Graph Bisection with Pareto-Optimization*

Michael Hamann, Ben Strasser

Institute of Theoretical Informatics,
Karlsruhe Institute of Technology,
P.O. Box 6980, 76128 Karlsruhe, Germany.

michael.hamann@kit.edu, strasser@kit.edu

Abstract

We introduce FlowCutter, a novel algorithm to compute a set of edge cuts or node separators that optimize cut size and balance in the Pareto-sense. Our core algorithm heuristically solves the balanced connected *st*-edge-cut problem, where two given nodes s and t must be separated by removing edges to obtain two connected parts. Using the core algorithm we build variants that compute node separators and are independent of s and t. Using the computed Pareto-set we can identify cuts with a particularly good trade-off between cut size and balance that can be used to compute contraction and minimum fill-in orders, which can be used in Customizable Contraction Hierarchies (CCH), a speed-up technique for shortest path computations. Our core algorithm runs in $O(cm)$ time where m is the number of edges and c the cut size. This makes it well-suited for large graphs with small cuts, such as road graphs, which are our primary application. For road graphs we present an extensive experimental study demonstrating that FlowCutter outperforms the current state of the art both in terms of cut sizes as well as CCH performance.

1 Introduction

Cutting a graph into two pieces of roughly the same size along a small cut is a fundamental and NP-hard [14] graph problem that has received a lot of attention [2, 8, 18, 21, 23] and has many applications. The application motivating our research is accelerating shortest path computations on roads [3, 7, 10, 16, 24], but in the extended version of our paper [15] we also present bisection experiments on non-road graphs. Dijkstra's algorithm [12] solves the shortest path problem in near-linear time. However, this is not fast enough if the graph consists of a whole continent's road network. Acceleration algorithms exploit that road networks rarely change and compute auxiliary data in a *preprocessing phase*. This data is independent of the path's endpoints and can therefore be reused for many shortest path computations. Often the auxiliary data consists of cuts. The basic idea is: Given a graph G and a cut C the algorithms precompute for every node how to get to every edge/node in C. To compute a path the algorithms first determine whether the endpoints are on opposite sides of C or not. If they are on opposite sides then the algorithms only need to assemble the precomputed paths towards C and pick the best one. If they are on the same side then the graph search can be pruned at C. This halves the graph that needs to be searched. As half a continent is still large the idea is applied recursively. The effectiveness of these techniques crucially depends on the size of the cuts found. Fortunately road graphs have small cuts because of geographical features such as rivers or mountains. Previous work has coined the term *natural cuts* for this phenomenon [8]. However, identifying these natural cuts is a difficult problem. Fortunately, as roads change only slowly, preprocessing running times are significantly less important than cut quality. One of these preprocessing-based techniques are *Customizable Contraction Hierarchies* (CCH) [10]. We demonstrate the performance of our algorithms using CCH. The CCH-auxiliary data is tightly coupled with tree-decompositions [4] and minimum fill-in orders. Our algorithms are therefore also applicable in that domain.

Graph partitioning software used for road graphs include *KaHip* [21], *Metis* [18], *Inertial Flow* [23], or *PUNCH* [8]. We experimentally compare FlowCutter with the first three as we unfortunately have no implementation of PUNCH. Further Microsoft holds a PUNCH-patent which restricts commercial applica-

*Partial support by DFG grant WA654/19-1 and Google Focused Research Award.

tions. The cut problem is formalized as a bicriteria problem optimizing the cut size and the imbalance. The *imbalance* measures how much the sizes of both sides differ and is small if the sides are balanced. The standard approach is to bound the imbalance and minimize the cut size. However, this approach has some shortcomings. Consider a graph with a million nodes and set the max imbalance to 1%. An algorithm finds a cut C_1 with 180 edges and 0.9% imbalance. Is this a good cut? It seems good as 180 is small compared to the node count. However, we would come to a different conclusion, if we knew that a cut C_2 with 90 edges and 1.1% imbalance existed. In our application — shortest paths — moving a few nodes to the other side of a cut is no problem. However, halving the cut size has a huge impact. The cut C_2 is thus clearly superior. Further assume that a third cut C_3 with 180 edges and 0.7% imbalance existed. C_3 dominates C_1 in both criteria. However, both are equivalent with respect to the standard problem formulation and thus a tool is not required to output C_3 instead of C_1. To overcome these problems our approach computes a set of cuts that optimize cut size and imbalance in the Pareto sense. A further significant shortcoming of the state-of-the-art partitioners, with the exception of Inertial Flow, is that they were designed for small imbalances. Common benchmarks, such as the one maintained by Chris Walshaw [25], only include test cases with imbalances up to 5%. However, for our application imbalances of 50% are fine. For such high imbalances unexpected things happen with the standard software, such as increasing the allowed imbalance can increase the achieved cut sizes.

Contribution. We introduce FlowCutter, a graph bisection algorithm that optimizes cut size and imbalance in the Pareto sense. The core FlowCutter algorithm aims to solve the balanced edge-*st*-cut graph bisection problem with connected sides. Using this core as subroutine we design algorithms to solve the node separator and non-*st* variants. Using these we design a nested dissection-based algorithm to compute contraction node orders as needed by Customizable Contraction Hierarchies (CCH). These orders are also called minimum fill-in orders or elimination orders and can be used to compute good tree-decompositions. We prove that the core algorithm's running time is in $O(cm)$ where m is the edge count and c the cut size. We show in an extensive experimental evaluation that this is a perfect fit for road graphs that are large in terms of edge count but small in terms of cut size.

Outline. We define our terminology and introduce related concepts in the preliminaries. The next section introduces the core idea of the *st*-bisection algorithm. In the following section we describe the piercing heuristic, a subroutine needed in the core algorithm. In the section afterwards we describe extensions of the core algorithms: general bisection, node bisection, and computing contraction orders. Finally, we present an experimental evaluation with a comparison against the current state of the art. The long version [15] contains further experiments including experiments on non-road graphs.

2 Preliminaries

A *graph* is denoted by $G = (V, A)$ with *node set* V and *arc set* A. We set $n := |V|$ and $m := |A|$. As input graphs we consider undirected, simple graphs which we interpret as symmetric directed graphs. Our core algorithm also works on directed graphs which is important for the computation of node separators. An *edge* is a pair of forward and backward arcs in a symmetric graph. The *out-degree* $d_o(x)$ of a node x is is the number of outgoing arcs. Similarly the *in-degree* $d_i(x)$ is the number of incoming arcs. In symmetric graphs we refer to the value as *degree* $d(x)$ of x, as $d_i(x) = d_o(x)$. A *degree-2-chain* is a sequence of adjacent nodes $x, y_1 \dots y_k, z$ in a symmetric graph such that $k \geq 1$, $d(x) \neq 2$, $d(y) \neq 2$, and $\forall i : d(y_i) = 2$. An *xy-path* P is a list $(x, p_1), (p_1, p_2) \dots (p_i, y)$ of adjacent arcs and i is P's length. The *distance* $\text{dist}(x, y)$ is defined as the minimum length over all xy-paths.

