Fast and Stable Repartitioning of Road Networks

Valentin Buchhold
Karlsruhe Institute of Technology, Germany

Daniel Delling
Apple Inc, Cupertino, CA, USA

Dennis Schieferdecker
Apple Inc, Cupertino, CA, USA

Michael Wegner
Apple Inc, Cupertino, CA, USA

Abstract

We study the problem of graph partitioning for evolving road networks. While the road network of the world is mostly stable, small updates happen on a relatively frequent basis, as can been observed with the OpenStreetMap project (`http://www.openstreetmap.org`). For various reasons, professional applications demand the graph partition to stay roughly the same over time, and that changes are limited to areas where graph updates occur. In this work, we define the problem, present algorithms to satisfy the stability needs, and evaluate our techniques on continental-sized road networks. Besides the stability gains, we show that, when the changes are low and local, running our novel techniques is an order of magnitude faster than running graph partitioning from scratch.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Dynamic graph algorithms

Keywords and phrases Graph repartitioning, stable partitions, road networks, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.26

Funding This work was done while the first author was interning at Apple Inc.

1 Introduction

Graph partitioning is a core subroutine of many applications such as distributed computing, VLSI design, or shortest path computation. Although being NP-hard in general, many very efficient algorithms perform really well for realistic input problems (see [5] for an overview). In particular, the road network of the world can be partitioned with high quality in a multi-threaded fashion on a single machine in a few hours [8, 26]. However, one problem when applying these techniques to production systems is that the partition is not *stable*, i.e., small changes to the input may result in very different solutions (see Figure 1). This is a problem for various reasons. Firstly, road networks update surprisingly frequent, as can be seen by OpenStreetMap data, but these changes are often only minor, and restricted to local areas. Secondly, computing via paths [1] with customizable route planning [7] or customizable contraction hierarchies [10] exploit boundary nodes of partitions, which then may show very different results between two inputs in areas where no update has been applied. Finally, rerunning the full partitioning for every update seems like a waste of CPU hours, unnecessarily increasing data build times.

Our Contribution. We study the problem of making graph partitioning stable. The key idea is to obtain a partition for a slightly changed graph but analyzing both the previous graph and its partition. We identify regions that have changes and then try to repair the partition by rerunning graph partitioning subroutines on much smaller subgraphs. As a

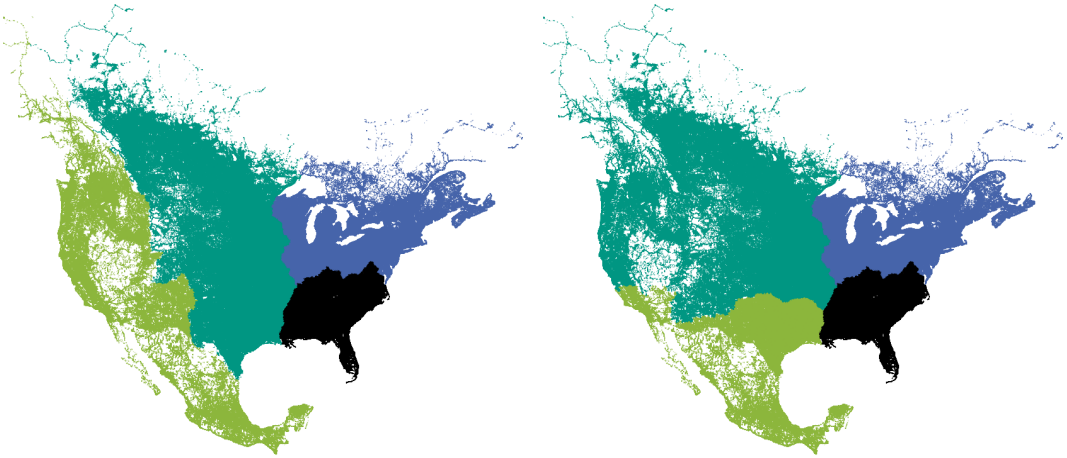

Figure 1 Two graph partitions with 4 cells each for the OpenStreetMap data of North America from August 2018 (left) and September 2018 (right). Both partitions have been obtained from the same algorithm with the same parameters and the same seeds. While the eastern part of the continent keeps roughly the same cut, the western part is cut very differently. However, within that month, only 0.5% of the vertices have churned (removed or added).

result, for an evolving road network like the OpenStreetMap data, we can keep the partition relatively stable, and are able to compute the updated partition an order of magnitude faster compared to running a graph partitioning algorithm from scratch.

Related Work. Motivated by several important applications, the graph partitioning problem has received considerable attention recently; see e.g. [5] for an overview. To partition road networks, early work [14, 2, 3] used general-purpose partitioners like Scotch [20], METIS [16], or Party [18]. However, one can compute partitions of significantly better quality when using special-purpose partitioners tailored to road networks.

The first such partitioning algorithm is PUNCH [8]. It introduces and exploits the concept of *natural cuts*, which are natural or man-made obstacles, such as rivers, mountains, and highways. At its heart is the *filtering phase*, which finds natural cuts by local maximum-flow computations and contracts all edges not contained in any natural cut. The *assembly phase* heuristically combines the resulting *fragments* to build a partition. Buffoon [24] incorporates the filtering phase of PUNCH into the general-purpose partitioning algorithm KaHIP [25].

An alternative approach to the filtering phase of PUNCH is Inertial Flow [26], which recursively bisects the network until the fragments are sufficiently small. To bisect the network, it exploits the geometric embedding of the network.

Like Inertial Flow, the FlowCutter algorithm [12] recursively bisects the network. For each bisection, it computes a Pareto set of nondominating cuts with respect to the cut size and balance, and picks a cut among those with a good tradeoff between cut size and balance. The recent InertialFlowCutter algorithm [11] is a variant of FlowCutter that uses geometric information, based on ideas from Inertial Flow. It is about 6 times faster than FlowCutter while preserving (or even slightly improving) the partition quality.

