subproblem) in parallel.

3 Assembly Phase

During the assembly phase, we finally find a partition. We take as input the graph produced by the filtering phase. Although this is the graph of fragments (not the original input), we also refer to it as $G = (V, E)$ in this section to simplify notation. Any valid partition of this input corresponds to a valid partition of the original graph, with the same cost. To obtain good partitions, the assembly phase uses several tools: a greedy algorithm, a local search, and a multistart heuristic with combination. We discuss each in turn.

Greedy Algorithm. We use a randomized greedy algorithm to find a reasonable initial partition. It repeatedly contracts pairs of adjacent vertices, and stops when no new contraction can be performed without violating the size constraint. Each step picks, among all pairs of adjacent vertices with combined size at most U, the pair $\{u, v\}$ that minimizes a *score* function. This function is randomized and depends on the sizes of both vertices and on the weight of the edge between them. We tried many score functions and settled for $score(\{u, v\}) = r \cdot w\{u, v\} \cdot (\sqrt{1/s(u)} + \sqrt{1/s(v)})$, where r is a random number between 0 and 1.

Intuitively, we want to merge vertices that are relatively small but tightly connected. The precise formula is based on the observation that, on road networks, we expect a region of size k to have about $O(\sqrt{k})$ outgoing edges. Moreover, by adding two independent fractions we implicitly give higher importance to the smaller region. Different score functions may work better for other graph classes.

The randomization term (r) is relevant for the local search and the multistart heuristic, as we shall see. It is biased towards 1 to ensure that the contribution of the deterministic term is not too small. More precisely, we use two constants a and b, both between 0 and 1. With probability a, we pick r uniformly at random in the range $[0, b]$; with probability $1 - a$, we pick r uniformly at random from $[b, 1]$. After some parameter testing, we ended up using $a = 0.03$ and $b = 0.6$.

For a fixed pair of vertices, the score function is computed once and stored. After a contraction, it is recomputed (with fresh randomization terms) for all edges incident to the contracted vertex.

Local Search. Greedy solutions may be reasonable, but they can be greatly improved by *local search*. The local search views the current partition as a *contracted graph* H. Each vertex of H corresponds to a cell of the partition, and there is an edge $\{R, S\}$ in H between cells R and S if there is at least one edge $\{u, v\}$ in G with $u \in R$ and $v \in S$. As usual, the *weight* of $\{R, S\}$ in H is the sum of the weights of corresponding edges in G.

We tried several variants of the local search, all of them consisting of a sequence of *reoptimization steps*. Each such step first creates an auxiliary instance $G' = (V', E')$ consisting of a connected subset of cells of the current partition. In this auxiliary instance, some of the original cells are *uncontracted* (i.e., decomposed into their constituent *fragments* in G), while others remain contracted. The weight of an edge in E' is given by the sum of the weights of the corresponding edges in G.

We run the randomized greedy algorithm on G', and use the result to build the corresponding modified solution H' (of G) in a natural way. If H' is better than H, we make H' our new current solution, replacing H. Otherwise, we say that this reoptimization step *failed*, and keep H as the current solution.

We tested three local searches, which differ in how they build the auxiliary instances G', as Figure 3 illustrates. The simplest variant picks, in each step, a pair $\{R, S\}$ of adjacent cells and creates an auxiliary instance G'_{RS} consisting of the uncontracted versions of R and S. We call this variant $\mathcal{L}_2$. The second variant, $\mathcal{L}_2^+$, is similar, but also includes in G'_{RS} the (contracted) neighbors of R and S in H. The third variant, $\mathcal{L}_2^*$, extends $\mathcal{L}_2^+$ by also uncontracting the neighbors of R and S.

For all variants, each step is fully determined by a pair $\{R, S\}$ of cells. The reoptimization step itself, however, is heuristic and randomized. In practice, it is worth repeating it multiple times for the same pair $\{R, S\}$. We maintain for each pair $\{R, S\}$ of adjacent cells a counter φ_{RS}. Initially set to zero, it roughly measures the number of unsuccessful reoptimization steps applied to $\{R, S\}$. If a reoptimization step on $\{R, S\}$ fails, we increment φ_{RS}. If it succeeds, we reset the counters associated with all edges in H' having at least one endpoint in an uncontracted region of G'_{RS}.

