Graph Partitioning with Natural Cuts

Daniel Delling Andrew V. Goldberg Ilya Razenshteyn[1] Renato F. Werneck

Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA

December 2010

Technical Report
MSR-TR-2010-164

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 significantly, 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 the best known partitions for continental-sized networks, significantly improving on previous results.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1] *Mathematics Department, Logic and Algorithms Theory Division Lomonosov Moscow State University, Russia.* This work was done while the author was visiting Microsoft Research.

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 [3], computer vision [20], image analysis [36], distributed computing [21], and route planning [6]. Most variants of this problem are known to be NP-hard [9] and focus on minimizing the number of edges linking vertices from different cells, the *cut size*. Given its importance, there is a rich literature on the problem, including a wealth of heuristic solutions (see [32, 35] for comprehensive 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 [26], METIS [17], DiBaP [23], JOSTLE [35], CHACO [14], PARTY [27], KaPPa [16], and KaSPar [24].

Although MGP approaches can be used for road networks, they do not exploit the natural properties of such networks in full. In particular, road networks are not uniform: there are densely populated regions (which should not be split) 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, KaPPa, KaSPaR) 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 [15, 22, 5], distribution of data [19], and computation of centrality measures [11].

In this paper, we introduce 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 an input 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. (See Figure 1 for an example.) Our algorithm runs in two phases: *filtering* and *assembly*.

The filtering phase aims to reduce significantly the size of the graph 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 pairs of random 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, reducing the graph size by up to three orders of magnitude. Despite being much smaller, the 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 instance, our algorithm can afford to 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 vertices). In contrast with existing partitioners, we do not disassemble individual fragments.

Note that our focus is 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

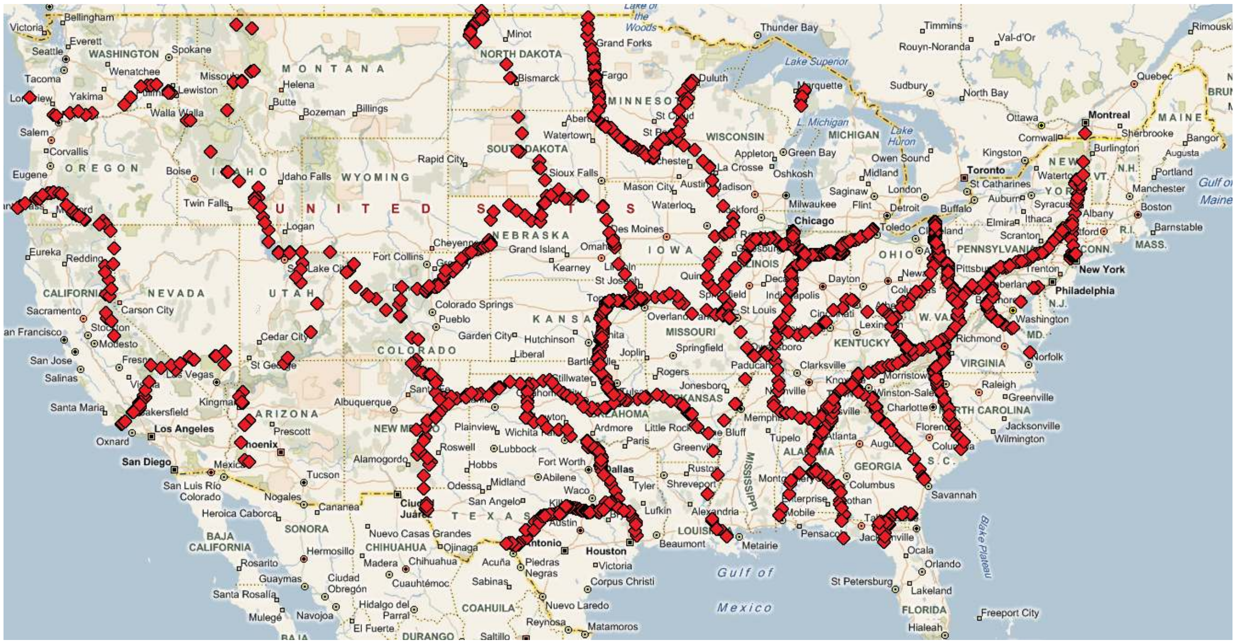

Figure 1: Example for a partition generated by PUNCH. The input is the road network of the US, taken from [7], having 24 million nodes and 29.1 million edges. We set U (the maximum cell size) to 1 million. The number of resulting cut edges is 1493 and the number of cells obtained is 30.

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. Our comparison shows that PUNCH significantly improves the best previous bounds for road networks.

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 and KL heuristics [8, 18]. 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 [1] 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 [4]. Besides vertex swapping and minimum cuts, local search based on diffusion gives good results as well [23]. 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. Section 2 gives basic definitions and notation. We explain the two phases of our algorithm in Sections 3 (filtering) and 4 (assembly). Section 5 shows how to find balanced partitions with PUNCH. In Section 6 we present extensive experiments. Section 7 contains concluding remarks.

2 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. Without loss of generality, we assume that G is simple. To simplify the description of our algorithm, we assume G is connected, since we can process each connected component independently.

By extension, the size $s(C)$ of any 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., $cost(P) = w(\delta(P))$.

The *graph partitioning problem* is to find a mimimum-cost partition $P = \{V_1, V_2, \ldots, V_k\}$ such that the size of each cell is bounded by U. This problem is NP-hard [9].

3 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 other all edges are contracted.

To *contract* vertices u and v, we replace these two vertices with a new vertex x with $s(x) = s(u) + s(v)$. Moreover, 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.

3.1 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 identifies all edge-connected components of the graph using depth-first search. They form a tree T. We make T rooted by choosing as a root the edge-connected component with maximum size, which on road networks typically corresponds to most of the graph. We then traverse T in a top-down fashion. As soon as we enter a subtree S of total size at most U, we contract it into a single vertex, as Figure 2 illustrates. It easy to see 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.)

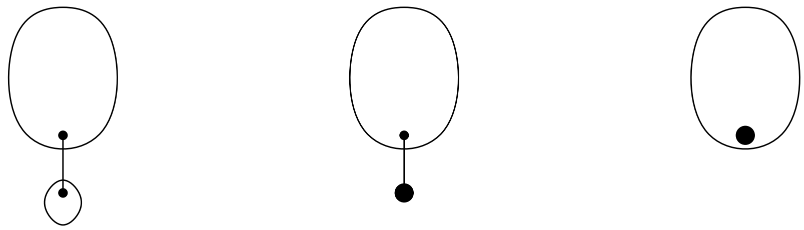

Figure 2: Contracting one-cuts. If there is a one-cut separating a small component from the rest of the graph (left), we contract the small component into a single vertex (center). If the combined size of this vertex and its neighbor is sufficiently small, we contract them again (right).

