Graph Bisection with Pareto Optimization

MICHAEL HAMANN and BEN STRASSER, Institute of Theoretical Informatics, Karlsruhe Institute of Technology

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 as a subroutine, we build variants that compute node separators that are independent of s and t. From the computed Pareto set, we can identify cuts with a particularly good tradeoff 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 (CCHs), a speed-up technique for shortest-path computations. Our core algorithm runs in $O(c|E|)$ time, where E is the set of edges and c is the size of the largest outputted cut. This makes it well suited for separating large graphs with small cuts, such as road graphs, which is the primary application motivating our research. For road graphs, we present an extensive experimental study demonstrating that FlowCutter outperforms the current state of the art in terms of both cut sizes and CCH performance. By evaluating FlowCutter on a standard graph partitioning benchmark, we further show that FlowCutter also finds small, balanced cuts on nonroad graphs. Another application is the computation of small tree decompositions. To evaluate the quality of our algorithm in this context, we entered the PACE 2016 challenge [13] and won first place in the corresponding sequential competition track. We can therefore conclude that our FlowCutter algorithm finds small, balanced cuts on a wide variety of graphs.

CCS Concepts: • **Mathematics of computing** → *Network flows*; *Graph algorithms*;

Additional Key Words and Phrases: Graph bisection, graph partitioning, graph clustering, shortest path, flow

ACM Reference format:
Michael Hamann and Ben Strasser. 2018. Graph Bisection with Pareto Optimization. *J. Exp. Algorithmics* 23, 1, Article 1.2 (February 2018), 34 pages.
https://doi.org/10.1145/3173045

1 INTRODUCTION

A graph cut is a set of edges whose removal separates a graph into two sides. Similarly, a node separator is a set of nodes whose removal separates a graph into two sides. A cut or separator is balanced if the number of nodes in both sides is roughly the same. Balanced graph bisection is the problem of finding a balanced cut or separator. This is a fundamental and NP-hard [24] graph problem that has received a lot of attention [2, 16, 34, 39, 42] and has many applications. We present FlowCutter, a novel algorithm to compute edge cuts and node separators. It computes a set of cuts or separators that trade off cut and separator size, respectively, for imbalance in the Pareto sense. For edge cuts, FlowCutter guarantees that the two sides of the cut are connected subgraphs.

Authors' addresses: M. Hamann and B. Strasser, Karlsruher Institut für Technologie (KIT), Institut für Theoretische Informatik, Lehrstuhl für Algorithmik I, Postfach 6980, 76128 Karlsruhe; emails: michael.hamann@kit.edu, academia@ben-strasser.net.

https://doi.org/10.1145/3173045

Experimentally, we show that the computed cuts and separators have good quality. However, as the problem is NP-hard, we cannot prove that the cuts and separators found are always small. The application motivating our research is accelerating shortest-path computations in road graphs [3, 15, 19, 30, 43]. The main part of our article sticks with this application, and we demonstrate the usefulness of our proposed graph bisection algorithm directly by applying it in the context of shortest-path computations. However, our proposed algorithm is not specific to road graphs. We present experiments on graphs from other applications to demonstrate this. Earlier versions of our work have been uploaded to ArXiv [28] and presented at the ALENEX 2016 conference [29].

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 a 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 (CCHs) [19]. We prove that the core algorithm's running time is in $O(c|E|)$, where E is the set of edges and c the size of the largest cut found. We show in an extensive experimental evaluation that FlowCutter is a good fit for road graphs, as road graphs tend to be large in terms of edge count but small in terms of cut size. We further show that FlowCutter also finds small, balanced cuts on nonroad graphs by comparing the cuts found by FlowCutter to the state-of-the-art cuts on the benchmark set maintained by Chris Walshaw [44]. Finally, we demonstrate the performance of our algorithm on a broad class of graphs, by entering and winning the corresponding sequential competition track of the PACE 2016 tree decomposition challenge.

Outline. Section 2 presents an overview of related work and the core ideas of the shortest-path application driving our research as well as some other applications including tree decompositions. Section 3 presents our notation and an overview of flows, CCH, tree decompositions, separators, and cuts and how some of these concepts relate to one another. Section 4 introduces the core FlowCutter *st*-bisection algorithm. Section 5 extends the core algorithm to general bisection and node bisection and describes how to compute CCH contraction orders. Section 6 presents an extensive experimental evaluation on road graphs against the current state of the art and on the well-known Walshaw benchmark for graph partitioning.