There has also been previous work on the graph *repartitioning* problem, although not in the context of road networks, but scientific computing applications. Various problems in solid and structural mechanics [37] and fluid dynamics [38] can be described by partial

differential equations (PDEs). Such PDEs can be solved by the finite-difference or finite-element method [36], which discretize the domain of the PDE by a *mesh*. The function value at each discretization point (i.e., vertex in the mesh) is approximately computed from the values at its neighboring vertices. Using an iterative scheme, new approximate values are determined by the values at the neighboring vertices from the previous iteration. Since finite-element meshes can become very large, they are partitioned into well-separated cells and distributed over multiple processing elements [5].

During the solution process, the mesh is refined in regions where large errors exist and coarsened in well-behaved regions. To maintain load balance, the mesh is periodically repartitioned. Besides the cut size (which correlates with the *communication volume*), the similarity between the old and new partition (which correlates with the *migration volume*) is an important optimization criterion for this application [5].

One approach are scratch-remap repartitioners [21, 32, 19, 29]. These first partition the new mesh using a state-of-the-art partitioner and then compute a migration-minimal mapping between the old and new partition. Since the new partition is produced from scratch, its cut size is small. However, the migration volume is often very high, since this criterion is considered only in the second phase.

Another approach are diffusion-based repartitioners [35, 27, 29]. These are inspired by the physical process of *diffusion*, i.e., vertices move from a cell of higher to one of lower vertex concentration. While this results in a good migration volume, the cut is often large.

The unified repartitioning algorithm [28] optimizes both criteria directly by combining the above-mentioned approaches. Following the multilevel graph partitioning approach, it iteratively contracts the new mesh using a variant of the heavy-edge matching algorithm [16], which matches two vertices only if they are in the same cell in the old partition. Next, the contracted mesh is partitioned twice using a scratch-remap and diffusion-based algorithm, respectively, and the partition with the better tradeoff between cut size and migration volume is picked. Finally, the mesh is iteratively uncontracted, using an improvement heuristic in each iteration to optimize the partition locally.

A rather simple approach [13, 33] is to introduce a zero-weight vertex for each cell in the old partition, which is not allowed to change its cell. This vertex is connected to each other vertex v in its cell by an edge whose weight represents the migration cost for v.

The problem we address is similar to the mesh repartitioning problem in that we optimize both the cut size and similarity. Note, however, that the old and new mesh are nested in the sense that new vertices result from splitting or merging vertices in the old mesh. Hence, there is a natural assignment of all vertices in the new mesh to cells in the old partition [34]. In fact, most work [21, 32, 27, 19, 29] considers a mesh with *fixed* topology, where adaptive mesh refinements are handled as vertex weight increases. In contrast, we allow vertices to be freely inserted into and removed from the network. For example, a newly constructed bridge that connects two previously disconnected cells has no natural assignment to any cell. Moreover, in the mesh repartitioning problem, the number of cells is fixed (since it is equal to the number of processing elements), while we allow the number of cells to change.

2 Preliminaries

We consider undirected graphs $G = (V, E)$ where each vertex $v \in V$ has a positive size $s(v)$ and each edge $\{u, v\} \in E$ has a positive weight $w(u, v)$. Our focus is on road networks, where vertices represent intersections and edges represent road segments. Partitioning algorithms often use *edge contractions*. To contract an edge $\{u, v\}$, we replace its endpoints by a single vertex w of size $s(w) = s(u) + s(v)$ and relink all edges incident on u or v to w. Multiple parallel edges are combined (adding up their weights) and self-loops are removed.

A *partition* of V is a set $P = \{C_1, \ldots, C_k\}$ of cells $C_i \subseteq V$ with the property that each vertex is contained in exactly one cell. A multilevel partition of V is a sequence $\mathcal{P} = \langle P^1, \ldots, P^L \rangle$ of partitions P^l, where l denotes the *level* of the partition. For ease of notation, we set $P^0 = \{\{v\} : v \in V\}$ and $P^{L+1} = \{V\}$. Since we use *nested* multilevel partitions, for each cell C_i^l, there is a cell $C_j^{l'}$ with $C_i^l \subseteq C_j^{l'}$ on all levels above; C_i^l is called a *subcell* of each $C_j^{l'}$. We denote by $c^l(v)$ the cell that contains v on level l. A *boundary* or *cut* edge on level l is an edge $\{u, v\}$ with $c^l(u) \neq c^l(v)$. Its endpoints are *boundary vertices* on level l. We denote by B^l the set of boundary vertices on level l.

2.1 Problem Statement

We are given two graphs $G = (V, E)$ and $\bar{G} = (\bar{V}, \bar{E})$, where $\bar{V}$ is obtained from V by inserting some vertices V^+ with $V^+ \cap V = \emptyset$ and removing some vertices $V^- \subseteq V$. Analogously, $\bar{E}$ is obtained from E by inserting some edges E^+ with $E^+ \cap E = \emptyset$ and removing some edges $E^- \subseteq E$. Hence, $\bar{V} = (V \setminus V^-) \cup V^+$ and $\bar{E} = (E \setminus E^-) \cup E^+$. We call G the *old graph* and $\bar{G}$ the *new graph*. In addition, we are given an L-level partition $\mathcal{P}$ of G with maximum cell sizes $U^1, \ldots, U^L$. The problem we consider is computing an L-level partition $\bar{\mathcal{P}}$ of $\bar{G}$ such that for each level l, $1 \leq l \leq L$, the size of each cell is bounded by U^l, the cut size is minimized, and the similarity between P^l and $\bar{P}^l$ is maximized.

It remains to formalize our notion of similarity. For real-world route planning systems following the partition-based overlay approach [30, 31, 15, 7], it is often desirable to keep the overlay topology fairly stable, as discussed in Section 1. Therefore, we define the similarity between two partitions P^l and $\bar{P}^l$ as the fraction of boundary vertices that are boundary vertices of both P^l and $\bar{P}^l$, i.e., $S^l = |B^l \cap \bar{B}^l| / |B^l \cup \bar{B}^l|$.