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. They form paths, which we contract to a single vertex unless their total size is bigger than 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:

$$\begin{aligned}(e, f) \in P \leftrightarrow e = f \text{ or } e \text{ and } f \text{ form a 2-cut,}\\ \text{but neither } e \text{ nor } f \text{ form a 1-cut.}\end{aligned}$$

We use the (quite elegant) algorithm of Pritchard and Thurimella [28] to identify these equivalence classes in linear time. 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. See Figure 3.

Note that we cannot afford to spend $O(|V|)$ time 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, will not be fully visited. In total, to process the equivalence class we visit no more than twice the number of vertices in the smaller components.

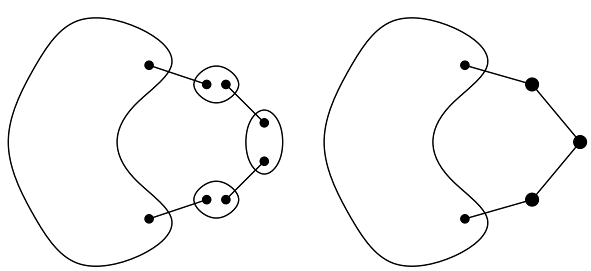

Figure 3: Contracting 2-cuts. The set of edges in the same equivalence class determines a set of *components*. We contract each component of size at most U into a single vertex.

3.2 Detecting Natural Cuts

The second stage of the filtering phase of our algorithm detects *natural cuts* in the graph. Unlike the cuts in the previous section, natural cuts 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 [2].

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 [12].

Our algorithm works in iterations. In each iteration, it 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 4.

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 procedure $\mathcal{C}$ times in order to increase the number of marked edges, where $\mathcal{C}$ (the *coverage*) is a user-defined parameter.

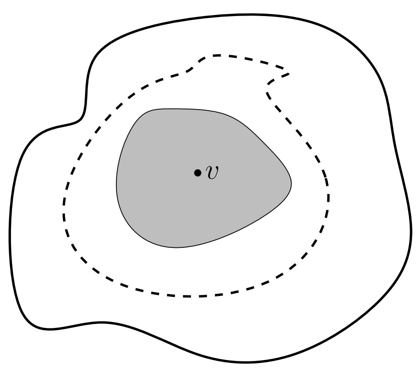

Figure 4: Computing a natural cut. A BFS tree of size less than αU is grown from a center vertex v. The external neighboring vertices of this tree are the *ring* (solid line). The set of all vertices visited by the BFS while the tree had size less than $\alpha U/f$ is the *core* (gray region). The natural cut (dashed line) is the minimum cut between the contracted versions of the core and the ring.

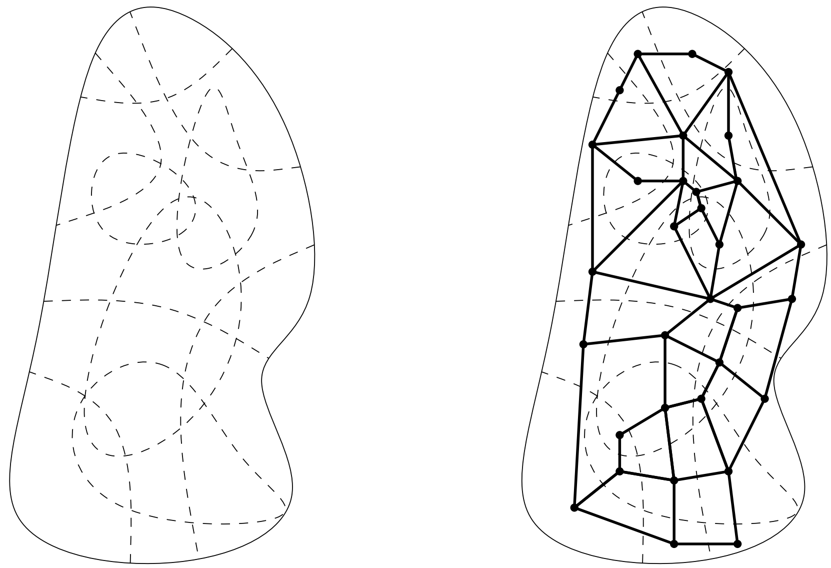

Figure 5: During the filtering phase, several natural cuts are detected in the input graph (left). At the end of this phase, any edge not contributing to a cut is contracted. Each vertex of the resulting graph (right) represents a *fragment*.

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 5 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 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 parallelized easily. In our implementation, we first pick all centers sequentially (for simplicity), and then run each minimum-cut computation (including the creation of the appropriate subproblem) in parallel.

4 Assembly Phase

During the assembly phase, we finally find a partition. We take the graph produced by the filtering phase as input. To simplify notation, in this section we use $G = (V, E)$ to refer to this filtered instance (which is actually the graph of fragments, not the original graph). 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 different tools: a greedy algorithm, a local search, and a multistart heuristic with combination. We discuss each in turn.

4.1 Greedy Algorithm

To find a reasonably good initial partition, we use a *greedy algorithm*. It repeatedly contracts pairs of adjacent vertices, and stops when no new contraction can be performed without violating the size constraint.

In each step, the algorithm picks, among all pairs of adjacent vertices with combined size at most U, the pair $\{u, v\}$ that minimizes a certain *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 several score functions and settled for

$$score(\{u,v\}) = r \cdot \left(\frac{w\{u,v\}}{\sqrt{s(u)}} + \frac{w\{u,v\}}{\sqrt{s(v)}} \right),$$

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 classes of graphs.

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 in most cases 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 each contraction, the function is recomputed (with fresh randomization terms) for all edges adjacent to the contracted vertex.

4.2 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 then run the randomized greedy algorithm (described above) on G'. The solution on G' can then be used to create the modified solution H' 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 the original solution H.