Cuts & Separators. A *cut* (V_1, V_2) is a partition of V into two disjoint sets V_1 and V_2 such that $V = V_1 \cup V_2$. An arc (x, y) with $x \in V_1$ and $y \in V_2$ is called *cut-arc*. The *size of a cut* is the number of cut-arcs. A min-cut is a cut of minimum size. A *separator* (V_1, V_2, Q) is a partition of V into three disjoint sets V_1, V_2 and Q such that $V = V_1 \cup V_2 \cup Q$. No arc connecting V_1 and V_2 must exist. The cardinality of Q is the *separator's size*. The *imbalance* $\epsilon \in [0, 1]$ of a cut is defined as the smallest number such that $\max\{|V_1|, |V_2|\} \leq \lceil (1 + \epsilon)n/2 \rceil$. The imbalance of a separator is defined analogously. However, note that because the separator itself may contain nodes it is possible that for separators the minimum ϵ is smaller than 0. A negative ϵ is not possible for edge cuts. A *ST-cut/separator* is a cut/separator between two disjoint node sets S and T such that $S \subseteq V_1$ and $T \subseteq V_2$. If $S = \{s\}$ and $T = \{t\}$ we write *st-cut/separator*. The *expansion of a cut/separator* is the cut's size divided by $\min\{|V_1|, |V_2|\}$.

Pareto-Optimization & NP-hardness. Computing cuts (and separators) is inherently a bicriteria problem: We want to minimize the cut size and minimize the imbalance. A cut C_1 dominates a cut C_2 if C_1 is strictly better with respect to one criterion and no worse with respect to the other criterion. A cut that is not dominated by any other cut is *Pareto-optimal*. We

refer to the pair of imbalance and cut size of a Pareto-optimal cut as *Pareto-trade-off*. It is possible that several Pareto-optimal cuts exist with the same trade-off. We consider the problem of computing for every Pareto-trade-off one cut.

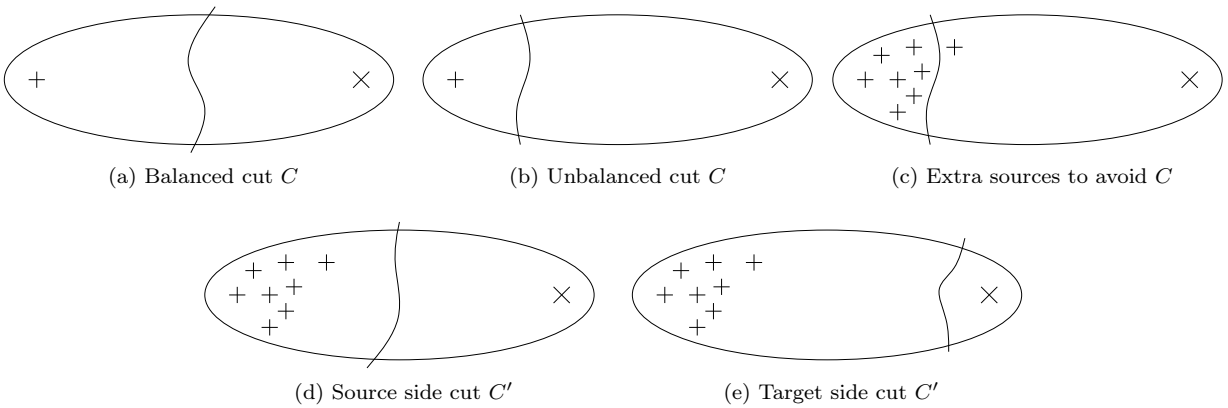
This is a departure from existing experimental papers [2, 6, 18, 21, 23, 25, 26] that consider the problem of finding a smallest cut subject to an imbalance bounded by an input parameter. Note that, given a cut for every Pareto-trade-off it is easy to find a smallest cut with a bounded imbalance. However, a cut with minimum size with an imbalance bounded by an input parameter is not necessarily Pareto-optimal: It is possible that a more balanced cut with the same size exists. Our problem setting is therefore a strict generalization of the problem setting considered in previous works.

The minimum cut problem disregarding the imbalance is polynomially solvable [13]. However, nearly all cut-problems that combine optimizing imbalance and cut size are NP-hard. Finding a minimum cut with $\epsilon = 0$ is NP-hard [14]. A *sparsest cut* is a cut that minimizes $\frac{c}{|V_1|\cdot|V_2|}$. Note that, a sparsest cut is Pareto-optimal. Finding a sparsest cut is NP-hard [19]. Even computing, for a fixed st-pair, a most balanced cut among all st-cuts of minimum size is already NP-hard [5]. Being able to compute a cut for every Pareto-trade-off efficiently would yield an efficient algorithm for all these NP-hard problems. Unless P=NP, we can therefore not hope to find an efficient algorithm that provably computes an optimal cut for every Pareto-trade-off. Our algorithm tries to heuristically compute in a single run a cut for every Pareto-trade-off.

Flows. Our method builds upon unit flows that are computed using augmenting paths [13, 1]. Formally, a flow is a function $f : A \to \{0, 1\}$. An arc a with $f(a) = 1$ is *saturated*. Denote by $p(x) = \sum_{(x,y)\in A} f(x, y)$ the *surplus of a node* x. A flow is valid with respect to a source set S and target set T if and only if: (i) flow may be created at sources, i.e., $\forall s \in S : p(s) \geq 0$, (ii) flow may be removed at targets, i.e., $\forall t \in T : p(t) \leq 0$, (iii) flow is conserved at all other nodes, i.e., $\forall x \in V\backslash(S\cup T) : p(x) = 0$, and (iv) flow does not flow in both directions, i.e., for all $(x, y) \in A$ such that $(y, x) \in A$ exists it holds that $f(x, y) = 0 \vee f(y, x) = 0$. The flow *intensity* is defined as the sum over all $f(x, y)$ for arcs (x, y) with $x \in S$ and $y \notin S$. The flow intensity is sometimes also called flow value. A *saturated path* $a_1, a_2 \ldots, a_i$ is a path such that $\exists i : f(a_i) = 1$. A node x is *source-reachable* if a non-saturated sx-path exists with $s \in S$. Similarly a node x is called *target-reachable* if a non-saturated xt-path exists with $t \in T$. We denote by S_R the set of all *source-reachable nodes* and by T_R the set of all *target reachable nodes*. In [13] it was shown that a flow is maximum if and only if no non-saturated st-path with $s \in S$ and $t \in T$ exists. If such a path exists then it is called *augmenting path*. The classic approach to computing max-flows consists of iteratively searching for augmenting paths. Our algorithm uses this approach. The minimum ST-cut size corresponds to the maximum ST-flow intensity. We define the *source-side cut* as $(S_R, V\backslash S_R)$ and the *target-side cut* as $(T_R, V\backslash T_R)$. Note that in general max-flows and min-cuts are not unique. However, the source-side and target-side cuts are.