2.2 PUNCH

PUNCH [8] is a partitioning algorithm tailored to road networks. It has been applied [7, 6] to various partition-based shortest-path techniques, including CRP [7], Arc Flags [14, 17], and CHASE [3]. Given a graph G and a maximum cell size U, PUNCH splits the graph into cells of maximum size U while minimizing the cut size. It works in two phases. At its heart is the *filtering phase*, which finds *natural cuts* (natural or man-made obstacles, such as rivers, mountains, and railway tracks) and contracts all edges not contained in any natural cut. The *assembly phase* heuristically combines the resulting *fragments* to build a partition.

Filtering Phase. The *natural-cut heuristic* is executed in iterations. In each iteration, it picks a center c at random and builds a breath-first search (BFS) [23] tree T rooted at c until the total size of the vertices visited during the BFS reaches αU, where α is a parameter in $(0, 1]$. The neighbors $v \notin T$ of the vertices in T form the *ring* of c. Moreover, the vertices visited by the BFS before the total size reached $\alpha U / f$ form the *core* of c, where $f > 1$ is a second parameter. Then, the natural-cut heuristic temporarily contracts the core and the ring into a single source and sink vertex, respectively, determines a maximum flow between the source and the sink, finds a corresponding minimum cut, and marks all edges in this cut. The procedure stops when each vertex has been contained in at least one core.

To increase the number of marked edges, the iterative procedure is repeated $\mathcal{C}$ times. Afterwards, the natural-cut heuristic contracts all unmarked edges. The vertices in the resulting graph are called *fragments*. Note that each fragment represents a subgraph of G that was never cut and that each two adjacent fragments are separated by a natural cut. Typical parameter values for the filtering phase are $\alpha = 1$, $f = 10$, and $\mathcal{C} = 2$.

Assembly Phase. PUNCH runs a *greedy algorithm* to compute an initial partition of G from the fragment graph. It repeatedly contracts two adjacent vertices in the fragment graph until no contraction is possible without violating the upper bound U. The two vertices to be contracted next are picked based on a randomized *score function* [8]. Intuitively, the algorithm prefers small vertices that are tightly connected. The output of the greedy algorithm is a *contracted graph* H, where each vertex represents a cell in the partition of G.

The initial partition is then improved by an iterative *local search*. In each iteration, it picks two adjacent cells R, S at random from H. Let H_{RS} be the subgraph of H induced by R, S, and their neighbors in H and let G'_{RS} be obtained from H_{RS} by unpacking R and S into their constituent fragments (the neighbors remain contracted). Then, the greedy algorithm is run on G'_{RS}, outputting a contracted graph H'_{RS}. If H'_{RS} represents a better partition (i.e, one with a smaller cut size) than H_{RS}, we replace H_{RS} by H'_{RS} in the current solution H. The local search stops when each pair of adjacent cells in H has been considered φ times in succession without improving the current solution.

Since both the greedy algorithm and the local search are randomized, PUNCH uses a *multistart heuristic*, which runs the greedy algorithm followed by the local search multiple times on the fragment graph. After M candidate solutions have been generated, the best solution seen so far is returned. Alternatively, the candidate solutions generated by the multistart heuristic can be combined using an *evolutionary algorithm* [8]. Typical parameter values for the assembly phase are $\varphi = 16$ and $M = 9$.

2.3 Inertial Flow

Inertial Flow [26] is a partitioner tailored to road networks that exploits their geometric embedding. It has been applied [9, 4] to the partition-based shortest-path algorithms CRP [7] and CCH [10]. Its core algorithm bisects a graph $G = (V, E)$ with an embedding $\sigma : V \to \mathbb{R}^2$ into two balanced parts as follows:

(1) Pick a line ℓ with direction $d \in \mathbb{R}^2$.
(2) Project each point $\sigma(v)$, $v \in V$, orthogonally onto ℓ.
(3) Sort the vertices by their occurrence on ℓ.
(4) Determine a maximum flow between the first $\lfloor b|V| \rfloor$ vertices and the last $\lfloor b|V| \rfloor$ vertices.
(5) Find a corresponding minimum cut.

A typical parameter value is $b = 0.25$.

To find a partition of G with maximum cell size U, Inertial Flow recursively bisects G until the resulting parts have a size of at most U. For each bisection, Inertial Flow runs the core algorithm multiple times with parameter d set to $(0, 1)$, $(1, 0)$, $(1, 1)$, and $(-1, 1)$, respectively, and picks the smallest cut among those.

Inertial Flow computes partitions of reasonable quality. To improve the quality, Inertial Flow can be used to produce a partition with at most U/f vertices per cell, where $f > 1$ is a parameter. Contracting the edges within each cell yields a fragment graph. The fragments can then be combined as in the assembly phase of PUNCH. A typical parameter is $f = 32$.

3 Our Approach

This section discusses our approach to repartition road networks. Instead of partitioning the new graph from scratch, we start from the given partition, incorporate the vertices V^+, and repair and reoptimize the partition. We assume there are stable identifiers associated with the vertices in both graphs that allow us to map vertices in the old and new graph to

each other. Both OpenStreetMap and the proprietary data we are aware of provide such identifiers in the form of 64-bit integers. In case there are no stable identifiers available, we can heuristically map the vertices using, for example, their coordinates.

Our approach starts by mapping the partition $\mathcal{P}$ of the old graph G to the new graph $\bar{G}$. More precisely, each vertex $v \in V \cap \bar{V}$ inherits its cell identifiers from $\mathcal{P}$, i.e., we set $\bar{c}^l(v) = c^l(v)$ for all levels l. The vertices V^+ are not assigned to a cell on any level. After a quick preprocessing step (Section 3.1), we consider each cell $\bar{C}_i^l$ in the partition in descending level order, starting with the single cell on level $L+1$, which contains the entire new graph. Each cell $\bar{C}_i^l$, which induces the subgraph $\bar{G}\left[\bar{C}_i^l\right]$ of $\bar{G}$, is processed in two phases. The first phase assigns each vertex $v \in V^+$ to an existing cell on level $l-1$ (Section 3.2). The second phase repairs and reoptimizes the partition (Section 3.3).