We tested three local searches, which differ in how they build the auxiliary instances G', as Figure 6 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)

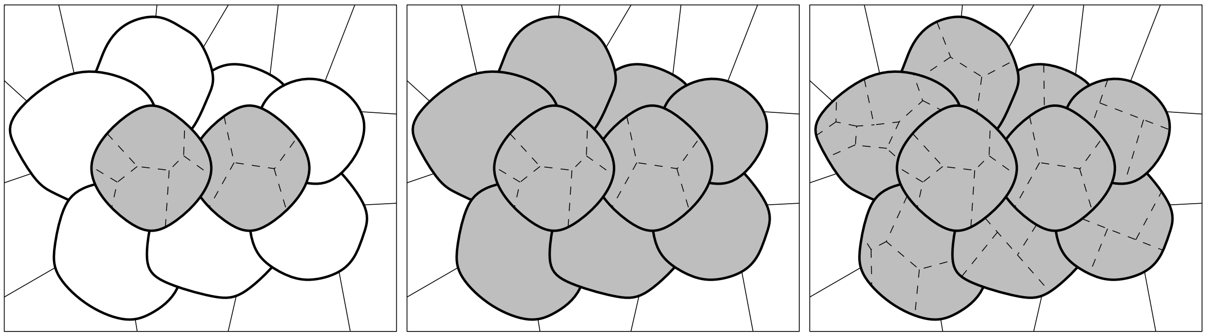

Figure 6: Local searches $\mathcal{L}_2$, $\mathcal{L}_2^+$, and $\mathcal{L}_2^*$ as defined by the same pair of cells. Search $\mathcal{L}_2$ (left) reoptimizes an auxiliary instance corresponding to the uncontracted versions of the two central cells. $\mathcal{L}_2^+$ (center) also includes the (contracted) neighboring cells in the auxiliary instance. Finally, in $\mathcal{L}_2^*$ (right) the neighboring cells are uncontracted as well. (In the picutre, cells subdivided by dashed lines are uncontracted. Cells in the auxiliary instance are shaded.)

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, which 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, together with 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 $\varphi_{RS} < \varphi$, the algorithm picks one uniformly at random for the next reoptimization step. If no such edge is available, the algorithm stops. As one would expect, increasing φ leads to better solutions, but slows down the algorithm.

Our implementation of the local search algorithm is parallelized in a straightforward way. We try several pairs of regions simultaneously and, whenever an improving move is found, we make the corresponding change to the solution sequentially.

4.3 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 the greedy algorithm from Section 4.1 and applies local search to the resulting solution. Since both the greedy algorithm and the local search are randomized, different iterations tend to 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, representing 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 in combinatorial optimization heuristics, including genetic algorithms [13] and path-relinking [10].

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 perform a few additional steps. First, we 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 still have to describe how to combine two solutions and how to decide whether a new solution should be inserted into the pool or not. We discuss each issue in turn.

4.3.1 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 boundary 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 $b(e)$ as the number of solutions (among P_1 and P_2) in which (u, v) is a boundary edge. Clearly, the only possible values of $b(e)$ are 0, 1, and 2. 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 mimick 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 greedy 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 applying a greedy algorithm and local search to a perturbed version of original instance has been used before (see [31], for example).

4.3.2 Pool management

The purpose of the pool of solutions 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 [30].

4.4 Local Branch and Bound

Recall that our local search algorithms explore well-defined neighborhoods, but in a heuristic fashion. We use a randomized greedy heuristic to find a solution to the subproblem defined by a pair of adjacent cells (and their neighbors) in the current partition. To determine how far our heuristic solutions are from true local minima (i.e., to measure how good our local search is), we would like to solve these subproblems exactly. They are, however, NP-hard.

Even so, we can still find exact solutions to the subproblems generated by a restricted variant of $\mathcal{L}_2$, the simplest local search we studied. For each pair $\{R, S\}$ of adjacent cells, we consider the auxiliary instance defined solely by uncontracted versions of R and S. Moreover, for efficiency we restrict ourselves to solutions to this instance with at most two cells (in principle, one could find solutions with even fewer cut edges by allowing a partition in three cells).

We now describe a branch-and-bound algorithm to find solutions to this restricted problem. Formally, we have an undirected graph $G' = (V', E')$ with vertex sizes and edge weights denoted by $s(.)$ and $w(.)$. (In our case, V' is the set of all fragments of R and S, whereas E' represents all edges between these fragments.) We want to divide it into two parts (A, B) such that:

1. the total size of each part is at most U;

2. the total weight of the edges that connect A and B is minimized.

Our algorithm implicitly enumerates all possible sets A and B, in search for the sets A^* and B^* that represent the optimum solution to our problem.

At all times, we maintain A' and B', two disjoint subsets of V', both initially empty. The assumption is that $A' \subseteq A^*$ and $B' \subseteq B^*$. In each step, we divide the current subproblem in two by picking a vertex v from $V' \setminus (A' \cup B')$ and assigning it to either A' or B'. (Of course, the vertex is not assigned to a particular set if their combined size exceeds U.) We then recursively solve both subproblems created, and return the best solution found, if any.

To make this algorithm practical, we need a good lower bound on the cost of any solution (A, B) with $A' \subseteq A$ and $B' \subseteq B$. Note that, for any such pair (A, B), the set of edges between A and B is also a cut separating A' and B'. Therefore, the *minimum cut* between A' and B' in G' is a lower bound on the value of the final solution—and a good one in practice. Whenever the lower bound is at least as high as the best known upper bound (initialized with our heuristic solution), we make no recursive calls, effectively pruning the search tree. After all, we know no subproblem would lead to a better solution. Moreover, if the minimum cut happens to be balanced (i.e., with the size of both parts bounded by U), we prune the tree as well, updating the best upper bound if necessary.

Note again that we use this branch-and-bound routine only to evaluate the solution quality of our heuristic local searches. We do not use it in the "production" version of PUNCH.

5 Balanced Partitions

As described so far, PUNCH solves the standard graph partitioning problem, which does not guarantee any bound on the number of cells in the solution; it only ensures that no cell will have a size greater than U. However, we can use PUNCH to compute a balanced partition as well. In this variant of the problem, the inputs are the number of cells (k) and the tolerated level of 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.

Suppose we want to find an ε-balanced partition that consists of at most k cells. We do so by first using 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 can potentially violate is that it could contain $\ell > k$ 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.

More precisely, let $V_1, V_2, \ldots, V_k$ denote the base cells, and let W be the set of fragments of the remaining cells. We start an iterative process. In each round, we set $U' = \max_{1 \le i \le k}(U - s(V_i))$ 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. 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'.

Note that some of the details of this rebalancing algorithm have yet to be specified.

First, we have to specify how to choose the k initial base cells. Our implementation uses a randomized algorithm. Each cell C of the initial solution we assign the score $(2 + r)s(C)$, where r is picked uniformly at random between 0 and 1, and choose the k cells with highest score.

Second, we need to specify how we assign cells of P' to the base cells. Once again, we use randomization. We process the cells of P' in decreasing order of their sizes. For each cell $C \in P'$, we check if it fits in any of the base cells V_i. Among all base cells V_i with $s(V_i) + s(C) \le U$, we pick one at random with probability proportional to $1/s(V_i)$, thus preferring tighter fits. If cell C does not fit anywhere, we just skip it; it will be split in the next round.

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

One problem remains: our approach may fail due to the fact that 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. However, for the inputs tested, setting the threshold to $U/3$ is sufficient.

Note that, like other algorithms for the balanced problem, we may sacrifice the connectivity of the cells to make them balanced.