Customizable Contraction Hierarchy (CCH) is an acceleration algorithm for shortest path computations. We only give a high-level overview, as we use CCH only to evaluate the quality of our cuts. No part of FlowCutter builds upon CCH. The details are in [10, 11]. The central operation is the node contraction: Contracting a node v consists of removing v and adding edges between all of v's unconnected neighbors. The input to CCH consists of a node *contraction order* along which the nodes are iteratively contracted. This yields a supergraph G' of the input graph. The weights of G' are computed using an algorithm that essentially enumerates all triangles in G' in the so-called *customization phase*. Note that contrary to the order computation, having a fast *customization phase* is useful as it allows us to incorporate changes to the weights quickly. Such changes could for example be caused by traffic congestion. CCH can also be used if several weights exist on the same road graph. Having regular cars and trucks is an example of such a situation. The CCH structure can be shared and does not have to be replicated for each weight. It is sufficient to replicate the weights. We therefore discern between memory that is independent of the weights and shared and memory that is needed per weight. Given the weights of G', the *shortest path query* consists of a bidirectional search in G' only following arcs (x, y) such that x appears before y in the order. The search space of a node z is the subgraph of G' that is reachable from z while only following such arcs. Smaller search spaces yield faster queries. Fewer triangles in G' yield a faster customization. Less arcs in G' result in less memory consumption. All these quality metrics depend on the contraction order, whose quality depends on the cuts used in its construction. Finding these cuts is where FlowCutter fits into the big picture.

Tree-Decompositions. The constructed supergraph G' is chordal, which is a graph class tightly coupled with tree-decomposition [4]. The maximum cliques of G', which can be efficiently identified in chordal graphs, correspond to the bags of a tree-decomposition. A corresponding tree backbone can be efficiently computed. The maximum clique size in G' is thus an upper bound to the treewidth of the input graph.



(a) Balanced cut C (b) Unbalanced cut C (c) Extra sources to avoid C

(d) Source side cut C' (e) Target side cut C'

Figure 1: The ellipse represents a graph and the curved lines are cuts. The "+"-signs represent source nodes and "×"-signs represent target nodes. The nodes s and t are represented by the "+" and "×" in Figure 1a and the equally positioned "+" and "×" in the other figures.

3 Core Algorithm

Our algorithm works by computing a sequence of st-min-cuts of increasing size. The more imbalanced cuts are computed first and are followed by more balanced ones. The cuts in this sequence form, after removing dominated ones, the heuristically approached Pareto-set. During its execution our algorithm maintains a maximum flow. With respect to this flow there is a source-side cut C_S and a target-side cut C_T. Our algorithm picks one of the two as the next cut C that it inserts into the set. After choosing C it modifies the set of source and target nodes and potentially augments the maintained flow. This results in a new pair of source-side and target-side cuts. Our algorithm picks C_S as C if there are less or equally many nodes on the source side of C_S than there are on the target side of C_T.

Consider the situation depicted in Figure 1. Initially s is the only source node and t is the only target node. Our algorithm starts by computing a maximum st-flow. If we are lucky and the cut C is perfectly balanced as in Figure 1a then our algorithm is finished. However, most of the time we are unlucky and we either have the situation depicted in Figure 1b where the source's side of C is too small or the analogous situation where the target's side of C is too small. Assume without loss of generality that the source's side is too small. Our algorithm now transforms non-source nodes into additional source nodes to invalidate C. To invalidate C our algorithm does two things: It marks all nodes on the source's side of C as source nodes and marks one node as source node on the target's side of C that is incident to a cut edge. This node on the target's side is called the *piercing node* and the corresponding cut arc is called *piercing arc*. The situation is illustrated in Figure 1c. All nodes on the source's side are marked as source node to assure that C' does not cut through the source's side. The piercing node is necessary to assure that $C' \neq C$. Choosing a good piercing arc is crucial for good quality. In this section we assume that we have a *piercing oracle* that determines the piercing arc given C in time linear in the size of C. In Section 4 we describe heuristics to implement such a piercing oracle.

To make progress we need that C' is non-dominated. As its size is a least the size of C, this is equivalent with C' being more balanced than C. However, we can only guarantee this if C' is, just as C, a source-side cut as in Figure 1d. If C' is a target-side cut as in Figure 1e then C' might have a worse balance than C. Luckily, as our algorithm progresses, either the target side will catch up with the balance of the source side or another source side cut is found. In both cases our algorithm eventually finds a cut with a better balance than C.

Our algorithm computes the st-min-cuts by finding max-flows and using the max-flow-min-cut duality [13]. Our algorithm assigns unit capacities to every edge and compute the flow by successively searching for augmenting paths. A core observation of our algorithm is that turning nodes into sources or targets never invalidates the flow. It is only possible that new augmenting paths are created increasing the maximum flow intensity. Given a set of nodes X we say that *forward growing* X consists of adding all nodes y to X for which a node $x \in X$ and a non-saturated xy-path exist. Analogously *backward growing* X consists of adding all nodes y for which a non-saturated yx-path exists. The growing operations are implemented using a graph traversal algorithm (such as a DFS or BFS) that only follows non-saturated arcs. The algorithm maintains besides the flow values four node sets: the

```
1  S ← {s}; T ← {t};
2  S_R ← S; T_R ← T;
3  forward-grow S_R; backward-grow T_R;
4  while S ∩ T = ∅ do
5      if S_R ∩ T_R ≠ ∅ then
6          augment flow by one;
7          S_R ← S; T_R ← T;
8          forward-grow S_R; backward-grow T_R;
9      else
10         if |S_R| ≤ |T_R| then
11             forward-grow S;
               // now S = S_R
12             output source side cut arcs;
13             x ← pierce node;
14             S ← S ∪ {x};
15             S_R ← S_R ∪ {x};
16             forward-grow S_R;
17         else
               // Analogous for target side
```

Figure 2: *st*-Bisection Algorithm

set of sources S, the set of targets T, the set source-reachable nodes S_R, and the set of target-reachable nodes T_R. Note that an augmenting path exists if and only if $S_R \cap T_R \neq \emptyset$. Initially we set $S = \{s\}$ and $T = \{t\}$. Our algorithm works in rounds. In every round it tests whether an augmenting path exists. If one exists the flow is augmented and S_R and T_R are recomputed. If no augmenting path exists then it must enlarge either S or T. This operation also yields the next cut. It then selects a piercing arc and grows S_R and T_R accordingly. The pseudo-code is in Figure 2.