We handle cells on the same level in parallel if multiple CPU cores are available. Moreover, if there is no change within a cell, we skip its subcells on all levels below.

3.1 Detecting Tiny Components

For several reasons (including data errors), there can be distinct components consisting of only a few vertices in the old graph that are connected to their neighborhood in the new graph. For example, consider a newly constructed road. In the old graph, it could still be under construction and not connected to the main network, and therefore a distinct component of its own. A partition could assign this component and its neighborhood to different cells, since there are no cut edges between them. However, connecting the new road to the main network leads to cut edges that are often unsuitably chosen. Hence, before the first cell assignment phase, we reset all cell identifiers for each vertex that belongs to a *tiny component* in the old but not in the new graph. We want such vertices to inherit the cell identifiers of their neighborhood rather than keeping their old identifiers. Formally, a tiny component is a connected component with a size that is below a given threshold.

3.2 Cell Assignment

Executing the first phase on cell $\bar{C}_i^l$ assigns each vertex $v \in V^+ \cap \bar{C}_i^l$ to a level-$(l-1)$ cell based on the cells of its neighbors. This phase resembles the label propagation algorithm [22] for clustering (evolving) networks. Each vertex is assigned to the cell to which the majority of its neighbors belong, with ties broken uniformly at random. We perform this process iteratively, where at every step, one vertex updates its cell. Note that the first assignment of a vertex is not necessarily final. Therefore, the process continues until no vertex $v \in V^+ \cap \bar{C}_i^l$ changes its cell anymore. Convergence is guaranteed since we move a vertex v from C_i to C_j only if $N_{C_j}(v)$ is *strictly* greater than $N_{C_i}(v)$, where $N_C(v)$ is the number of neighbors in cell C. Hence, with every such move, the sum $\sum_v N_{c(v)}(v)$ increases by $2(N_{C_j}(v) - N_{C_i}(v)) \geq 2$ (note that $N_{c(u)}(u)$ changes not only for $u = v$ but also for each neighbor u of v in $C_i \cup C_j$, causing the factor of 2). As the sum cannot exceed $\sum_v \deg(v)$, the process eventually stops.

To implement this approach efficiently, we keep track of the next vertex to assign with a min-heap, initialized with all new vertices adjacent to at least one old vertex. Every time we assign a vertex to a different cell, we insert its neighbors in V^+ into the min-heap. The priority of a vertex v is given by $key(v) = N_\perp(v) + N_2(v) - N_1(v)$, where $N_i(v)$ is the number of neighbors in the i-th most common neighboring cell (ties broken uniformly at random), and $N_\perp(v)$ is the number of as-yet-unassigned neighbors. The intuition here is that all assignments are both final and unambiguous as long as we only extract vertices v with $key(v) < 0$. This is easy to verify by induction on the number of delete-min operations.

When $key(v) = 0$, the assignment is not unambiguous but still final. Therefore, the choice of priorities ensures that we start with as many final assignments as possible, and thus reduces the number of cell corrections and the time to converge. Note that vertices unreachable from any vertex $v \in (V \setminus V^-) \cap \bar{C}_i^l$ remain unassigned after this phase. These vertices will be assigned to cells during the second phase that repairs and reoptimizes the partition.

To process the single cell on level $L + 1$, we must run cell assignment on the full input graph. For each cell C on all levels below, cell assignment must be run on $\bar{G}[C]$. For efficiency, we create a temporary copy of $\bar{G}[C]$ and run cell assignment on it. This simplifies cell assignment, allows us to use sequential local IDs, and improves locality.

3.3 Repair and Reoptimization

After the cell assignment phase, the partition of $\bar{G}\left[\bar{C}_i^l\right]$ is not necessarily feasible. First, there may be *oversized cells*, i.e., cells containing more than U^{l-1} vertices. Second, vertices unreachable from any vertex $v \in (V \setminus V^-) \cap \bar{C}_i^l$ have not been assigned to a cell yet. In the second phase, we repair both issues and locally reoptimize the partition.

Let K be a graph whose vertices are the cells in the partition. Each as-yet-unassigned vertex in $\bar{G}\left[\bar{C}_i^l\right]$ forms a cell of its own. The size of each vertex in K is the number of vertices in the corresponding cell. There is an edge $\{R, S\}$ in K if there is an edge $\{u, v\}$ in $\bar{G}\left[\bar{C}_i^l\right]$ with $u \in R$ and $v \in S$. Its weight is the total weight of the corresponding edges in $\bar{G}\left[\bar{C}_i^l\right]$.

Let K' be a graph obtained from K by *unpacking* some of the cells (we will discuss cell unpacking in detail in Section 3.4). In the following, we compute a partition of K', which can easily be transformed into a partition of $\bar{G}\left[\bar{C}_i^l\right]$. Note that to obtain a feasible partition, we must unpack at least each oversized cell. To increase the similarity between $\mathcal{P}$ and $\bar{\mathcal{P}}$, we can relax our definition of oversized cells, allowing cells to contain at most $g^{l-1}U^{l-1}$ vertices, where $g^{l-1} \geq 1$ is the *growth factor* on level $l - 1$.

To find an initial feasible partition, we run the greedy algorithm from PUNCH on K', yielding a contracted graph H. This partition is then reoptimized by running a variant of the local search from PUNCH on H. That is, we repeatedly pick two adjacent cells R, S at random from H, run the greedy algorithm on the subgraph of H induced by R, S, and their neighbors in H, and update the current solution H accordingly. However, while PUNCH unpacks R and S into their constituent fragments, we unpack them into the corresponding vertices in K'. Since both the greedy algorithm and the local search are randomized, we run them M times on K' and return the partition with the smallest cut size.