6 Experiments

6.1 Methodology

We implemented our partitioning algorithm 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, 6 x 64 kB L1, 6 x 256 kB L2, and 12 MB L3 cache.

We use two graphs in our main experiments, both taken from the webpage of the 9th DIMACS Implementation Challenge [7]. 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 [29]. The *USA* road network (generated from TIGER/Line data [33]) 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 [12] to compute s–t cuts. For our application, we found that the version using FIFO order, frequent global relabelings, and the *send* operation performs best (see [12] for details).

Unless otherwise stated, we use the following parameters for PUNCH. During the filtering phase, we use $\alpha = 1$, $f = 10$, and $\mathcal{C} = 2$, and detect both tiny and natural cuts. In the assembly phase, we use the variant $\mathcal{L}_2^+$ as local search, with $\varphi = 16$. We do not use the combination heuristic by default. After giving our main results in the following subsection, we present additional experiments that confirm that these parameter choices present a reasonable trade-off between running times and solution quality. Unless otherwise mentioned, we use all 12 cores during natural-cut detection and the assembly phase; our implementation of tiny-cut detection is sequential.

6.2 Main Results

We start our experimental evaluation by reporting the performance of the default version of PUNCH on Europe and USA when varying U from 2^{10} to 2^{22}. Table 1 reports the average number of cells, the average solution quality (i.e., the number of cut edges), and the average running time of PUNCH. Since our algorithm is nondeterministic due to parallelism and randomness, all values are aggregated over 50 runs, with different random seeds. (To speed up the computation, we ran the filtering phase 5 times, generating 5 contracted graphs, and then ran the assembly phase 10 times on each.)

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. The actual number varies because the filtering phase grows BFS trees parameterized by U. When U is small, more edges are marked as candidates and are kept uncontracted.

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. 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 because filtering is less effective. In total, we need between 1 and 3 minutes to find good partitions of Europe or the USA, which is quite practical.

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.[1] The difference is much less pronounced for smaller values of U.

We observe that, although randomized, our algorithm is very robust. Over 50 executions, the best and worst solutions found are not far from the average. Another interesting observation is that the solutions found by our algorithm, although not perfectly balanced, are not too far from

[1] This could be—at least partially—an artifact of this particular data set; as observed on the DIMACS webpage, several important road segments, including some on bridges and freeways, are missing from the USA graph.

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

it. On average, PUNCH finds solutions with about 15% more cells than a perfectly balanced partition.

In the remainder of this section we study the effects of various parameters on the overall performance of PUNCH. To simplify the exposition, the experiments focus on a single instance: Europe with $U = 65\,536$.

6.3 Filtering Phase

We start our parameter tests by determining whether the filtering phase is helpful. On Europe with $U = 65\,536$, we ran different versions of the filtering phase, with or without the routines for detecting tiny and natural cuts. Whenever natural cuts were used, we set $\mathcal{C} = 2$. In every case we ran the standard PUNCH assembly phase (with $\varphi = 16$).

Table 2 summarizes the results obtained. In each case, we show the average number $|V'|$ of vertices (fragments) at the end of the filtering phase, the average solution cost (after the assembly phase), and the running times in second (of each subphase separately and in total). Because these experiments are rather slow, each entry is aggregated over 10 runs.

These results show that natural cuts are crucial to the overall performance of PUNCH. Without natural cuts, the assembly phase must work on a much larger graph. Not only does this make the overall algorithm much slower (by orders of magnitude), it also leads to significantly worse solutions on average. By contracting dense regions, natural cuts eliminate many potential bad choices the assembly phase could make.

In contrast, the effect of tiny cuts is more muted. They do not have any discernible positive effect on solution quality. (The effect may even be slightly negative, maybe due to the biases it

Table 2: Performance of PUNCH on Europe with $U = 65\,536$ depending on whether tiny cuts (tny) and/or natural cuts (nat) are processed (✓) or not (×) during the filtering phase.

CUTS				TIME [S]			
TNY	NAT	$\lvert V' \rvert$	SOL	TNY	NAT	ASM	TOTAL
×	×	18010173	11996	—	—	2234.6	2234.6
✓	×	8966360	12124	24.4	—	1148.5	1172.9
×	✓	114268	11382	—	109.0	5.3	114.3
✓	✓	104410	11432	24.4	48.2	4.9	77.5

introduces when trees are grown during natural cut generation.) By reducing the total number of fragments, however, tiny cuts accelerate the assembly phase slightly. But the biggest advantage of tiny cuts is making the generation of natural cuts much faster, thus reducing the overall running time of the entire algorithm. The difference would be even more pronounced on machines with fewer cores: while tiny-cut detection uses only one core, natural cuts use twelve.

Next, we check how the choice of natural cuts affects the performance of PUNCH. Recall that the main parameters are α (which determines the ring size) and f (which sets the core size relative to the ring). Table 3 reports the average solution values on Europe with $U = 65\,536$ when α and f vary. As usual, averages are taken over 50 runs.

Table 3: Average solutions for $U = 65\,536$ on Europe depending on the sizes of the ring and the core, as controlled by parameters α and f.

	f			
α	5	10	15	20
0.25	11614	11596	11617	11683
0.50	11546	11484	11522	11582
0.75	11481	11453	11458	11499
1.00	11471	11413	11402	11444

The table shows that the algorithm is not too sensitive to the exact choice of parameters, but solutions improve slightly as α increases. In particular, growing BFS trees of size exactly U (i.e., setting $\alpha = 1.0$) seems to be a reasonable choice, since it generates cuts at the appropriate scale and creates fewer candidate edges. (In fact, when $\mathcal{C} > 1$, one could even consider having $\alpha > 1$ in at least one iteration.) The dependence on f is less clear, with slight advantage to values between 10 and 15, supporting our choice of $f = 10$ as default.

Another parameter of the filtering phase is the coverage factor $\mathcal{C}$, which controls how many times a vertex belongs to a core. Table 4 reports, for various coverage factors $\mathcal{C}$, the number of vertices in the contracted graph, the running time of each phase, and the quality of the final solutions obtained.

As one would expect, increasing $\mathcal{C}$ yields bigger contracted graphs (and higher running times), since more edges are marked. In contrast, the solution quality increases only slightly when switching from $\mathcal{C} = 1$ to $\mathcal{C} = 2$, then it gets worse again. This indicates that the assembly phase can be "misguided" when given too many options, since it is just a heuristic. We have already observed a similar behavior in Table 2.

In fact, one should expect this effect to become less pronounced as more sophisticated (and time-consuming) procedures are used in the assembly phase. To test this, we repeated the same

Table 4: Performance of PUNCH on Europe with $U = 65\,536$ and different coverage ($\mathcal{C}$) values.