Running Time. Assuming a piercing oracle with a running time linear in the current cut size, we can show that the algorithm has a running time in $O(cm)$ where c is the size of the most balanced cut found and m is the number of edges in the graph. The detailed argument requires a non-trivial amortized running time analysis and is provided below. However, the core argument is simple: All sets only grow unless we find an augmenting path. As each node can only be added once to each set, the running time between finding two augmenting paths is linear. In total we find c augmenting paths. The total time is thus in $O(cm)$.

The lines 1-3 have a running time in $O(m)$ and are therefore unproblematic. The condition in line 4 can be implemented in $O(1)$ as following: S and T only grow. We can therefore check when adding a node to one of the sets, whether it is contained in the other set. If this is the case we abort the loop. Outside of the true-branch of the if-statement in line 5 also S_R and T_R only grow. We can therefore use the same argument for the condition in line 5. Lines 6-8 need $O(m)$ running time each time they are executed. However, they are only executed when the flow is augmented. This happens c times. The total running time is thus in $O(cm)$. Showing that the running time of the lines 11-16 is amortized sub-linear is the complex part of the analysis. Implementing the growing operations in lines 11 and 16 the naive way needs linear running time and is therefore too slow. The naive approach looks at all internal nodes to determine all outgoing edges. These are needed to determine which are the non-saturated edges. However, either the sets only contain a single node x or they were generated by growing them and afterwards adding a single additional node y. In either case it is sufficient to look at the outgoing edges of x or y because all other outgoing edges must be saturated, as otherwise they would have been followed in a previous iteration. Outputting the cut in line 12 causes costs linear in the cut size. We account for these when calling the piercing oracle in line 13. However, it is non-trivial that we can list all edges in the cut in linear time. We do this by maintaining two additional edge sets C_S and C_T. The source side cut is in C_S and the target side cut is in C_T. We only describe how to maintain C_S. The algorithm for C_T is analogous. Each time we grow S and the graph search algorithm encounters a saturated edge e it adds e to C_S. Every cut edge is saturated and therefore the desired cut is a subset of C_S. As S never shrinks each edge can only be added at most once and therefore these additions have running time costs within $O(m)$. In line 12 it is possible that C_S contains edges that are saturated but not part of the cut. We filter those edges by iterating over all edges and removing those for which both end points are in S. As each edge can be removed at most once the removal costs are within $O(m)$. The remaining edges are the cut. We account for the running time needed to skip the cut edges during the filter step when calling the piercing oracle in line 13. The lines 14-15 have a constant running time. It remains to show that all the calls to the piercing oracle in line 13 in total do not need more than $O(cm)$ running time. The key observation here is that each time that the oracle is called it names a piercing arc e. The next time the oracle is called e is no longer part of the cut and therefore the oracle can no longer return e. Each arc is therefore only at most in one iteration the piercing arc. The oracle is therefore called at most m times. Each time it has a running time linear in the cut size. We can bound the cut size of each step by the final cut size c as the cut sizes only increases. The total running time spent in the piercing oracle is therefore bound by $O(cm)$.

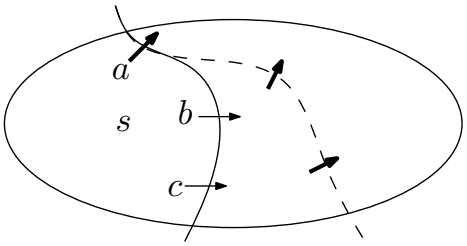


Figure 3: The curves represent cuts, the current one is solid. The arrows are cut-arcs, bold ones result in augmenting paths. The dashed cut is the next cut where piercing any arc results in an augmenting path.

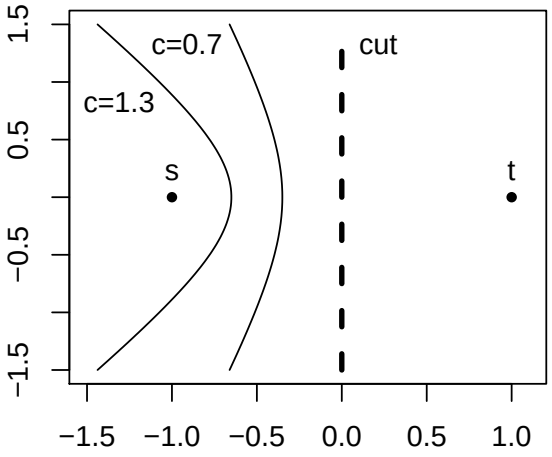


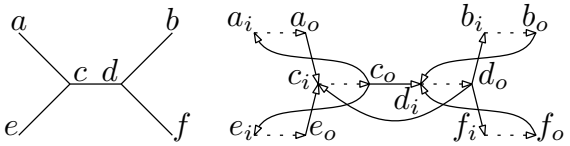
Figure 4: Geometric interpretation of the distance heuristic.

4 Pierce Heuristic

In this section we describe how we implement the piercing oracle used in the previous section. Given an unbalanced arc cut C the piercing oracle should select a piercing arc that is not part of the final balanced cut in at most $O(|C|)$ time. Piercing the source side and target side cuts and are analogous and we therefore only describe the procedure for the source side. Denote by $a = (q, p)$ the piercing arc with piercing node $p \notin S$.