2 APPLICATIONS AND RELATED WORK

We start by giving an overview of the core ideas employed to accelerate shortest-path computations, illustrating how to apply the main topic of this work, which is graph bisection, to this application. We limit this overview to the works immediately relevant to graph bisection and refer readers interested in further details to a recent survey article [3].

Dijkstra's algorithm [20] solves the shortest-path problem in near-linear running time. However, this is not fast enough if the graph consists of a whole continent's road network. Acceleration techniques usually compute auxiliary data in a *preprocessing phase* and compute the shortest paths in a *query phase*. This auxiliary data is independent of the path's endpoints and can therefore be reused for many shortest-path computations. As roads only change slowly over time, the preprocessing phase can be slow as it does not have to be rerun very frequently. Computing the auxiliary data usually involves optimizing some criteria, which most of the time is NP-hard [4]. In this application, it is therefore important to produce solutions of good quality, but it is not as important to do this quickly. Trading running time in the preprocessing phase for an improvement of the auxiliary data is therefore most of the time worthwhile. Some techniques such as [15, 19] split the preprocessing phase and introduce a *customization phase*. In the preprocessing phase, only the graph, not its weights, is known. The weights are introduced in the customization phase. The idea

behind this setup is to be able to adapt more quickly to changes in the weights, which could, for example, be caused by traffic congestion.

In many shortest-path acceleration techniques, the preprocessing phase involves computing balanced graph edge cuts or node separators. The central idea can be formulated in terms of edge cuts as well as node separators. We present the node separators variant as we will evaluate our bisection algorithms using *Customizable Contraction Hierarchies* [19], an accelerating technique based on the separator variant. However, many acceleration techniques, such as [15], are based on the edge cut variant. The idea can be described as follows: given a graph G and a node separator S, the algorithms precompute for every node in the graph how to get to every node in S. Further, they precompute the shortest paths among all nodes of S. Consider a query that asks to compute a shortest path from a node s to a node t. Either s and t is on the same or on opposite sides of S. If they are on opposite sides, a shortest path can be assembled by iterating over all nodes v in S and combining the precomputed paths from s to v and from v to t and picking the shortest path. The running time of a distance query in this case is thus in $O(|S|)$, which is assumed to be small for road graphs. However, if s and t are on the same side, then a graph search is necessary using, for example, Dijkstra's algorithm. On the side of s and t, the search is unrestricted. However, it does not cross S and instead makes use of the shortest paths precomputed between the nodes of S. If the sides are of the same size and s and t are chosen uniformly at random, then there is a 50% probability that they are on opposite sides. Half of the queries can therefore be answered quickly. For the other half of queries, this approach restricts the search to half of the graph. However, as half of a continent is still large, one usually applies this idea recursively.

The effectiveness of these techniques crucially depends on the size of the separators found. The balance is less important. Perfect balance is not necessary to ensure that the recursion has a logarithmic depth. This application does not, however, benefit from a perfect balance. In practice, the contrary is true: requiring perfect balance results in many small, slightly imbalanced separators not being found. Therefore, the perfectly balanced separators found can be larger. This larger size is detrimental to the running time of the query phase, compared to using the smaller, slightly imbalanced separators. Fortunately, road graphs have small separators and cuts because of geographical features such as rivers or mountains. Previous work has coined the term *natural cuts* for this phenomenon [16]. However, identifying these natural cuts is a difficult problem.

Graph partitioning software used for road graphs includes *KaHip* [39], *Metis* [34], *InertialFlow* [42], and *PUNCH* [16]. We experimentally compare FlowCutter with the first three. As we unfortunately have no implementation of PUNCH, we omitted an experimental comparison with it. All of these works formalize the graph bisection problem 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 several shortcomings. Consider a graph with a million nodes and set the maximum imbalance to 1%. Suppose an algorithm finds a cut C_1 with 180 edges and 0.9% imbalance. This is all the information of the cut's quality that is provided. Can you decide solely based on this information whether this is 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 program 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; i.e., it tries to compute solutions that are Pareto optimal. As this problem is

NP-hard, one cannot expect to always find the exact Pareto curve. A further significant shortcoming of the state-of-the-art partitioners, with the exception of InertialFlow, is that they were designed for small imbalances. Common benchmarks, such as the one maintained by Chris Walshaw [44], only include test cases with imbalances up to 5%. However, for our application, imbalances of 50% can be fine. For such high imbalances, unexpected things happen with the standard software, such as that increasing the allowed imbalance can increase the achieved cut sizes. Indeed, KaHip, one of the competitors, has been updated, as a reaction to the conference version of our work [29], to overcome some of these shortcomings. The newer version produces better results for higher imbalances than the old version.