			TIME [S]		
$\mathcal{C}$	$\lvert V' \rvert$	SOL	FLT	ASM	TOTAL
1	62546	11425	50.2	2.7	53.0
2	104410	11415	74.3	4.8	79.0
3	134146	11438	100.2	6.4	106.6
4	157318	11464	123.4	7.6	131.0
5	176523	11470	147.4	8.8	156.2

experiment but varied φ (the maximum number of failures in the local search) as well. The average solutions thus obtained are shown in Table 5.

Table 5: Average solution quality on Europe with $U = 65\,536$ for different coverage ($\mathcal{C}$) and maximum failure (φ) values.

	COVERAGE				
φ	1	2	3	4	5
4	11635	11652	11708	11750	11769
16	11425	11415	11438	11464	11470
64	11299	11250	11258	11285	11300

With $\varphi = 4$, the assembly phase actually finds the best solutions when $\mathcal{C} = 1$; for such a simple heuristic, increasing $\mathcal{C}$ only hurts. For $\varphi = 16$, as already seen, $\mathcal{C} = 2$ is slightly better. When $\varphi = 64$, setting $\mathcal{C}$ to 2 or even 3 is clearly better than 1. Overall, using $\mathcal{C} = 2$ is a reasonable compromise between speed and solution quality.

Finally, we test the scalability of our implementation. As already mentioned, the tiny-cut processing routine is sequential, but both the natural-cut heuristic (the main bottleneck of the filtering phase) and the assembly phase are parallelized. Table 6 reports the average running times of each stage of our algorithm when the number of cores varies. We notice that running on more cores significantly accelerates the algorithm. The speedups is not perfect, however, mainly because all phases still have some sequential subroutines, such as finding centers or updating the current solution. Moreover, contention to access the two memory banks in our machine can be an issue.

Table 6: Execution time (in seconds) of each stage (tiny cuts, natural cuts, and assembly) when varying numbers of cores.

	AVERAGE TIME [S]			
CORES	TNY	NAT	ASM	TOTAL
1	24.4	247.1	27.1	298.6
2	24.3	140.9	14.9	180.2
4	24.4	82.4	8.8	115.5
6	24.4	65.5	6.8	96.6
8	24.4	59.1	5.8	89.2
12	24.3	51.1	4.9	80.3

6.4 Assembly Phase

We now consider different parameter choices during the assembly phase. We start by evaluating how running times and solution quality are affected by φ, the parameter that controls the number of failures allowed per pair during the local search. Table 7 reports the results on Europe with $U = 65\,536$. Values are aggregated over 50 runs of $\mathcal{L}_2^+$, our default local search. For each value, we show the average solution found, the number of subproblems improved and tested, and the average running time of the assembly phase in seconds. (See Table 1 for data on the filtering stage, including running times.)

Table 7: Performance of $\mathcal{L}_2^+$ on Europe for $U = 65\,536$ and varying φ: average solution, average number of subproblems improved and tested, and average assembly time (in seconds).

		SUBPROBLEMS		
φ	SOL	IMPROV	TESTED	TIME
0	14422	0	0	0.2
1	12106	461	2486	1.6
2	11838	572	4746	2.1
4	11660	672	8670	2.7
8	11513	755	15852	3.5
16	11395	838	28866	4.8
32	11327	899	52953	6.9
64	11256	946	96924	10.8
128	11195	1003	180506	18.1
256	11161	1029	328338	30.8
512	11113	1060	616928	55.8

These results demonstrate that the local search procedure finds significantly better results than the constructive algorithm (shown as $\varphi = 0$ in the table). As expected, both solution quality and running times increase with φ. More interestingly, the success rate varies significantly: about 20% of the subproblems examined are actually improved with $\varphi = 1$, but fewer than 1% for $\varphi \geq 64$. Our default value $\varphi = 16$ is a reasonable compromise between solution quality and running time.

We now compare the three neighborhoods we considered: $\mathcal{L}_2$, $\mathcal{L}_2^+$, and $\mathcal{L}_2^*$. As shown in Figure 6, they differ in how a subproblem is defined: $\mathcal{L}_2$ reoptimizes two neighboring cells, $\mathcal{L}_2^+$ also considers contracted versions of its neighbors, and $\mathcal{L}_2^*$ disassembles the neighbors as well. We tested each local search with three different values of φ (16, 64, and 256). Table 8 reports, in each case, the average solution found, the assembly time in seconds, and the average number of subproblems improved and tested. Averages are taken over 50 runs of each local search, from the same 50 initial solutions.

As one would expect, $\mathcal{L}_2$ and $\mathcal{L}_2^+$, which must solve subproblems with similar sizes, have comparable running times, and are much faster than $\mathcal{L}_2^*$. Curiously, $\mathcal{L}_2^+$ is actually slightly faster than $\mathcal{L}_2$, despite the fact that the subproblems it must solve are bigger. This is because $\mathcal{L}_2^+$ needs fewer improvements to reach a local optimum, which leads to fewer subproblems tested in total.

In terms of solution quality, the algorithm based on $\mathcal{L}_2^+$ also dominates the other neighborhoods. This is not surprising in the case of $\mathcal{L}_2$, which visits a much more restricted neighborhood. In contrast, the neighborhood explored by $\mathcal{L}_2^*$ dominates $\mathcal{L}_2^+$ in theory, but ends up being worse in practice because we solve subproblems only heuristically.

Table 8: Different combinations of local search (ls) and maximum failure (φ) on Europe with $U = 65\,536$: average solution, number of subproblems improved and tested, and assembly times in seconds.

φ	LS	SOL	SUBPROBLEMS IMPROV	SUBPROBLEMS TESTED	TIME
16	$\mathcal{L}_2$	12036	954	35486	5.1
	$\mathcal{L}_2^+$	11398	834	28849	4.8
	$\mathcal{L}_2^*$	12366	324	34902	18.8
64	$\mathcal{L}_2$	11784	1124	115534	11.6
	$\mathcal{L}_2^+$	11255	947	96273	10.7
	$\mathcal{L}_2^*$	12140	398	126744	60.5
256	$\mathcal{L}_2$	11640	1232	385100	33.0
	$\mathcal{L}_2^+$	11152	1025	329584	30.9
	$\mathcal{L}_2^*$	11933	472	473439	219.5

In fact, the branch-and-bound algorithm proposed in Section 4.4 can improve the solutions found by all local searches. For example, it improves a local optimum found by $\mathcal{L}_2^+$ (with $\phi = 256$) by more than 1% (from 11134 to 10980), but the computation takes more than a day. As the next section will show, however, there are faster ways of obtaining good solutions.

6.5 Combination

We now consider how to obtain better solutions if more time is available. One possibility has already been discussed: increasing the maximum number of failures. As shown in Table 7, setting φ to 512 instead of 16 improves the average solution by roughly 3%.