Primary Heuristic: Avoid Augmenting Paths. The first heuristic consists of avoiding augmenting paths whenever possible. Piercing an arc a leads to an augmenting path if and only if $p \in T_R$, i.e., a non-saturated path from p to a target node exists. As our algorithm has computed T_R it can determine in constant time whether piercing an arc would increase the size of the next cut. The proposed heuristic consists of preferring edges with $p \notin T_R$ if possible. It is possible that none or multiple $p \notin T_R$ exist. In this case our algorithm employs a further heuristic to choose the piercing arc among them. However, note that the secondary heuristic is often only relevant in the case that none exists. Consider the situation depicted in Figure 3. Suppose for the argument that the target node is still far away and that the perfectly balanced cut is significantly larger. Our algorithm can choose between three piercing arcs a, b, and c. It will not pick a as this would increase the cut size. The question that remains is whether b or c is better. The answer is that it nearly never matters. Piercing b or c does not modify the flow and thus does not change which piercing arcs result in larger cuts. The algorithm will therefore eventually end up with the dashed cut independent of whether b or c is pierced. We know that the dashed cut has the same size as all cuts found between the current cut and the dashed cut. Further the dashed cut has the best balance among them and therefore dominates all of them. This means that most of the time our avoid-augmenting-paths heuristic does the right thing. However it is less effective when cuts approach perfect balance. The reason is that that the source and target sides meet. When approaching perfect balance our algorithm results in a race between source and target sides to claim the last nodes. Not the best side wins, but the first that gets there.

Secondary Heuristic: Distance-Based. Our algorithm picks a piercing arc such that $\text{dist}(p, t) - \text{dist}(s, p)$ is maximized, where s and t are the original source and target nodes. The $\text{dist}(p, t)$-term avoids that the source side cut and target side cut meet as nodes close to t are more likely to be close to the target side cut. Subtracting $\text{dist}(s, p)$ is motivated by the observation that s has a high likelihood of being positioned far away from the balanced cuts. A piercing node close to s is therefore likely on the same side as s. Our algorithm precomputes the distances from s and t to all nodes before the core algorithm is run. This allows it to evaluate $\text{dist}(p, t) - \text{dist}(s, p)$ in constant time inside the piercing oracle. The distance heuristic has a geometric interpretation as depicted in Figure 4. We interpret the distance as euclidean distance. If s and t are points in the plane then the set of points p for which $\|p - s\|_2 - \|p - t\|_2 = c$ holds for some constant c is one branch of a hyperbola. The figure depicts the branches for $c = 1.3$ and $c = 0.7$. The heuristic prefers piecing nodes on the $c = 1.3$-branch as it maximizes c. A consequence of this is that the heuristic works well if the desired cut follows roughly a line perpendicular to the line through s and t. This heuristic works on many graphs but there are instances where it breaks down such as cuts that follow a circle-like shape. Note that this geometric interpretation also works in higher-dimensional spaces.



(a) Input graph (b) Expanded graph

Figure 5: Expansion of an undirected graph G into a directed graph G'. The dotted arrows are internal arcs. The solid arrows are external arcs.

5 Extensions

Our base algorithm can be extended to compute general small cuts that are independent of an input st-pair, to compute node separators, and it can be used to compute contraction orders.

General Cuts. Our core algorithm computes balanced st-cuts. However, in many situations cuts independent of a specific st-pair are needed. This problem variant can be solved with high probability by running FlowCutter multiple times with st-pairs picked uniformly at random. Indeed, suppose that C is an Pareto-optimal cut such that the larger side has αn nodes (i.e. $\alpha = (\epsilon + 1)/2$) and q is the number of st-pairs. The probability that C separates a random st-pair is $2\alpha(1-\alpha)$. The success probability over all q st-pairs is thus $1 - (1 - 2\alpha(1-\alpha))^q$. For $\epsilon = 33\%$ and $q = 20$ the success probability is 99.99%. For larger α this rate decreases. However, it is still large enough for all practical purposes, as for $\alpha = 0.9$ (i.e. $\epsilon = 80\%$) and $q = 20$ the rate still is 98.11%. The number of st-pairs needed does not depend on the size of the graph nor on the cut size. If the instances are run one after another then the running time depends on the worst cut's size which may be more than c. We therefore run the instances simultaneously and stop once one instance has found a cut of size c. The running time is thus in $O(cm)$.

Note that this argumentation relies on the assumption that it is enough to find an st-pair that is separated. However, in practice the positions of s and t in their respective sides influence the performance of our piercing heuristic. As a result it is possible that in practice more st-pairs are needed than predicted by theory.

Node Separators. To compute contraction orders node separators are needed and not edge cuts. To achieve this we employ a standard construction to model node capacities in flow problems [1]. We transform the symmetric input graph $G = (V, A)$ into a directed expanded graph $G' = (V', A')$ and compute flows on G'. We expand G into G' as follows: For each node $x \in V$ there are two nodes x_i and x_o in V'. We refer to x_i as the *in-node* and to x_o as the *out-node* of x. There is an *internal arc* $(x_i, x_o) \in A'$ for every node $x \in V$. We further add for every arc $(x, y) \in A$ an *external arc* (x_o, y_i) to A'. The construction is illustrated in Figure 5. For a source-target pair s and t in G we run the core algorithm with source node s_o and target node t_i in G'. The algorithm computes a sequence of cuts in G'. Each of the cut arcs in G' corresponds to a separator node or a cut edge in G depending on whether the arc in G' is internal or external. From this mixed cut our algorithm derives a node separator by choosing for every cut edge in G the endpoint on the larger side. Unfortunately using this construction, it is possible that the graph is separated into more than two components, i.e., we can no longer guarantee that both sides are connected.

Contraction Orders. Using a nested dissection [20] variant our algorithm constructs contraction orders. It bisects G along a node separator Q into subgraphs G_1 and G_2. It recursively computes orders for G_1 and G_2. The order of G is the order of G_1 followed by the order of G_2 followed by the nodes in Q in an arbitrary order. Selecting Q is non-trivial. After some experimentation we went with the following heuristic: Pick a separator with minimum expansion and at most 60% imbalance. Note that we can abort FlowCutter early once we can guarantee that all subsequently found cuts have a larger expansion than the best cut found so far. Suppose that c is the size up to which FlowCutter has computed the cuts, then $\frac{c+1}{|V|/2}$ is a lower bound on the expansion of all subsequent cuts. If the expansion of the best cut seen so far is below or equal to this lower bound, then we abort FlowCutter. As base case for the recursion we use trees and cliques. On cliques any order is optimal and on trees an optimal order can be derived from a linear-time-computable [22] optimal node ranking.