Our algorithm uses the φ_{RS} counters and a user-defined parameter $\varphi \geq 1$ to decide when to stop. The parameter limits the maximum number of allowed failures per pair. Among all pairs $\{R, S\}$ with $\varphi_{RS} < \varphi$, we pick one uniformly at random for the next reoptimization step. If no such pair is available, the

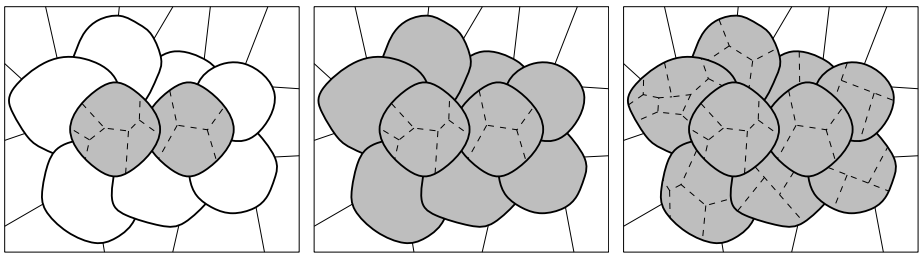

Fig. 3. Local searches $\mathcal{L}_2$, $\mathcal{L}_2^+$, and $\mathcal{L}_2^*$ as defined by the same pair of cells. Search $\mathcal{L}_2$ (left) reoptimizes an auxiliary instance corresponding to the uncontracted versions of the two central cells. $\mathcal{L}_2^+$ (center) also includes the (contracted) neighboring cells in the auxiliary instance. In $\mathcal{L}_2^*$ (right) the neighboring cells are uncontracted as well. (Cells split by dashed lines are uncontracted; cells in the auxiliary instance are shaded.)

algorithm stops. As one would expect, increasing φ leads to better solutions, but slows down the algorithm.

To parallelize the local search, we try several pairs of regions simultaneously and, whenever an improving move is found, we make the corresponding change to the solution sequentially.

Multistart and Combination. We use two strategies to improve the quality of the solutions we find. The first is to run a *multistart* heuristic. In each iteration, it runs our randomized greedy algorithm and applies local search to the resulting solution. Since both the greedy algorithm and the local search are randomized, different iterations may find distinct solutions. After M iterations (where M is an input parameter), the algorithm stops and returns the best solution found.

We can find even better partitions by *combining* pairs of solutions generated during the multistart algorithm. To do so, we keep a pool of *elite solutions* with capacity k, containing some of the best partitions found so far. Here k is a parameter of the algorithm; a reasonable value is $k = \lceil\sqrt{M}\rceil$. This is a standard application of the evolutionary approach, widely used by combinatorial optimization heuristics, including genetic algorithms [17] and path-relinking [14].

In the first k iterations of the multistart algorithm, we simply add the resulting partition P to the pool. Each subsequent iteration also starts by generating a new solution P (using the randomized greedy algorithm and local search), but P is not immediately added to the pool. Instead, we first create another solution P' by combining two distinct solutions picked uniformly at random from the pool. We then combine P and P', obtaining a third solution P''. Finally, we try to insert P'', P', and P into the pool, in this order.

We must still show how to combine two solutions and how to decide whether a new solution should be inserted into the pool. We discuss each issue in turn.

Combination. Let P_1 and P_2 be two partitions. The purpose of combining them is to obtain a third solution P_3 that shares the good features of the original ones. Intuitively, if P_1 and P_2 "agree" that an edge (u, v) is on the boundary between two regions, it should be more likely to be a cut edge in P_3 as well. Our algorithm implements this intuition as follows. First, it creates a new instance G' with the same vertices and edges as G. For each edge e, define $b(e) \in \{0, 1, 2\}$ as the number of solutions (among P_1 and P_2) in which (u, v) is a boundary edge. The weight $w'(e)$ of e in G' is its original weight $w(e)$ in G multiplied by a positive *perturbation factor* $p_{b(e)}$, which depends on $b(e)$. Intuitively, to make P_3 mimic P_1 and P_2 we want $p_0 > p_1 > p_2$, since lower-weight edges are more likely to end up on the boundary. The algorithm is not too sensitive to the exact choice of parameters; our experiments use $p_0 = 5$, $p_1 = 3$ and $p_2 = 2$.

We use the standard combination of constructive algorithm and local search to find a solution of G', which we turn into P_3 (a solution of G) by restoring the original edge weights. Note that the idea of combining solutions by perturbing the input has been used before, both for graph partitioning [35] and other problems [32].