We use CCHs [19] to evaluate the quality of the separators found by FlowCutter. We present a more detailed CCH overview in Section 3.3. The CCH auxiliary data at its core is a chordal supergraph of the road graph. The maximum cliques of G', which can be efficiently identified in chordal graphs, correspond to the bags of a tree decomposition. This connection gives us a bridge to the vast field of tree decomposition theory. In Section 3.4, we give a high-level overview of the concepts related to tree decompositions and CCH. For a more in-depth survey, we refer to [7], [8], and [9]. In some works, tree decompositions are also called clique trees.

Other Applications. A vast amount of algorithms for NP-hard graph problems exist that are fixed-parameter tractable in the tree width of a graph G [9, 12]. It is therefore an interesting question whether algorithms being able to compute good tree decompositions in practice leads to practicable variants of these algorithms. To investigate this question, the PACE competition [13] was held in 2016 at IPEC, a conference with a focus on fixed-parameter tractable algorithms. The objective of two competition tracks was to compute a small tree decomposition within a limited timeframe. The tracks differed in whether parallelism was allowed or not. To evaluate the performance of FlowCutter in this context, we submitted our algorithm. Our implementation runs FlowCutter iteratively with varying parameters until the time limit is reached. The parallel version runs several FlowCutter instances in parallel. The code submitted to the PACE challenge is open source and available at [45]. In the sequential track, our implementation won first place out of six submissions, and in the parallel track, it won second place out of three submissions. Given these results, it is safe to say that our algorithm is at least highly competitive, if not the state of the art, in terms of computing tree decompositions in practice.

The contraction orders used by CCH, which are based on nested dissection [25, 36], are also called minimum fill-in orders in the context of sparse matrices. This establishes a connection to the theory of quickly solving sparse systems of linear equations. Indeed, METIS was developed with this application in mind [34]. METIS was not developed to bisect road graphs. The fact that we use METIS in the context of road graphs is therefore an example of this theoretical connection being exploited in practice. Using the same connection, it is also possible to use FlowCutter to solve a sparse system of linear equations. However, even though these two applications are so closely related, the precise tradeoff between the various optimization criteria differs. For example, in the context of sparse equation systems, cut size is less important than in the road setting, whereas having a small bisection algorithm running time is more important.

Another application is information propagation in belief networks [32]. In this setting, a set of random variables is given. It is known how these random variables depend on each other and their interactions are modeled as a graph whose nodes are the random variables. The question is how the distributions change throughout the graph if the distribution of a subset of the variables changes—i.e., some but not all variables are measured. To solve this problem, so-called junction trees are employed. Junction trees are essentially another name for tree decompositions. As we can use FlowCutter to compute tree decompositions, we can also use it to compute junction trees.

3 PRELIMINARIES

A *directed graph* is denoted by $G = (V, A)$ with *node set* V and *arc set* $A \subseteq V \times V$. Similarly, an *undirected* graph is denoted by $G = (V, E)$ with node set V and *edge set* $E \subseteq \{e \in 2^V : |e| = 2\}$. Arcs have an implicit direction, whereas edges are undirected. A directed graph is *symmetric* if for every arc (y, x) there exists an arc (x, y). In a slight abuse of notation, we do not discern between undirected and directed, symmetric graphs. We identify an edge $\{x, y\}$ with the corresponding pair of arcs (x, y) and (y, x). We set $n := |V|$ and $m := |A|$. As input, we only consider undirected graphs without multiedges and without reflexive loops, i.e., without arcs of the form (x, x). Road graphs that do not fit this description are modified[1] by removing multiedges, removing reflexive loops, and adding backarcs in the case of one-way streets. In intermediate steps of our algorithm, we also consider nonsymmetric directed graphs. The *out-degree* $d_o(x)$ of a node x 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 $k \geq 1$, $d(x) \neq 2$, $d(z) \neq 2$, 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 $i + 1$ is P's length. The *distance* $\text{dist}(x, y)$ is defined as the minimum length over all xy-paths.