Road graphs have many nodes of degree 1 or 2. We exploit this in a fast preprocessing step to significantly reduce the graph size. Our algorithm determines the largest biconnected component B using [17] in linear time. It then removes all edges from G that leave B. It continues independently on every connected component of G. The resulting orders are concatenated. The order of B must be last. The other orders can be concatenated in an arbitrary way. For each connected component our algorithm identifies the degree-2-chains. For a chain $x, y_1 \dots y_k, z$ it removes all y_i and adds an edge from x to z unless x or z have degree 1. The y_i nodes and x or z if they have degree 1 are positioned at the front of the order. Their relative order is determined using the optimal tree ordering algorithm. All remaining nodes are ordered behind them. After eliminating degree-2-chains our algorithm uses the nested dissection algorithm described above.

6 Experiments

We compare Flowcutter to the state-of-the-art partitioners KaHip, Metis, and InertialFlow. We present three experiments: (1) we compare the produced contraction orders in terms of CCH performance, (2) compare the Pareto-cut-sets, and (3) evaluate FlowCutter on non-road graphs using the Walshaw benchmark set. The last experiment is in the long version of this paper [15] and can be summarized as follows: For $\epsilon = 5\%$ there are only 6 out of 24 graphs where FlowCutter does not match the best known solutions. For 3 of them FlowCutter is off by at most 5 edges. All experiments were run on a Xeon E5-1630 v3 @ 3.70GHz with 128GB DDR4-2133 RAM.

6.1 Order Experiments We compute contraction orders for 4 DIMACS roads graphs [9]. The smallest is Colorado with $n = 436$K and $m = 1$M. Next is California and Nevada with $n = 1.9$M and $m = 4.6$M, followed by (Western) Europe with $n = 18$M and $m = 44$M and finally a graph encompassing the whole USA with $n = 24$M and $m = 57$M.

We use FlowCutter with all extensions in two variants denoted by F20 and F3, with 20 respectively 3 random source-target-pairs. We use the `ndmetis` tool of Metis 5.1.0 with the default parameters and refer to it as M. Unfortunately KaHip[1] and InertialFlow do not provide order computation tools. We therefore implemented basic nested dissection ordering algorithms on top of them. The KaHip implementation was already used in [10] and is the current state of the art in terms of order quality. We refer to it as K. The tool iteratively computes cuts using KaHip-strong 0.61 using different random seeds until the cut size does not decrease for 10 rounds. We set $\epsilon = 20\%$ for KaHip. This value is comparatively small, but KaHip has problems with large ϵ as demonstrated in the Pareto-cut experiments in Section 6.2. Note that this setup solely optimizes order quality disregarding order computation times, which therefore can certainly be improved. We report the corresponding running times therefore as upper bounds. Note that we argue that FlowCutter is superior mostly because of the achieved order quality, not because it is particularly fast. Not having well-tuned KaHip running times is therefore not problematic for our comparison. We reimplemented InertialFlow and were able to reproduce the cuts and running times of the original publication with our implementation. It is not randomized and therefore computing several cuts with different random seeds per graph as for KaHip is not useful. As consequence the reported running times adequately represent the performance of a basic nested dissection algorithm combined with Inertial Flow. InertialFlow is denoted by I and we set $\epsilon = 60\%$. Both KaHip and InertialFlow compute edge cuts. We turn them into node separators by choosing the endpoints of the cut edges on the larger side.

Results. Our results are summarized in Table 1. We observe that, modulo small cache effects, the customization time is correlated with the number of triangles and the average query running time is correlated with the number of arcs in the CCH. The memory needed per weight are correlated with the number of arcs in the CCH. The CCH-structure memory consumption is dominated by the list of precomputed triangles and thus the amount of necessary memory is correlated with the number of triangles. All these correlation are non-surprising and were predicted by CCH theory. Denote by n_s and m_s the number of nodes and arcs in the search space. For the average numbers we observe that $1.7 \leq \frac{n_s(n_s-1)}{2}/m_s \leq 2.6$ and for the maximum numbers we observe that $2.1 \leq \frac{n_s(n_s-1)}{2}/m_s \leq 3.9$, which indicates that the search spaces are nearly complete graphs. The number of nodes and the number of arcs are thus related. We can thus say that search space is small or large without indicating whether we refer to nodes or arcs.

Search Space. FlowCutter produces the smallest search spaces. Using more source-target pairs results in better orders, but already 3 give a decent order. Inertial Flow is dominated by KaHip with the exception of the USA graph. Metis is last by a significant margin on all but the smallest graph. The ratio between the average and the maximum size is very interesting. A high ratio indicates that a partitioner often finds good cuts, but at least one cut is comparatively bad. This ratio is never close to 1, indicating that road graphs are not perfectly homogeneous. In some regions, probably cities, the cuts are worse than in some other regions, probably the country-side. Compared to the competitors, the ratio is however higher for InertialFlow. This illustrates that its geography-based heuristic is effective most of the time but not always.

CCH Size. A small search size is not equivalent with the CCH containing only few arcs. It is possible that vertices are shared between many search spaces and thus the CCH can be significantly smaller than the sum of the search space sizes. This effect occurs and explains why the number of arcs in CCH is orders of magnitude smaller than the sum over the arcs in all search spaces. Further, minimizing the number of arcs in the CCH is not necessarily the same as minimizing the search space sizes. This explains why Metis beats KaHip in terms of CCH size but not in terms of search space

[1] Some preliminary work was done in [26].

		Search Space				#Arcs		Up.	Running times				
		Nodes		Arcs [$\cdot 10^3$]		in CCH	#Tri.	Tw.	Order	Cust.	Query	Mem. [MiB]	
		Avg.	Max.	Avg.	Max.	[$\cdot 10^6$]	[$\cdot 10^6$]	Bd.	[s]	[ms]	[μs]	per w.	indep.
Col	M	155.6	354	6.1	22	1.4	6.4	102	2.0	18	26	10	61
	K	135.1	357	4.6	22	1.7	7.2	103	$\leq$3 837.1	21	20	13	70
	I	151.2	542	6.2	38	1.5	7.4	119	7.4	21	25	11	69
	F3	126.3	280	4.1	15	1.3	4.8	91	10.3	15	18	10	48
	F20	122.4	262	3.8	14	1.3	4.4	87	61.0	14	18	10	44
Cal	M	275.5	543	17.3	53	6.5	36.4	180	9.9	88	60	50	335
	K	187.7	483	7.0	37	7.5	34.2	160	$\leq$18 659.3	90	30	57	326
	I	191.4	605	7.1	53	6.9	34.1	161	42.6	84	31	52	320
	F3	177.5	356	6.2	24	5.9	23.4	127	64.1	69	27	45	231
	F20	170.0	380	5.6	26	5.8	21.8	132	386.8	66	26	44	218
Eur	M	1 223.4	1 983	441.4	933	69.9	1 390.4	926	125.9	2 242	1 162	533	11 210
	K	638.6	1 224	114.3	284	73.9	578.2	482	$\leq$213 091.1	975	304	564	5 044
	I	732.9	1 569	149.7	414	67.4	589.7	516	1 017.2	932	385	514	5 082
	F3	734.1	1 159	140.2	312	60.3	519.4	531	2 532.7	853	366	460	4 491
	F20	616.0	1 102	102.8	268	58.8	459.6	455	16 841.5	780	271	449	4 024
USA	M	990.9	1 685	249.1	633	86.0	1 241.1	676	170.8	2 084	651	656	10 217
	K	575.5	1 041	71.3	185	97.9	737.1	366	$\leq$265 567.3	1 250	202	747	6 462
	I	533.6	1 371	62.0	291	88.8	682.0	384	1 076.8	1 122	177	677	5 972
	F3	562.7	906	66.4	159	75.9	478.4	321	2 117.7	856	190	579	4 320
	F20	490.6	868	52.7	154	74.3	440.5	312	12 379.2	811	156	567	4 019