Pool management. The purpose of the pool is to keep good solutions found by the algorithm. While the pool has fewer than k solutions, any request to add a new solution P is granted. If, however, the pool is already full, we must decide whether to actually add P or not and, if so, we must pick a solution to evict. If all solutions already in the pool are better than P, we do nothing. Otherwise, among all solutions that are no better than P, we evict that one that is *most similar* to P. For this purpose, the difference between two solutions is defined as the cardinality of the symmetric difference between their sets of cut edges. This replacement strategy has been shown to make similar evolutionary algorithms more effective by ensuring some diversity in the pool [31].

4 Balanced Partitions

As described, PUNCH solves the standard graph partitioning problem, which has no hard bound on the number of cells in the solution; it only ensures that no cell will have size greater than U. We now show how to use PUNCH to compute balanced partitions. In this variant, the inputs are the number of cells (k) and the tolerated imbalance (ε); the partitioner must find k cells, each with size at most $(1+\varepsilon)\lceil n/k \rceil$, where n is the total number of vertices in the original graph.

To find an ε-balanced partition with at most k cells, we first use the algorithms described so far (a combination of filtering and assembly) to produce a standard (potentially unbalanced) partition with $U = \lfloor (1+\varepsilon)\lceil n/k \rceil \rfloor$. The only constraint the partition may violate is having $\ell > k$ cells. To fix this, we run a *rebalancing* algorithm: we choose a set of k *base cells* and distribute the fragments of the remaining $\ell - k$ cells among the base cells. Like other algorithms for the balanced problem, we may sacrifice cell connectivity.

More precisely, to select the base cells, each cell C of the initial solution is assigned the score $(2+r)s(C)$, where r is picked uniformly at random between 0 and 1; the k cells with highest score are chosen. Let $V_1, V_2, \ldots, V_k$ denote the base cells, and let W be the set of fragments of the remaining cells. Then, we start an iterative process. In each round, we set $U' = \max_{1 \leq i \leq k}(U - s(V_i))$ and find a partition P' of $G[W]$ (the subgraph induced by W) with U' as an upper bound on the cell size. We then heuristically merge cells of P' with base cells in the following manner. We process the cells of P' in decreasing order of size. Take a cell $C \in P'$. Among all base cells V_i with $s(V_i) + s(C) \leq U$, we pick one at random with probability proportional to $1/s(V_i)$, thus favoring tighter fits. If C does not fit anywhere (no base cell is small enough), we skip it: C will be split in the next round. If all cells of P' can be thus allocated, we are done. Otherwise, we proceed to the next round by decreasing U' (taking the modified base cells into account) and finding a new partition P'.

Because the rebalancing algorithm is randomized (and relatively quick), we run it several times to rebalance a single initial solution, and pick the best result.

One problem remains: our approach may fail if the fragments built during the filtering phase are too big. Especially when ε is very small, it may happen that we cannot rebalance the partition. To make this less likely, when computing

balanced partitions we actually use $U/3$ during the initial filtering stage, thus creating smaller fragments. If the rebalancing procedure still fails, we could reduce the threshold during filtering even further and start all over again. For the inputs tested, however, setting the threshold to $U/3$ is sufficient.

5 Experiments

We implemented PUNCH in C++ and compiled it with Microsoft Visual C++ 2010. For parallelization, we use OpenMP. The evaluation was conducted on a machine equipped with two Intel Xeon X5680 processors and 96 GB of DDR3-1333 RAM, running Windows 2008R2 Server. Each CPU has 6 cores clocked at 3.33 GHz, with 6 x 64 KB L1, 6 x 256 KB L2, and 12 MB L3 cache.

We use two graphs in our main experiment, both taken from the webpage of the 9th DIMACS Implementation Challenge [11]. The *Europe* instance represents the road network of Western Europe, with 18 million vertices and 22.5 million edges, and was made available by PTV AG [30]. The *USA* road network (generated from TIGER/Line data [36]) has 24 million vertices and 29.1 million edges. In all cases we use an *undirected* and *unweighted* variant of the graphs. Note that the USA data is already undirected. For Europe, this means interpreting all input arcs as undirected and eliminating all parallel edges: If arcs (v, w) and (w, v) are in the input, we have a single edge $\{v, w\} = \{w, v\}$.

We implemented the push-relabel algorithm of Goldberg and Tarjan [16] to compute s–t cuts. For our application, the version using FIFO order, frequent global relabelings, and the *send* operation performs best (see [16] for details).