We now consider two other methods. The first is a simple *multistart* heuristic: run the combination of constructive algorithm and local search M times and pick the best solution thus found. The second is a *combination* method: as explained in Section 4.3.1, it maintains a pool of $\sqrt{M}$ *elite solutions*, and combines them periodically (among themselves and with new solutions).

Table 9 reports the average solutions and running times of both heuristics as a function of M (the number iterations) and φ. Running times are for the assembly phase only, since all methods share the same filtering stage (see Table 1). Solution values and times are averages over five runs.

As expected, solutions get better as the number of multistart iterations increases. They do so quite slowly, though. For a fixed φ, increasing M from 16 to 256 improves the solution by less than 1%. Comparing this with Table 7, increasing φ seems to be a more effective use of the extra time.

In contrast, the combination algorithm leads to significantly better solutions. With the highest values of M and φ we tested, the algorithm takes almost an hour, but improves the average solutions found by the basic version of PUNCH, reported in Table 1, by almost 6%. Even a less agressive choice of parameters, such as $M = 16$ and $\varphi = 32$, already leads to improvements of up to 4% over the basic solution in less than five minutes (including the filtering stage). By setting these parameters appropriately, users can trade off solution quality for running time.

Table 9: Average results on Europe for $U = 65\,536$ when varying φ and the number of iterations (M). "Multistart" is a plain multistart algorithm. "Combination" includes the multistart algorithm plus recombination, using a pool with capacity $\sqrt{M}$. All times are in seconds and refer to the assembly phase only.

		MULTISTART		COMBINATION	
M	φ	SOL	TIME	SOL	TIME
16	1	11944	29.5	11274	43.0
	2	11733	36.9	11185	53.0
	4	11531	46.9	11081	66.5
	8	11395	59.4	10991	85.4
	16	11311	79.8	10970	119.3
	32	11223	114.5	10901	179.6
64	1	11913	118.4	11073	179.2
	2	11686	148.6	10977	219.7
	4	11483	186.0	10900	274.8
	8	11367	240.2	10864	359.9
	16	11273	319.1	10814	500.2
	32	11200	453.8	10812	753.2
256	1	11874	467.9	10955	714.7
	2	11638	595.2	10862	888.6
	4	11464	744.0	10812	1107.4
	8	11347	953.6	10764	1444.7
	16	11251	1277.3	10731	2023.8
	32	11155	1816.7	10722	3070.4

6.6 Balanced Partitions

We now study how effective PUNCH is when computing *balanced* partitions, in which the maximum number k of cells is bounded. The size of each cell must be 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 the literature [26, 17, 16, 24].

As described in Section 5, to find a balanced partition with PUNCH we first run the filtering stage with $U = U^*/3$. We can then run the assembly stage with $U = U^*$ to generate an initial unbalanced solution, which we make balanced by reassigning some fragments. In practice, we noticed 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 and picking the best result.

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 5); (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 tested values of k ranging from 2 to 64, and ran the entire algorithm 100 times for each k. Table 10 reports the results, including the time spent during each phase of the algorithm: filtering (FLT), generating the $\lceil 32/k \rceil$ unbalanced solutions (UNB) and rebalancing each 50 times (BAL). Averages in the table are taken over all 100 executions.

For consistency with previous work, we ran our algorithm on the version of the European road network tested in [16, 24]. Although it comes from the same source [29], it has a few more vertices

Table 10: Performance of PUNCH when finding balanced partitions on Europe* with $\varepsilon = 0.03$, aggregated over 100 runs. We report the average (avg sol) and best (best sol) solutions found, both before (unb) and after (bal) rebalancing. We also show the average running time of each stage: filtering, finding a potentially unbalanced solution, and rebalancing it.

INSTANCE		AVG SOL		BEST SOL		TIME [S]		
GPH	CEL	UNB	BAL	UNB	BAL	FLT	UNB	BAL
DEU	2	166	166	164	164	36.5	8.6	4.8
	4	408	410	400	400	30.4	21.5	5.5
	8	702	746	689	711	25.3	24.3	11.7
	16	1158	1188	1136	1144	21.5	25.0	3.4
	32	1962	2032	1913	1960	18.9	20.7	3.2
	64	3185	3253	3127	3165	14.0	32.1	3.9
EUR	2	130	130	129	129	183.2	10.1	61.9
	4	307	309	307	309	163.2	16.5	35.1
	8	668	671	634	634	137.3	27.7	10.8
	16	1322	1353	1268	1293	115.4	28.3	7.6
	32	2328	2362	2269	2289	97.6	28.7	3.7
	64	3867	3984	3796	3828	82.8	52.1	9.7
USA	2	70	70	69	69	137.8	6.2	11.3
	4	263	263	263	263	127.6	6.1	5.4
	8	514	546	514	530	118.6	6.8	4.9
	16	1009	1037	988	1011	110.3	9.6	2.8
	32	1838	1883	1808	1829	102.5	11.5	1.3
	64	3246	3350	3204	3260	93.5	22.8	3.6

(less than 1%) than our version (downloaded from [7]), and it is not connected. We refer to the disconnected instance as EUR, as in [16, 24]. We also run our algorithm on DEU, a subgraph of EUR (also disconnected) representing Germany, and on the USA graph.

The table shows that rebalancing an unbalanced partition increases the cut size only slightly. The additional rebalancing time does not depend much on k, and is usually much smaller than the time spent to generate the unbalanced partition.

As already mentioned, the most prominent general-purpose software libraries focus on finding balanced partitions. Tables 11 and 12 show how the average and best solutions found by PUNCH compare with those found by other popular methods (as reported in [25], the full version of [24]). Note that DEU and EUR are the only road networks for which detailed results are presented in that paper. For consistency with the other methods, Table 12 only considers the first 10 random seeds for PUNCH.

Note that, on average, PUNCH finds much better solutions than METIS or SCOTCH do. These methods, however, are faster than ours, as Table 13 indicates. Their emphasis is on finding an adequate solution very quickly.

In contrast, KaPPa, KaSPar, and PUNCH sacrifice some running time in order to obtain significantly better partitions. (The values we report refer to the “strong” versions of KaPPa and KaSPar.) Even so, for every k the average solution found by our algorithm improves on the previously best known results, which were obtained by KaSPar. The improvement is often higher than 10%. Moreover, KaSPar is considerably slower (but sequential) on a comparable machine (an Intel Xeon X5355 running at 2.67 GHz), as Table 13 shows. We conclude that PUNCH is

Table 11: Average balanced partitions with $\varepsilon = 0.03$ for various algorithms on Europe.