Table 1: Contraction Order Experiments. We report the average and maximum over all nodes v of the number of nodes and arcs in the CCH-search space of v, the number of arcs and triangles in the CCH, and the induced upper treewidth bound. We additionally report the order computation times, the customization times, and the average shortest path distance query times. Only the customization times are parallelized using 4 cores. The customization times are the median over 9 runs to eliminate running variance. The query running times are averaged over 10^6 *st*-queries with s and t picked uniformly at random. Finally, we report the memory needed per directed 32bit weight, including the input graph weights, and for the weight-independent CCH structure. Note that several CCH customization variants exist. The one we report is non-amortized, non-perfect, with SSE and uses precomputed triangles. The CCH structure space consumption includes the precomputed triangles.

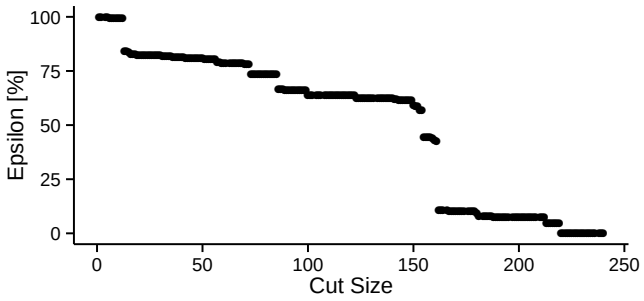


Figure 6: F20 Pareto cuts, Central Europe

size. InertialFlow seems to be comparable to Metis in terms of CCH size, as the CCH arc count is sometimes slightly below and sometimes slightly larger. However, FlowCutter beats all competitors and clearly achieves the smallest CCH sizes.

Triangles. A third important order quality metric is the number of triangles in the CCH. Metis is competitive on the two smaller graphs, but is clearly dominated on the continental sized graphs. InertialFlow and KaHip seem to be very similar, with the exception of the USA graph where InertialFlow comes out slightly ahead. FlowCutter also wins with respect to this quality metric producing between 20% and 30% less triangles compared to the closest competitor.

Treewidth. As the CCH is essentially a chordal graph which are closely tied to tree decomposition, we obtain upper bounds on the treewidth of the input graphs as a side product. This quality metric is not directly related to CCH performance, but is of course indirectly related as most of the other criteria can be bounded in terms of it. As such it reflects the same trend: Metis is worst, followed by InertialFlow, followed by KaHip, and FlowCutter with the best bounds.

Order Computation Times. Quality comes at a price and thus the computation times of the orders follow the opposite trend: FlowCutter is the slowest, followed by InertialFlow, while Metis is astonishingly fast. Where KaHip fits into the picture is unclear, as the nested dissection implementation employed is not tuned for computation speed and only for order quality. However, the times in the next experiment suggested that a well-tuned implementation is between FlowCutter and InertialFlow.

6.2 Pareto Cut Set Experiments. In the previous experiment we have demonstrated that FlowCutter produces the best contraction orders. In this section we look at the Parteo-cut sets of two graph in more detail. Selecting meaningful and representative testing instances is difficult. The cuts of the USA graph are dominated by the cut induced by the Mississippi, as demonstrated in the long version of this paper [15]. The Europe graph is problematic as the top level cuts behave differently from nearly all lower level cuts. On the top level there are many comparatively weakly connected peninsulas. This structure is very rare on the lower levels. This leads to a special behavior that we discuss in detail in the long version of this paper [15] which can be summarized as follows: Cutting the peninsulas leads to a smaller cut but only delays the inevitable cut through central Europe in a recursive setup. Cutting the peninsulas thus looks clearly better, even though it is not clearly superior when considering a recursive partitioning. We therefore run experiments on a subgraph of the Europe with a latitude $\in [45, 52]$ and longitude $\in [-2, 11]$ that encompasses most of Central Europe, i.e., with all the peninsulas cut off. We additionally pick the DIMACS California&Nevada graph because [6] determined an optimal cut of this graph for $\epsilon = 0$. The long version of this paper [15] additionally contains numbers for the Colorado graph. We compare KaHip 0.71, Metis 5.1.0, InertialFlow and FlowCutter-20 in terms of edge cut sizes. The first three compute a single cut, whereas FlowCutter computes a Pareto-set, such as the one illustrated in Figure 6. We therefore run the first three for various choices of ϵ. We use KaHip-strong with `--enforce_balance` for $\epsilon = 0$. All other parameters have default values.

Results. Table 2 summarizes our results. Metis produces extremely bad cuts for imbalances above 70%. Strangely KaHip has problems with perfect balance. This is unexpected as KaHip was optimized for perfect balance [21]. This is most likely the result of the default parameters not being optimized for road graphs. KaHip and Metis mostly ignore the allowed imbalance. The maximum achieved imbalance of KaHip is 3.2% even though 90% is allowed. Metis is nearly always well below 1%. Interestingly increasing the allowed imbalance can increase the achieved cut sizes. We conclude that computing a full Pareto-cut-set for a road graph is not possible in the straight-forward way with KaHip or Metis.