We use the following parameters for PUNCH. The filtering phase uses $\alpha = 1$, $f = 10$, and $\mathcal{C} = 2$, and detects both tiny and natural cuts. The assembly phase uses the $\mathcal{L}_2^+$ local search with $\varphi = 16$. We do not use the combination heuristic by default. (A detailed study of the effects of these parameters can be found in the full paper [8].) We use all 12 cores during natural-cut detection and the assembly phase; our implementation of tiny-cut detection is sequential.

Unbalanced Partitions. Table 1 shows how PUNCH performs on Europe and USA when the maximum cell size U varies from 2^{10} to 2^{22}. It reports the average number of cells, the average number $|V'|$ of fragments after filtering, the solution value (number of cut edges), and the average running time of PUNCH (in total and of each part). Since our algorithm is nondeterministic due to parallelism and randomness, all values are aggregated over 50 runs, with varying random seeds.

As expected, the filtering phase reduces the graph size significantly. The tiny-cut procedure eliminates about half the vertices, while the natural-cut routine further decreases the number of vertices by 1 to 4 orders of magnitude, depending on U. Because the filtering phase grows BFS trees parameterized by U, more edges are marked as candidates (and kept uncontracted) when U is small.

This dependence also explains our running times. The procedure for detecting tiny cuts, which is not parallelized, is almost independent of U and takes about 30 seconds. Natural-cut detection is executed on bigger subgraphs as U increases.

Table 1. Performance of 50 runs of PUNCH on Europe and USA, with varying maximum cell sizes (U). Under "cells", we report the lower bound (LB = $\lceil n/U \rceil$) and the average number of cells in the actual solution. Column $|V'|$ refers to the average number of vertices (fragments) after filtering. "solution" reports the average and best solutions found. Finally, the average running times of each phase of the algorithm (tiny cuts, natural cuts, and assembly) and in total are shown.

		CELLS			SOLUTION			TIME [S]			
GRAPH	U	LB	AVG	$\lvert V' \rvert$	BEST	AVG	WORST	TNY	NAT	ASM	TOTAL
Europe	1024	17589	20128.7	1366070	168463	168767	169098	24.5	17.5	37.6	79.7
	4096	4398	5000.4	605864	68782	69034	69290	24.5	18.2	19.9	62.5
	16384	1100	1247.5	258844	28279	28448	28604	24.5	27.1	10.1	61.6
	65536	275	313.9	104410	11257	11403	11518	24.4	51.3	4.8	80.5
	262144	69	80.9	34768	4124	4194	4268	24.4	80.0	1.7	106.1
	1048576	18	21.8	10045	1422	1464	1527	24.4	122.9	0.6	147.9
	4194304	5	5.8	2014	369	371	376	24.2	172.2	0.3	196.6
USA	1024	23387	26725.2	1826293	222349	222636	222896	33.1	21.1	50.3	104.6
	4096	5847	6642.6	787382	87584	87762	87949	33.1	21.3	25.5	79.9
	16384	1462	1661.2	293206	34175	34345	34523	33.1	30.6	11.3	75.0
	65536	366	417.7	89762	12627	12767	12906	33.1	53.1	3.7	89.9
	262144	92	108.6	22728	4506	4556	4616	33.1	69.2	1.0	103.3
	1048576	23	27.4	4615	1415	1504	1607	33.1	84.3	0.3	117.6
	4194304	6	7.0	931	381	383	389	33.1	105.3	0.3	138.7

Still, the total time spent on it increases only by one order of magnitude as U increases by more than three (from 2^{10} to 2^{22}). The reason is that the number of min-cut computations decreases as U increases. Conversely, the assembly phase gets faster as U increases because it operates on smaller graphs. For very small values of U, the assembly phase is the main bottleneck. In total, we need between 1 and 3 minutes to find good partitions of Europe or the USA.

We note there are differences between the two graphs. When U is large, the contracted version of USA has less than half as many vertices as the corresponding graph of Europe, even though Europe is 25% smaller than USA before contraction. This indicates the USA network has more obvious natural cuts at a more global scale. (This could be—at least partially—an artifact of this particular data set; as observed on the DIMACS webpage [11], several important road segments, including some on bridges and freeways, are missing from the USA graph.) The difference is much less pronounced for smaller values of U.