3.1 Cuts and 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*. In another slight abuse of notation, we do not discern between the node partition and the set of cut-arcs. 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 $V = V_1 \cup V_2 \cup Q$. There must be no arc between V_1 and V_2. The cardinality of Q is the *separator's size*. The *imbalance* ϵ of a cut or separator 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. For edge cuts, $0 \leq \epsilon \leq 1$ holds. This is not necessarily the case for node separators. The separator itself may contain nodes, making it possible that the minimum ϵ is smaller than 0, as both sides can have fewer than $n/2$ nodes. An *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 and 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 tradeoff*. It is possible that several Pareto-optimal cuts exist with the same tradeoff. The problem we consider asks one to compute one cut for every Pareto tradeoff. If there are several, then the algorithm is free to pick any one of them.

This is a departure from existing experimental papers [2, 14, 34, 39, 42, 44, 47] that consider the problem of finding a smallest cut subject to an imbalance bounded by an input parameter. Given a cut for every Pareto tradeoff, 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 solvable in polynomial time [21]. However, nearly all cut problems that combine optimizing imbalance and cut size are NP-hard. Examples include:

[1] As shown in the CCH paper [19], this simplification does not hinder the applicability of CCH to directed graphs.

- Finding a perfectly balanced minimum cut, i.e., one with $\epsilon = 0$, is NP-hard [24].
- A *sparsest cut* C is a cut that minimizes $\frac{|C|}{|V_1| \cdot |V_2|}$. A sparsest cut is Pareto optimal. Finding a sparsest cut is NP-hard [35].
- Even computing, for a fixed st-pair, a most balanced cut among all st-cuts of minimum size is already NP-hard [11].
- In [46], it was shown that computing a minimum cut that respects a given imbalance is NP-hard.

Being able to compute a cut for every Pareto tradeoff 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 tradeoff. Our algorithm tries to heuristically compute in a single run a cut for every Pareto tradeoff.

3.2 Flows

In this article, we only consider *unit flows*. These are a restricted variant of the flow problem: every arc has capacity 1 and an integral flow intensity of either 0 or 1. Formally, a flow is a function $f : A \to \{0, 1\}$. An arc a with $f(a) = 1$ is *saturated*. Denote by $p(x) = \sum_{(x,y)\in A} f(x, y) - \sum_{(y,x)\in A} f(y, x)$ 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

- flow may be created at sources—i.e., $\forall s \in S : p(s) \geq 0$;
- flow may be removed at targets—i.e., $\forall t \in T : p(t) \leq 0$;
- flow is conserved at all other nodes—i.e., $\forall x \in V \backslash (S \cup T) : p(x) = 0$; and
- 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 $y \notin S$. In other works, the flow intensity is sometimes also called the flow value. A path $a_1, a_2 \ldots, a_i$ is *saturated* if there exists an i with $f(a_i) = 1$. A node x is *source reachable* if a nonsaturated sx-path exists with $s \in S$. Similarly, a node x is called *target reachable* if a nonsaturated 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 [21], it was shown that a flow is maximum if and only if no nonsaturated st-path with $s \in S$ and $t \in T$ exists. If such a path exists, then it is called an *augmenting path*. The classic approach to compute max-flows consists of iteratively searching for augmenting paths. Our algorithm builds upon this classic 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. The source-side and target-side cuts are the same for every max-flow. The latter is an implication of Corollary 5.3 of [22], which states that every min-cut is saturated with respect to every max-flow.

3.3 Customizable Contraction Hierarchy

A 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 separators. No part of FlowCutter builds upon CCH. The CCH details are explained in [19]. The technique uses three phases, i.e., a preprocessing phase, a customization phase, and a query phase. In the preprocessing phase, the arc weights are unknown. These are integrated into the auxiliary data in the customization phase. Shortest paths are computed in the query phase.

Preprocessing. The name-giving operation is the node contraction: contracting a node v consists of removing v and adding edges between all of v's neighbors, if they did not already exist. 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. By convention, we say for every edge $\{x, y\}$ where x is contracted before y that x is lower than y. The position of a node in the order is called *rank*.