INSTANCE		AVERAGE SOLUTION				
GPH	CEL	PUNCH	KASPAR	KAPPA	SCOTCH	METIS
DEU	2	166	172	221	295	286
	4	410	426	542	726	761
	8	746	773	962	1235	1330
	16	1188	1333	1616	2066	2161
	32	2032	2217	2615	3250	3445
	64	3253	3631	4093	4978	5385
EUR	2	130	138	—	469	—
	4	309	375	619	952	1626
	8	671	786	1034	1667	3227
	16	1353	1440	1900	2922	9395
	32	2362	2643	3291	4336	9442
	64	3984	4526	5393	6772	12738

Table 12: Best balanced partitions (over 10 runs) found by various algorithms with $\varepsilon = 0.03$.

INSTANCE		BEST SOLUTION				
GPH	CEL	PUNCH	KASPAR	KAPPA	SCOTCH	METIS
DEU	2	164	167	214	295	268
	4	404	419	533	726	699
	8	721	762	922	1235	1174
	16	1159	1308	1550	2066	2041
	32	1993	2182	2548	3250	3319
	64	3187	3610	4021	4978	5147
EUR	2	129	133	—	469	—
	4	309	355	543	952	846
	8	639	774	986	1667	1675
	16	1293	1401	1760	2922	3519
	32	2314	2595	3186	4336	7424
	64	3912	4502	5290	6772	11313

Table 13: Average running times (in seconds) of different partitioning packages to find balanced partitions with $\varepsilon = 0.03$.

INSTANCE		AVERAGE TIME [S]				
GPH	CEL	PUNCH	KASPAR	KAPPA	SCOTCH	METIS
DEU	2	50	231	68	3	5
	4	57	244	77	6	5
	8	61	250	100	10	5
	16	50	278	106	13	5
	32	43	284	73	16	5
	64	50	294	50	19	5
EUR	2	255	1946	—	12	—
	4	215	2168	441	25	29
	8	176	2232	418	39	29
	16	151	2553	498	52	31
	32	130	2599	418	65	31
	64	145	2534	308	77	30

an excellent alternative for finding balanced partitions of road networks. We stress that, unlike PUNCH, the other algorithms are general-purpose partitioners. Their performance could be much better on other types of graphs.

In order to evaluate whether these improvements stem from our filtering or assembly phase, we ran SCOTCH on our contracted graph (after filtering). The resulting cuts are much better than those found by pure SCOTCH (particularly for large k), but still worse than those found by PUNCH alone. This indicates that both phases of PUNCH are important for the improvements we observe.

7 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. More importantly, it preserves 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 the best known partitions for road networks, improving previous bounds by more than 10% on average. Altogether, PUNCH is slower compared to some previous graph partitioning algorithms, but it needs only a few minutes to generate an excellent partition, which is fast enough for most applications.

Regarding future work, we are interested in testing PUNCH on other inputs. Preliminary experiments on a set of standard benchmark instances [34] indicate that the filtering phase of PUNCH is not very helpful. This is expected, since these instances do not have as many natural cuts as road networks do. However, running the assembly phase of PUNCH alone improves the best known solution for some instances. We plan to investigate this more thoroughly in the future. In addition, it would be interesting to develop a distributed version of PUNCH, which we believe would work well because most of the computations are largely independent from one another. Finally, we are interested in developing a more realistic model for road networks, incorporating the concept of natural cuts.

Acknowledgment

We thank Dennis Luxen for providing us with the data from [16, 24].

References

[1] R. Andersen and K. J. Lang. An Algorithm for Improving Graph Partitions. In *Proceedings of the 19th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'08)*, pages 651–660, 2008.

[2] 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.

[3] 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.

[4] 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.

[5] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 73–92. American Mathematical Society, 2009.

[6] 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*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.

[7] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, 2006.

[8] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Conference on Design Automation*, pages 175–181, 1982.

[9] 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.

[10] 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*, pages 1–75. Kluwer, 1996.

[11] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 93–139. American Mathematical Society, 2009.

[12] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[13] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[14] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC'05)*, page 28. ACM Press, 1995.

[15] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.

[16] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a Scalable High Quality Graph Partitioner. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–12. IEEE Computer Society, 2010.

[17] 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.

[18] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[19] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In P. Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*. Springer, May 2010.

[20] 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.

[21] 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 *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC'09)*, pages 6–6, New York, NY, USA, 2009. ACM.

[22] 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.

[23] 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.

[24] V. Osipov and P. Sanders. n-Level Graph Partitioning. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, Lecture Notes in Computer Science, pages 278–289. Springer, 2010.

[25] V. Osipov and P. Sanders. n-Level Graph Partitioning. *CoRR*, abs/1004.4024, 2010.

[26] 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*, Lecture Notes in Computer Science, pages 493–498. Springer, 1996.

[27] R. Preis and R. Diekmann. The PARTY Partitioning Library, User Guide . Technical report, University of Paderborn, Germany, 1996. tr-rsfb-96-02.

[28] D. Pritchard and R. Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transaction on Algorithms*, 2010. To appear.

[29] PTV AG - Planung Transport Verkehr. `http://www.ptv.de`.

[30] M. G. C. Resende and R. F. Werneck. A hybrid heuristic for the p-median problem. *Journal of Heuristics*, 10(1):59–88, 2004.

[31] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS J. Computing*, 14(3):228–246, 2002.

[32] 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*, pages 491–541. Morgan Kaufmann, 2003.

[33] D. US Census Bureau, Washington. UA Census 2000 TIGER/Line files. 2002.

[34] C. Walshaw. The Graph Partitioning Archive.

[35] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoulès, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007.

[36] 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.