Although randomized, PUNCH is quite robust: over 50 executions, the best and worst solutions found are close to average. Moreover, the solutions found, although not perfectly balanced, are not too far from it. On average, PUNCH finds solutions with about 15% more cells than a perfectly balanced partition.

Balanced Partitions. We now consider *balanced* partitions, where the maximum number k of cells is bounded and each cell must have size at most $U^* = \lfloor (1 + \varepsilon)\lceil n/k \rceil \rfloor$, where ε is the tolerated imbalance. We use $\varepsilon = 0.03$, as is common in

Table 2. Best solution quality of strong PUNCH when finding balanced partitions with $\varepsilon = 0.03$ for varying k, aggregated over 9 runs. Columns $|V|$ and $|E|$ report the number of vertices and edges of the instance.

			BEST SOLUTION					
INSTANCE	$\|V\|$	$\|E\|$	2	4	8	16	32	64
luxembourg	114599	119666	16	46	79	139	235	369
belgium	1441295	1549970	70	161	308	532	880	1401
netherlands	2216688	2441238	40	81	191	360	652	1186
italy	6686493	7013978	36	89	198	338	665	1166
great-britain	7733822	8156517	82	213	377	633	1118	1796
germany	11548845	12369181	108	276	485	845	1475	2282
asia	11950757	12711603	7	20	47	110	238	452
europe	50912018	54054660	138	311	515	905	1488	2509

Table 3. Performance of default PUNCH when finding balanced partitions with $\varepsilon = 0.03$ for varying k, aggregated over 9 runs.

	MEDIAN SOLUTION						AVERAGE TIME [S]					
INSTANCE	2	4	8	16	32	64	2	4	8	16	32	64
luxembourg	16	46	82	148	245	377	1.2	2.4	2.4	1.9	1.5	2.2
belgium	72	167	316	565	923	1436	16.0	19.9	20.8	20.4	15.7	18.3
netherlands	40	81	191	380	679	1210	28.1	17.1	15.2	15.0	12.1	16.9
italy	36	91	201	349	690	1187	97.8	78.6	65.0	51.9	41.7	40.0
great-britain	84	225	393	638	1175	1846	60.4	60.6	57.7	50.8	43.6	47.6
germany	113	283	509	881	1512	2332	128.6	125.8	104.7	91.5	74.3	76.4
asia	7	20	48	112	249	470	67.5	76.6	60.1	50.9	46.1	43.7
europe	140	312	523	955	1536	2576	1051.0	814.0	627.4	512.8	427.7	375.0

Table 4. Performance of strong PUNCH when finding balanced partitions with $\varepsilon = 0.03$ for varying k, aggregated over 9 runs.

	MEDIAN SOLUTION						AVERAGE TIME [S]					
INSTANCE	2	4	8	16	32	64	2	4	8	16	32	64
luxembourg	16	46	80	142	238	377	7.2	16.4	18.1	13.7	11.1	8.6
belgium	71	163	313	548	900	1421	51.2	99.9	113.6	115.0	94.9	58.5
netherlands	40	81	191	369	662	1199	132.2	57.3	52.8	59.2	50.1	48.4
italy	36	90	200	339	673	1175	157.2	173.8	174.3	135.1	110.2	80.7
great-britain	83	220	381	636	1140	1821	103.6	165.5	189.8	167.0	135.3	108.5
germany	111	279	503	852	1488	2317	195.6	347.7	291.8	253.9	214.1	153.0
asia	7	20	48	111	242	462	83.4	200.0	95.3	73.7	66.4	58.4
europe	139	311	522	923	1517	2538	2217.9	1451.8	939.8	732.5	604.0	494.6

the literature [4]. As described in Section 4, to find a balanced partition with PUNCH we first run the filtering stage with $U = U^*/3$. We could then run the assembly stage with $U = U^*$ to generate an initial unbalanced solution, which we make balanced by reassigning some fragments. In practice, the variance of

the assembly phase is rather large when k is very small (2 or 4), which we can remedy by running it several times.

More precisely, our default algorithm for finding a balanced partition is as follows: (1) run the filtering stage once with $U = U^*/3$; (2) use the multistart algorithm to create $\lceil 32/k \rceil$ (unbalanced) solutions with $U = U^*$; (3) rebalance each unbalanced solution 50 times (as described in Section 4); (4) return the best balanced solution thus found. We use $\varphi = 512$ when finding unbalanced solutions (step 2), and $\varphi = 128$ during rebalancing (step 3). We also consider a *strong* version of balanced PUNCH. The only difference from the default version is that it creates $\lceil 256/k \rceil$ (unbalanced) solutions, instead of $\lceil 32/k \rceil$.