As already mentioned, we handle cells on a level $l \leq L$ in parallel. On such levels, we run a sequential version of the local search. On level $L + 1$ (where there is only a single cell), we parallelize the local search by trying multiple pairs of adjacent vertices concurrently.

3.4 Cell Unpacking

We considered three variants of cell unpacking, inspired by PUNCH, which differ in how much they unpack. See Figure 2 for an illustration. The simplest variant replaces the single vertex in K representing an oversized level-l cell $\bar{C}_i^l$ with one vertex for each level-$(l - d)$ subcell C_j^{l-d} with $C_j^{l-d} \cap \bar{C}_i^l \neq \emptyset$ (where $d \geq 1$ denotes the *descent step*) and one vertex for each vertex $v \in V^+$ that has been assigned to $\bar{C}_i^l$ during the first phase (cell assignment). The size of these vertices is the number of vertices in the new graph that they represent (possibly one). We call this variant *simple unpacking*.

The second variant, *neighbor unpacking*, gives the second phase more degrees of freedom to reoptimize the partition. Besides unpacking oversized cells, this variant also unpacks all cells that have a common boundary with an oversized cell.

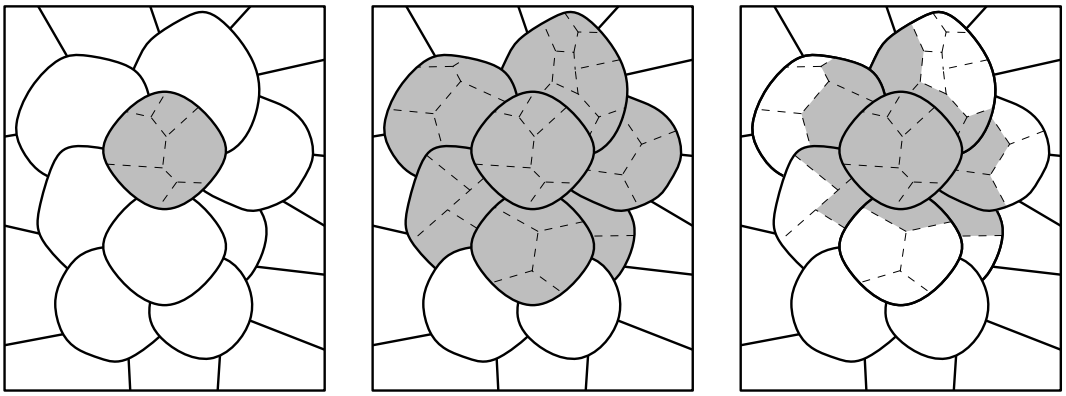

Figure 2 Unpacking an oversized cell during the reoptimization phase. Left: The simple variant unpacks only the oversized cell. Middle: Neighbor unpacking also unpacks all neighboring cells. Right: Partial unpacking unpacks the neighboring cells partially.

The third variant, which we call *partial unpacking*, unpacks each oversized level-l cell $\bar{C}_i^l$ fully and each level-l cell $\bar{C}_j^l$ that has a common boundary with $\bar{C}_i^l$ partially. More precisely, we replace the single vertex in K representing $\bar{C}_j^l$ with one vertex for each level-$(l-d)$ subcell C_k^{l-d} with $C_k^{l-d} \cap \bar{C}_j^l \neq \emptyset$ that directly borders on $\bar{C}_i^l$, one vertex for each vertex $v \in V^+$ that has been assigned to $\bar{C}_j^l$ during the first phase, and one vertex representing the remaining level-$(l-d)$ subcells of $\bar{C}_j^l$. Again, the size of each vertex is the number of vertices in the new graph that are represented by the vertex.

4 Experiments

4.1 Implementation and System

Both the partitioning and repartitioning algorithms are implemented in C++11 and were compiled with GCC 4.8.5 on a system running CentOS 7.7. For parsing the instances we use the RoutingKIT library [10]. The machine has 2 NUMA nodes, each equipped with a 10 core/20 threads Intel®Xeon®CPU E5-2640 v4 clocked at 2.40GHz with 2.5 MiB L2 and 25 MiB L3 cache. It has 192 GiB of DDR4-2400 RAM.

4.2 Instances

We evaluate and compare our algorithm with the full partitioning algorithm (FP) on road networks extracted from OpenStreetMap (`https://www.openstreetmap.org`). Our FP algorithm uses Inertial Flow to find a starting solution and an assembly phase similar to the one from PUNCH to optimize it. OpenStreetMap's contributors edit the map regularly which enables us to test our repartitioning algorithm on snapshots taken at different dates of the same cutout. We test our algorithms on the Australia, South America, North America and Europe instances available from GeoFabrik (`https://download.geofabrik.de/index.html`). Evaluation is performed between snapshots that are one year (1/1/2018 - 1/1/2019) and one month apart (10/1/2019 - 11/1/2019). Shorter time periods (e.g. a week) differ less and thus repartitioning performs at least as good as it does on monthly and yearly instances. More detailed information on the instances can be found in Table 1. For the remainder of this section, we reference the graph pairs of the time frames by an M and a Y suffix for the monthly and yearly instances respectively (e.g. $NorthAmerica_Y$ for the North America graph pair on 1/1/2018 and 1/1/2019).

Table 1 Instances and their properties. Absolute numbers in millions, relative numbers and vertex churn (VC) in percent. VC is defined as the ratio of $\frac{|V^+ \cup V^-|}{|V|}$.