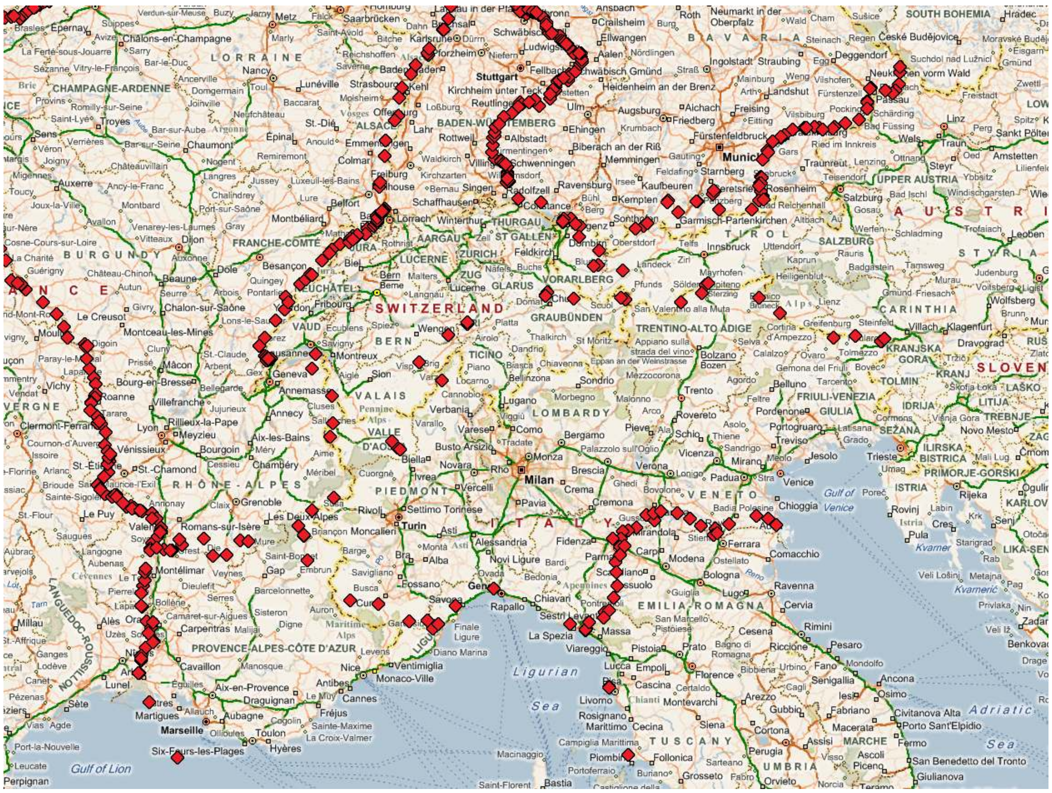

Figure 7: Partition generated by PUNCH. The input is the European road network, U is set to 1 000 000. PUNCH identifies the alps and the borders (Switzerland/France, France/Germany, Italy/France) as natural cuts.

Figure 8: Partition generated by PUNCH. The input is the European road network, U is set to 1 000 000. PUNCH identifies the Pyrenees as natural cut.

Table 14: Performance of 50 runs of PUNCH on subraphs of 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 and in total are shown.

INSTANCE			CELLS			SOLUTION			TIME (S)			
GPH	n	U	LB	AVG	$\|V'\|$	BEST	AVG	WORST	TNY	NAT	ASM	TOTAL
NY	264346	1024	259	300.6	24210	3156	3187	3221	0.3	0.3	0.6	1.1
		4096	65	76.1	8575	1067	1091	1132	0.3	0.4	0.2	0.9
		16384	17	18.9	2650	360	372	385	0.3	0.6	0.1	1.0
		65536	5	6.0	526	65	65	66	0.3	1.1	0.0	1.4
BAY	321270	1024	314	370.8	22853	2774	2802	2829	0.3	0.2	0.5	1.1
		4096	79	92.4	8588	879	893	909	0.3	0.3	0.2	0.8
		16384	20	23.9	2540	249	260	274	0.3	0.5	0.0	0.8
		65536	5	6.0	512	52	54	56	0.3	0.8	0.0	1.1
COL	435666	1024	426	497.7	26639	3204	3251	3287	0.4	0.3	0.6	1.3
		4096	107	126.9	10367	1131	1152	1175	0.4	0.3	0.2	1.0
		16384	27	31.1	3456	412	424	448	0.4	0.5	0.1	1.0
		65536	7	7.7	1047	126	135	156	0.4	0.9	0.0	1.3
FLA	1070376	1024	1046	1227.4	61553	7685	7723	7762	1.2	0.8	1.2	3.2
		4096	262	304.5	24636	2646	2674	2697	1.2	0.8	0.5	2.5
		16384	66	78.7	8763	831	846	862	1.2	1.4	0.2	2.7
		65536	17	18.8	2460	224	233	246	1.2	2.5	0.0	3.7
NW	1207945	1024	1180	1365.2	70626	8533	8575	8638	1.3	0.8	1.4	3.5
		4096	295	345.8	27764	3000	3038	3081	1.3	0.8	0.6	2.7
		16384	74	87.0	9519	1028	1043	1065	1.3	1.2	0.2	2.7
		65536	19	22.3	2490	310	322	337	1.3	2.2	0.1	3.6
NE	1524453	1024	1489	1704.4	123613	15300	15363	15437	1.9	1.5	2.9	6.2
		4096	373	421.2	51596	5847	5892	5937	1.9	1.6	1.3	4.8
		16384	94	106.6	18942	2229	2260	2286	1.9	2.5	0.6	4.9
		65536	24	28.9	5414	726	749	768	1.9	4.3	0.2	6.3
CAL	1890815	1024	1847	2140.6	129274	15952	16025	16095	2.3	1.6	3.0	6.9
		4096	462	533.7	49670	5634	5674	5722	2.3	1.7	1.2	5.2
		16384	116	135.4	16234	1896	1948	1983	2.3	2.6	0.4	5.3
		65536	29	33.0	4288	577	587	607	2.3	4.4	0.1	6.8
LKS	2758119	1024	2694	3053.9	257252	31316	31414	31504	3.5	2.7	7.0	13.1
		4096	674	758.3	113523	12839	12921	13009	3.5	2.8	3.7	10.0
		16384	169	190.6	42198	5121	5184	5244	3.5	4.3	1.7	9.4
		65536	43	48.7	12145	1845	1896	1943	3.5	7.1	0.5	11.0
E	3598623	1024	3515	4020.7	260877	31847	31921	31999	4.5	3.1	6.2	13.7
		4096	879	999.1	112245	12433	12503	12579	4.5	3.2	2.9	10.6
		16384	220	251.8	41452	4765	4816	4890	4.5	4.7	1.3	10.4
		65536	55	66.2	12513	1667	1711	1759	4.5	8.2	0.4	13.1
W	6262104	1024	6116	7089.0	409602	49953	50093	50223	8.2	5.5	9.7	23.5
		4096	1529	1765.7	158007	18001	18085	18190	8.2	5.6	3.9	17.7
		16384	383	441.9	53243	6475	6525	6593	8.2	8.0	1.4	17.6
		65536	96	109.0	15174	2167	2195	2237	8.2	13.7	0.4	22.3
CTR	14081816	1024	13752	15632.6	1147988	139717	139888	140153	25.2	18.8	34.9	79.0
		4096	3438	3881.5	511257	56394	56621	56746	25.2	18.3	18.5	62.0
		16384	860	972.7	194985	22490	22612	22728	25.2	22.9	8.3	56.4
		65536	215	244.9	60154	8381	8472	8617	25.2	36.0	2.7	63.9
USA	23947347	1024	23387	26725.3	1826293	222386	222564	222765	33.2	22.9	50.4	106.5
		4096	5847	6640.8	787382	87541	87785	87956	33.1	22.8	25.4	81.3
		16384	1462	1661.6	293206	34206	34355	34566	33.1	31.8	11.2	76.1
		65536	366	417.9	89762	12634	12761	12854	33.1	53.9	3.7	90.8