Table 2 reports the best solutions found by strong PUNCH on the road networks (streets) of the 10th DIMACS challenge [4]. Tables 3 and 4 report, for each variant, the median solution over 9 runs and average running times.

We observe that the strong version of PUNCH yields slightly better results but takes more time. However, the gain in solution quality is relatively small. We also observe that the median and best solutions (comparing Tables 4 and 2) are very close to each other. We conclude that PUNCH is also very robust for finding balanced partitions.

6 Conclusion

We presented PUNCH, a new algorithm for graph partitioning that works particularly well on road networks. The key feature of PUNCH is its graph reduction routine: By identifying natural cuts and contracting dense regions, it can reduce the input size by orders of magnitude, while preserving the natural structure of the graph. Because of this efficient reduction in size, we can run more time-consuming routines to assemble a good partition. As a result, we obtain partitions for road networks better than most previous approaches. Only Buffoon [33] sometimes finds better partitions than PUNCH, and it does so by using our filtering phase and running KaFFPaE on the fragment graph. The resulting partitions are sometimes slightly better than those found by PUNCH, but for the price of much higher running times. Altogether, PUNCH provides a good trade-off between solution quality and running times: It needs only a few minutes to generate an excellent partition, which is fast enough for most applications.

References

1. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *SEA*, LNCS 6630, pp. 230–241. Springer, 2011.
2. R. Andersen and K. J. Lang. An Algorithm for Improving Graph Partitions. In *SODA*, pp. 651–660, 2008.
3. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, and A. Marchetti-Spaccamela. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, second edition, 2002.

4. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, 2011. `http://www.cc.gatech.edu/dimacs10/index.shtml`.
5. 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. Y. Boykov, O. Veksler, and R. Zabih. Fast Approximate Energy Minimization via Graph Cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:1222–1239, 2001.
7. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *SEA*, LNCS 6630, pp. 376–387. Springer, 2011.
8. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *IPDPS*. IEEE Computer Society, 2011.
9. D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [11], pp. 73–92.
10. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. Zweig, editors, *Algorithmics of Large and Complex Networks*, LNCS 5515, pp. 117–139. Springer, 2009.
11. 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.
12. C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC*, pp. 175–181, 1982.
13. M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, 1979.
14. F. Glover. Tabu search and adaptive memory programming: Advances, applications and challenges. In R. S. Barr, R. V. Helgason, and J. L. Kennington, editors, *Interfaces in Computer Science and Operations Research*, pp. 1–75. Kluwer, 1996.
15. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [11], pp. 93–139.
16. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *Journal the ACM*, 35:921–940, 1988.
17. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
18. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *SC*, p. 28. ACM Press, 1995.
19. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [11], pp. 41–72.
20. G. Karypis and G. Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.
21. B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.
22. T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In *SEA*, LNCS 6049. Springer, May 2010.
23. 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(3):277–286, 2003.
24. 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*, pp. 6–6, New York, NY, USA, 2009. ACM.

25. J. Maue, P. Sanders, and D. Matijevic. Goal-Directed Shortest-Path Queries Using Precomputed Cluster Distances. *ACM Journal of Experimental Algorithmics*, 14:3.2:1–3.2:27, 2009.
26. 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.
27. 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.
28. R. Preis and R. Diekmann. The PARTY Partitioning Library, User Guide . Technical report, University of Paderborn, Germany, 1996. tr-rsfb-96-02.
29. D. Pritchard and R. Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transaction on Algorithms*, 7:46:1–46:30, 2011.
30. PTV AG - Planung Transport Verkehr. `http://www.ptv.de`, 1979.
31. M. G. C. Resende and R. F. Werneck. A hybrid heuristic for the p-median problem. *Journal of Heuristics*, 10(1):59–88, 2004.
32. C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS Journal on Computing*, 14(3):228–246, 2002.
33. P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *ALENEX*, pp. 16–29. SIAM, 2012.
34. K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High-Performance Scientific Simulations. In J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors, *Sourcebook of Parallel Computing*, pp. 491–541. Morgan Kaufmann, 2003.
35. 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.
36. D. US Census Bureau, Washington. UA Census 2000 TIGER/Line files. http://www.census.gov/geo/www/tiger/tigerua/ua.tgr2k.html, 2002.
37. 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.
38. 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.