Instance [1]	1/1/2018		1/1/2019			10/1/2019		11/1/2019		
	$\lvert V_{18}\rvert$	$\lvert E_{18}\rvert$	$\frac{\lvert V_{19}\rvert}{\lvert V_{18}\rvert}$	$\frac{\lvert E_{19}\rvert}{\lvert E_{18}\rvert}$	VC_Y	$\lvert V_{10}\rvert$	$\lvert E_{10}\rvert$	$\frac{\lvert V_{11}\rvert}{\lvert V_{10}\rvert}$	$\frac{\lvert E_{11}\rvert}{\lvert E_{10}\rvert}$	VC_M
Australia	1.23	2.85	7.83	6.63	11.33	1.41	3.18	0.45	0.41	0.62
N. America	24.90	61.38	1.27	0.98	6.84	26.05	63.94	0.28	0.26	0.99
Europe	30.55	71.46	3.44	3.05	7.12	32.69	76.00	0.54	0.52	1.00

1) Downloaded from `https://download.geofabrik.de` on 12/16/2019.

Table 2 Algorithm quality and performance on $Australia_Y$. Best values in bold.

Algorithm	CutSize [%]	S^1 [%]	S^2 [%]	S^3 [%]	S^4 [%]	S^5 [%]	Runtime [s]
SU	2.00	**79.53**	**71.18**	**65.15**	**57.19**	**49.78**	58.57
NU	**0.40**	68.05	63.57	58.25	50.64	22.11	98.06
PU	2.00	**79.53**	**71.18**	**65.15**	**57.19**	**49.78**	**58.35**

4.3 Parameters

Algorithms. We start with a comparison of our repartitioning algorithm variants SU (simple unpacking), NU (neighbor unpacking) and PU (partial unpacking) on the $Australia_Y$ instance with a cell growth of 0% and a descent step $d = 1$ (cf. Section 3.3) in Table 2. Using a larger d does not result in better quality or similarity based on our experiments. Simple unpacking and partial unpacking perform almost identically, both increasing the overall cut size by 2% compared to the result of the full partitioning algorithm. In our experiments we found that PU often has slightly worse similarity on the two lowest levels and identical similarity on the higher levels compared to NU while runtimes are comparable. The neighbor unpacking approach considers even more cells than PU for distributing unassigned vertices which results in smaller cuts at the cost of a reduced similarity and higher runtimes. Other instances produce similar results. Based on this evaluation, we focus on the simple unpacking approach a descent step $d = 1$ as it produces partitions of good quality and similarity with reasonable runtimes. In the remainder of this section, we use RP to denote our repartitioning algorithm using simple unpacking and FP to denote the full partitioning (from scratch) algorithm. The level-dependent parameters for both algorithms can be found in Table 3. We use the default parameters for our FP algorithm, whereas we reduce ϕ_{RP} and M_{RP} on the higher levels. Running more local searches and producing more candidates on these levels reduces similarity since the searches optimize cut size, not similarity and it increases the runtime.

Cell Growth. When new vertices are added to the graph, some cells have to be split in order to hold the size constraint on the cell size. However, most often these additional vertices do not affect the boundary of a cell but are contained in it. So instead of splitting cells, increasing the cut size and decreasing similarity, it is better to allow some cell growth in order to improve similarity. We evaluate the effect of cell growth on $Australia_Y$ and $Australia_M$ in Table 4. As expected, the similarity increases on all levels with higher cell growth at the cost of a more imbalanced partition - some cells utilizing all the allowed growth. The similarity change is most pronounced on the highest level, on the lowest level the change

Table 3 Common partitioning parameters per level of FP and RP.

Level	U	f	φ_{FP}	φ_{RP}	M_{FP}	M_{RP}
1	25	16	9	9	3	3
2	200	16	9	9	3	3
3	1600	32	16	9	4	3
4	12800	32	16	9	4	3
5	102400	32	32	9	6	3
6	819200	32	32	9	6	3
7	6553600	32	32	9	16	3

Table 4 Impact of allowing cell growth when running RP on $Australia_Y$ and $Australia_M$.

	Oversized Cells [%]		S^L [%]		Runtime [s]	
Cell Growth [%]	Year	Month	Year	Month	Year	Month
0	0.00	0.00	49.78	77.00	58.57	2.26
1	0.86	0.66	37.24	97.60	63.17	2.05
5	13.25	1.24	61.62	97.60	39.16	1.89
10	19.72	1.70	75.86	94.14	20.39	1.82

is less than 2%. There are now oversized cells that exceed the maximum cell size on each level. In the table we report the ratio of the total number of oversized cells over the total number of cells over all levels. The runtime of RP decreases with higher cell growth because the local optimizer has less work to do. While the imbalance introduced for the $Australia_M$ instance is always below 2%, it is significantly higher for $Australia_Y$ which is due to the fact that the latter instance has much more churn in both vertices and edges which makes it harder to repair the partition.

4.4 Comparison with Full Partitioning

Quality and Performance over Time. The more a graph churns over time, the harder it gets to keep the partition stable which is reflected in increased runtimes of our algorithm. Figures 3 and 4 show the effect of increased churn on the quality and runtime on monthly North America snapshots from 02/01/2018 to 12/01/2018 with 01/01/2018 used as the baseline partition that we want to keep stable. Vertex churn starts at 0.7% for 02/01/2018 and increases strictly monotonously to 6.1% for 12/01/2018. We report the similarity S^1, the total relative amount of oversized cells (over all levels) and the total runtime of RP with cell growth parameters 0%, 5% and 10%. For comparison, we include the FP algorithm with a cell growth of 0%. For FP, cell growth means increasing the maximum cell size U on each level. Higher cell growths for FP do not change the runtime significantly and lead to worse similarity as FP optimizes cut size and not similarity, so we exclude them in the figure. Similarity on higher levels follows the same trend as the similarity on level one, just slightly lower as is the case in our other experiments. A cell growth of 10% results in the best similarity values but there is never more than 1% difference between the different cell growth configurations. In contrast, the similarity of FP is much worse as FP does not optimize this measure. In terms of the partition quality, we compare cut size increases over the partition obtained by FP with the same amount of cell growth and notice that the