InertialFlow is bad at finding highly balanced cuts. Fortunately, for higher values of ϵ competitive cuts are found. This explains why the computed contraction orders are competitive. A significant advantage of InertialFlow compared to Metis and KaHip is that a higher maximum imbalance cannot increase the cut size. Unfortunately, InertialFlow has its own set of problems. It does not find the best cut just below the allowed maximum imbalance. For example the good cut through Europe with $\epsilon = 10.542\%$ is not found when allowing a maximum imbalance of 30%. A maximum

max ϵ	Achieved ϵ [%]				Cut Size				Running Time [s]			
[%]	F20	K	M	I	F20	K	M	I	F20	K	M	I
0	0.000	0.000	0.000	0.000	39	157	51	306	59.8	30.8	0.8	1.1
1	0.169	0.184	0.000	0.566	31	31	52	93	53.2	14.6	0.8	1.4
3	2.293	2.300	0.001	1.112	29	29	61	64	51.0	24.0	0.8	1.7
5	2.293	2.293	0.005	1.571	29	29	42	62	51.0	36.4	0.8	2.3
10	2.293	2.304	0.001	0.642	29	29	43	37	51.0	76.2	0.8	2.2
20	16.706	2.756	0.000	2.656	28	30	41	29	49.6	15.0	0.9	2.4
30	16.706	2.768	13.936	5.484	28	29	51	29	49.6	15.5	0.8	2.9
50	49.058	2.768	0.000	40.833	24	29	39	27	43.2	15.5	0.8	3.7
70	49.058	2.768	41.178	42.591	24	29	4 310	26	43.2	15.4	0.8	4.9
90	89.838	2.768	47.370	85.555	14	29	3 711	18	25.4	15.6	0.9	5.2

(a) California and Nevada

max ϵ	Achieved ϵ [%]				Cut Size				Running Time [s]			
[%]	F20	K	M	I	F20	K	M	I	F20	K	M	I
0	0.000	0.000	0.000	0.000	240	716	369	1 180	1 390.3	369.1	3.3	4.3
1	0.132	0.998	0.000	0.089	220	245	360	391	1 342.9	80.2	3.3	7.9
3	0.132	0.457	0.000	0.008	220	227	372	319	1 342.9	112.5	3.1	10.2
5	4.894	0.464	0.000	0.857	213	227	369	276	1 319.0	158.3	3.3	12.3
10	9.330	0.043	0.000	0.375	180	228	375	241	1 181.5	338.1	3.1	16.8
20	10.542	3.139	0.000	0.132	162	250	375	220	1 089.5	75.5	3.1	25.6
30	10.542	3.139	0.017	7.384	162	250	369	203	1 089.5	75.4	3.1	34.9
50	44.386	3.139	33.336	10.542	155	250	9 881	162	1 047.8	75.3	3.2	47.5
70	66.655	3.139	41.178	44.386	86	250	14 375	155	591.6	75.5	3.2	82.8
90	84.199	3.139	83.087	84.257	13	250	28	17	92.8	75.4	3.3	17.1

(b) Central Europe

Table 2: Pareto-Set Experiments. We report the balance, the cut size and the computation time for various partitioners and allowed maximum imbalance. For FlowCutter the computation time includes the time needed to compute all less balanced cuts in the Pareto cut set.

imbalance of 50% is necessary, i.e., the choice of 30% vs 50% determines whether a 10.5% cut is found or not. Unfortunately, a higher maximum imbalance is not always better. Consider the two cuts with 29 edges on California. They differ in the achieved balance, i.e., two cuts with the same size but a different balance exist. InertialFlow does not find the variant with the better balance, if the maximum allowed imbalance is too high. Further, it fails to find the 29 edge cut with the best balance which is only found by KaHip and FlowCutter. Unfortunately, also KaHip can not find it reliably, as it finds 4 different cuts with 29 edges and varying balances.

Only FlowCutter reliably finds the variant with the best balance. On the two tested graphs in Table 2 FlowCutter even finds for every evaluated ϵ a smaller or equally sized size than the best competitor[2].

In [6] an optimal California cut for $\epsilon = 0\%$ with 32 edges was computed. All tested algorithms are therefore suboptimal as the best one finds a cut with 39 edges. However, even a slight imbalance of 1% is enough for FlowCutter and KaHip to find cuts with 31 edges. The achieved 1% cuts can therefore be optimal.

Metis is the fastest, followed by InertialFlow, followed by KaHip. Positioning FlowCutter in this list is difficult, as it (a) is the only one to compute Pareto cut set, enabling plots such as those in Figure 6, and (b) even if one is only interested in a single cut, it honors the maximum imbalance parameter much better.

7 Conclusion and Future Research

We introduced FlowCutter, a bisection algorithm that optimizes balance and cut size in the Pareto sense. We used it to compute contraction orders (also called elimination or minimum fill-in orders) and have shown that it beats the state of the art in terms of quality on road graphs.

Future Research. FlowCutter needs two initial nodes on separate sides of the cut. Currently these are determined by random sampling. A better selection strategy could decrease the number of samples needed. Further investigating other piercing heuristics could also be beneficial.

Acknowledgment. We thank Roland Glantz for helpful discussions.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge*, volume 588. American Mathematical Society, 2013.

[3] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. Technical Report abs/1504.05140, ArXiv e-prints, 2015.

[4] Hans L. Bodlaender. Treewidth: Structure and algorithms. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity*, volume 4474 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 2007.

[5] Paul Bonsma. Most balanced minimum cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010.

[6] Daniel Delling, Daniel Fleischer, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. An exact combinatorial algorithm for minimum graph bisection. *Mathematical Programming*, pages 1–24, 2014.

[7] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 2015.

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

[9] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[10] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*, volume 8504 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2014.

[11] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. Technical Report abs/1402.0402, ArXiv e-prints, 2014.

[12] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] Lester R. Ford, Jr. and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[14] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified $\mathcal{NP}$-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[15] Michael Hamann and Ben Strasser. Graph bisection with pareto-optimization. Technical report, arXiv, 2015.

[16] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[17] John E. Hopcroft and Robert E. Tarjan. Efficient

[2] There are graphs where this is not the case, see extended version [15].

algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[18] 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, 1999.

[19] Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[20] Richard J. Lipton, Donald J. Rose, and Robert Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

[21] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[22] Alejandro A. Schæffer. Optimal node ranking of trees in linear time. *Information Processing Letters*, 33:91–96, November 1989.

[23] Aaron Schild and Christian Sommer. On balanced separators in road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, Lecture Notes in Computer Science. Springer, 2015.

[24] 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(12):1–23, 2000.

[25] A. J. Soper, Chris Walshaw, and Mark Cross. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *Journal of Global Optimization*, 29(2):225–241, 2004.

[26] Michael Wegner. Finding small node separators. Bachelor thesis, Karlsruhe Institute of Technology, October 2014.