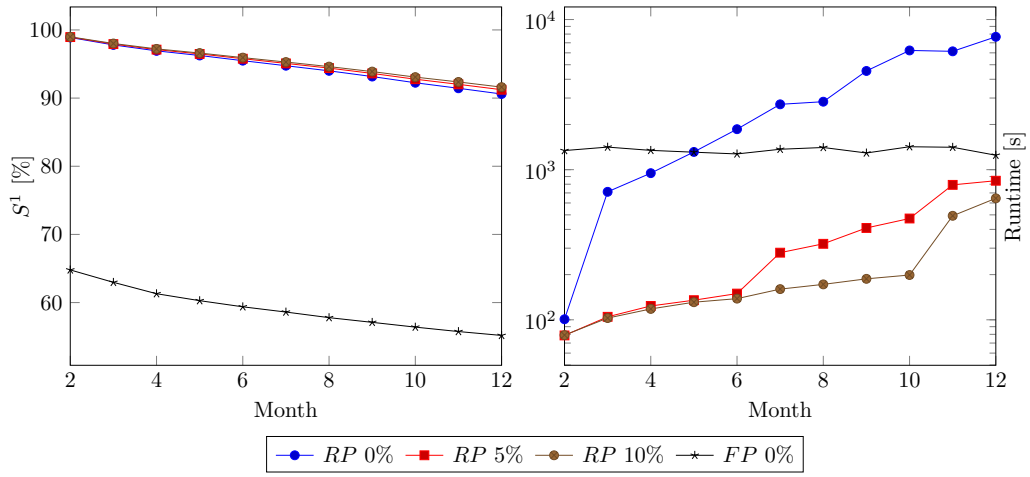

Figure 3 Similarity and runtime comparison of RP and FP for different cell growths on the monthly North America graphs between 02/01/2018 and 12/01/2018 with 01/01/2018 as the baseline partition to be kept stable.

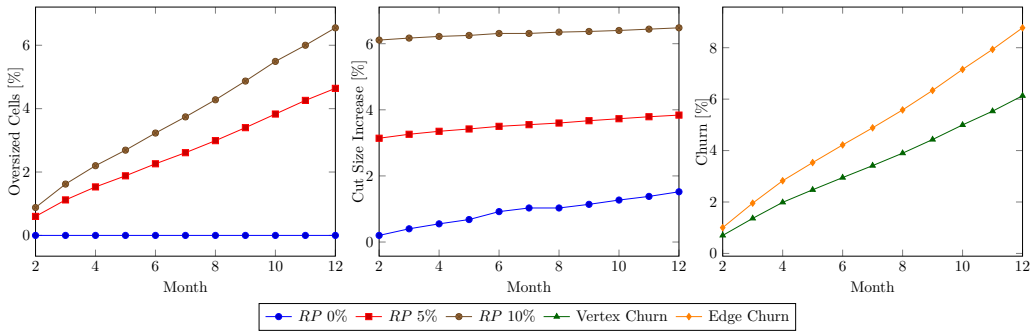

Figure 4 Oversized cells, cut size increase of RP for difference cell growths and graph churn on the monthly North America graphs between 02/01/2018 and 12/01/2018 with 01/01/2018 as the baseline partition to be kept stable. Cut size is compared to FP with the same cell growth.

cut size increases by roughly 1%, 3% and 6% for the respective cell growths and does not significantly increase with more churn. This can be explained by the fact that our algorithm mainly optimizes similarity to the input partition whereas allowing FP a higher imbalance can lead to different (smaller) cuts. As expected, the ratio of all oversized cells compared to the number of cells increases for a cell growth greater than 0% but stays within reasonable limits for the purpose of road network partitioning. The runtime of RP with 0% cell growth is comparable to the higher cell growths for the first month but increases sharply starting with the third month. Allowing cell growths of 5% and 10% yield similar running times up to month 6. Starting with month 7, however, RP with cell growth 5% has a significantly higher runtime, while the vertex/edge churn does not have any significant increase during these months. A possible explanation might be the merging of small connected components with bigger ones. In this case, a larger cell growth often allows to assign all vertices of the previously connected component to the now neighboring cell, improving the runtime and retaining the similarity.

Table 5 Comparison of RP and FP, both with 5% cell growth, on the monthly instances.

Instance	CutSize [%]	S^1 [%]		S^L [%]		Runtime [s]	
	RP	RP	FP	RP	FP	RP	FP
$Australia_M$	3.09	98.87	64.57	97.60	39.27	2.01	24.27
$NAmerica_M$	3.11	98.64	64.01	87.16	31.07	144.26	1390.57
$Europe_M$	3.24	98.35	62.70	94.70	50.73	197.90	1851.13

Table 6 Comparison of RP and FP, both with 20% cell growth, on the yearly instances.

Instance	CutSize [%]	S^1 [%]		S^L [%]		Runtime [s]	
	RP	RP	FP	RP	FP	RP	FP
$Australia_Y$	10.91	85.16	49.60	75.49	23.33	3.50	22.08
$NAmerica_Y$	13.14	91.04	54.55	62.46	32.53	356.81	1459.72
$Europe_Y$	12.83	90.41	52.98	73.64	46.03	251.47	1880.93

Quality and Performance on Monthly Instances. Based on the results of our monthly RP evaluation over the course of a year in the previous paragraph, we select a cell growth of 5% for the comparison with FP on the monthly instances. For FP, we use a cell growth of 0% for comparing similarity for best FP results and a cell growth of 5% for a fair comparison of cut sizes. The result of that evaluation can be found in Table 5. The similarity to the previous partition is always higher than 98% on the lowest level compared to about 67% for FP. RP is able to maintain high similarity values on higher levels as well. The cut size is no more than about 3% higher compared to FP and the runtime is up to a factor of 12 lower.

Quality and Performance on Yearly Instances. We also include quality and performance figures for the yearly instances where we select a cell growth of 20% to maximize similarity. For FP, we chose the same testing methodology as in the monthly comparison in terms of cell growth. While our algorithm is able to maintain good similarity values of 85% or higher on the lowest level, similarity decreases more drastically compared to the monthly instances. The FP algorithm only achieves up to 54% similarity on the lowest level and higher levels even worse. In terms of cut size, it shows the limitations of trying to keep a partition stable for a full year with good similarity as our algorithm produces cuts that are overall up to 14% higher compared to FP.

5 Conclusion

We studied the stable graph partitioning problem in road networks. We showed how to keep a graph partition stable on OpenStreetMap road networks over time. A nice benefit of our approach is a reduction of an order of magnitude in runtime compared to partitioning the networks from scratch. Regarding future work, we are interested in dealing with larger churn in the graph, like adding large new regions to the input. One possible approach here might be to find natural cuts in these new regions and then add the fragments to our scheme. Finally, we are interested in studying other types of evolving networks.

References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative routes in road networks. *ACM Journal of Experimental Algorithmics*, 18(1):1.3:1–1.3:17, 2013. doi:10.1145/2444016.2444019.

2 Reinhard Bauer and Daniel Delling. SHARC: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14:2.4:1–2.4:29, 2009. doi:10.1145/1498698.1537599.

3 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3:1–2.3:31, 2010. doi:10.1145/1671970.1671976.

4 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-time traffic assignment using engineered customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 24(2):2.4:1–2.4:28, 2019. doi:10.1145/3362693.

5 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. Springer, 2016. doi:10.1007/978-3-319-49487-6_4.

6 Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. doi:10.1016/j.jpdc.2012.02.007.

7 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/trsc.2014.0579.

8 Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato F. Werneck. Graph partitioning with natural cuts. In *25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011. doi:10.1109/IPDPS.2011.108.

9 Daniel Delling, Dennis Schieferdecker, and Christian Sommer. Traffic-aware routing in road networks. In *34th IEEE International Conference on Data Engineering (ICDE'18)*, pages 1543–1548. IEEE Computer Society, 2018. doi:10.1109/ICDE.2018.00172.

10 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, 2016. doi:10.1145/2886843.

11 Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):1–20, 2019. doi:10.3390/a12090196.

12 Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM Journal of Experimental Algorithmics*, 23(1):1.2:1–1.2:34, 2018. doi:10.1145/3173045.

13 Bruce Hendrickson, Robert W. Leland, and Rafael Van Driessche. Enhancing data locality by using terminal propagation. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences (HICSS'96)*, pages 565–574. IEEE Computer Society, 1996. doi:10.1109/HICSS.1996.495507.

14 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.

15 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13:2.5:1–2.5:26, 2008. doi:10.1145/1412228.1412239.

16 George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. doi:10.1137/S1064827595287997.

17 Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 19–39. American Mathematical Society, 2009.

18 Burkhard Monien and Stefan Schamberger. Graph partitioning with the party library: Helpful-sets in practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004. doi:10.1109/SBAC-PAD.2004.18.

19 Leonid Oliker and Rupak Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998. doi:10.1006/jpdc.1998.1469.

20 François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather M. Liddell, Adrian Colbrook, Louis O. Hertzberger, and Peter M. A. Sloot, editors, *Proceedings of the 4th International Conference and Exhibition on High-Performance Computing and Networking (HPCN'96)*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996. doi:10.1007/3-540-61142-8_588.

21 Eddy Pramono, Horst D. Simon, and Andrew Sohn. Dynamic load balancing for finite element calculations on parallel computers. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia Torczon, and Layne T. Watson, editors, *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 599–604. SIAM, 1995.

22 Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), 2007. doi:10.1103/PhysRevE.76.036106.

23 Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.

24 Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In David A. Bader and Petra Mutzel, editors, *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 16–29. SIAM, 2012. doi:10.1137/1.9781611972924.2.

25 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

26 Aaron Schild and Christian Sommer. On balanced separators in road networks. In Evripidis Bampis, editor, *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015. doi:10.1007/978-3-319-20086-6_22.

27 Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997. doi:10.1006/jpdc.1997.1410.

28 Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In Louis H. Turcotte, editor, *Proceedings of the 13th ACM/IEEE Supercomputing Conference (SC'00)*, pages 1–11. IEEE Computer Society, 2000. doi:10.1109/SC.2000.10035.

29 Kirk Schloegel, George Karypis, and Vipin Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001. doi:10.1109/71.926167.

30 Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5:1–23, 2000. doi:10.1145/351827.384254.

31 Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Using multi-level graphs for timetable information in railway systems. In David M. Mount and Clifford Stein, editors, *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002. doi:10.1007/3-540-45643-0_4.

32 Andrew Sohn and Horst D. Simon. JOVE: A dynamic load balancing framework for adaptive computations on an SP-2 distributed-memory multiprocessor. Technical Report 94-60, New Jersey Institute of Technology, Department of Computer and Information Science, 1994.

33 Chris Walshaw. Variable partition inertia: Graph repartitioning and load balancing for adaptive meshes. In Manish Parashar and Xiaolin Li, editors, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, pages 357–380. John Wiley & Sons, 2009. doi:10.1002/9780470558027.ch17.

34 Chris Walshaw and Martin Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995. doi:10.1002/cpe.4330070103.

35 Chris Walshaw, Mark Cross, and Martin G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997. doi:10.1006/jpdc.1997.1407.

36 Pei-bai Zhou. *Numerical Analysis of Electromagnetic Fields*. Electric Energy Systems and Engineering Series. Springer, 1993. doi:10.1007/978-3-642-50319-1.

37 Olgierd C. Zienkiewicz, Robert L. Taylor, and David D. Fox. *The Finite Element Method for Solid and Structural Mechanics*. Butterworth-Heinemann, 2014. doi:10.1016/C2009-0-26332-X.

38 Olgierd C. Zienkiewicz, Robert L. Taylor, and Perumal Nithiarasu. *The Finite Element Method for Fluid Dynamics*. Butterworth-Heinemann, 2014. doi:10.1016/C2009-0-26328-8.