Customization. The weights of G' are computed using an algorithm that essentially enumerates all triangles in G'. Initially, all edges of G' already present in G are assigned their input weights, whereas all edges introduced during the contraction are giving the weight ∞. The algorithm then enumerates all triangles $\{x, y, z\}$ in G' ordered increasingly by the position of the lowest node in the triangle. Suppose that z is this lowest node. For each triangle, the algorithm executes $w(x, y) \leftarrow \min\{w(x, y), w(x, z) + w(z, y)\}$. As shown in [19], for every pair of nodes s and t, there is a shortest *st-up-down-path* after the customization has finished. This is a path with nodes $v_1 \dots v_m \dots v_\ell$ such that the nodes $v_1 \dots v_m$ appear ordered increasingly by rank and the nodes $v_m \dots v_\ell$ appear ordered decreasingly by rank. The nodes $v_1 \dots v_m$ form the *upward part* of the path and the nodes $v_m \dots v_\ell$ form the *downward part*.

Query. Given the weights of G', the *shortest-path query* consists of a bidirectional search in G'. Both the forward and the backward searches follow only upward arcs (x, y) such that x has a lower rank than y. The forward search finds the upward part and the backward search the downward part of a shortest st-up-down-path. The *search space* of a node z is the subgraph of G' that is reachable from z while only following upward arcs. The query therefore only explores at most the whole search spaces of s and t.

Performance. Smaller search spaces yield faster queries. Fewer triangles in G' yield a faster customization. Fewer edges in G' result in less memory consumption. All these quality metrics depend on the contraction order.

We compute contraction orders using *nested dissection* [25, 36], which works as follows: (1) determine a small balanced separator S, (2) recursively compute orders L and R for both sides, and (3) the contraction order of G consists of first contracting the nodes along L, then R, and finally along an arbitrary order of the nodes in S. The quality of the so-obtained contraction order depends on the quality of the separators used in its construction. Finding these separators is where FlowCutter fits into the big picture.

3.4 Tree Decompositions

We present an overview of the theory of tree decompositions. As the sheer quantity of works written on this subject makes it impossible to introduce them all, we refer the interested reader to three survey articles [7–9].

Definition. A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair (B, T), where B is a set of subsets of V. The elements of B are called *bags*. Every bag is a set of nodes. The union of all bags must equal V. It is required that for every $\{x, y\} \in E$ there is a bag $b \in B$ such that $x \in b$ and $y \in b$; i.e., every edge is in one or more bags. T is a tree with the elements of B as nodes. T is called the *backbone* of the decomposition. Finally, it is required that for every pair of bags b_s and b_t whose intersection contains a vertex $v \in b_s \cap b_t$, all bags b_i on the unique path from b_s to b_t in the backbone T also contain v. The *width* of a decomposition is defined as the maximum bag size minus one. The *tree width* of a graph is the minimum width over all decompositions. For simplicity, we will further assume in our overview that all tree decompositions are well behaved; i.e., the subgraph of G induced by a bag is connected and no bag is a subset of another bag. Given

a tree decomposition that does not have these properties, it is possible to obtain one that does by removing superfluous bags and splitting disconnected bags.

Chordal Graphs and Perfect Elimination Orders. An undirected graph G is *chordal* if in every cycle C of at least four nodes there is a chord, i.e., a pair of nodes that are adjacent in the graph G but not adjacent in the cycle C. A node is call *simplicial* if its neighbors form a clique. Every chordal graph contains at least one simplicial node [23]. A perfect elimination order of an undirected graph G is a node order $v_1, v_2 \dots v_n$, such that v_i is simplicial in the subgraph of G induced by $v_i, v_{i+1} \dots v_n$. Not every graph possesses a perfect elimination order. Chordal graphs can be characterized as the graphs that possess a perfect elimination order [23]. Such an order can be constructed, given a chordal graph, by iteratively removing simplicial nodes from a chordal graph. As chordal graphs can have several simplicial nodes, chordal graphs can possess several perfect elimination orders.

The CCH contraction order is a perfect elimination order of the supergraph G' constructed in the CCH. G' is thus a chordal supergraph of the input graph G. In the context of chordal graphs, a contraction order is also called an *elimination order*.

Converting Structures. A tree decomposition can be constructed, given a chordal supergraph G' of a graph G, as follows: start by identifying the maximal cliques in G'. This can be done in polynomial running time using the perfect elimination ordering v_i. Every maximal clique of G' must appear as a union of a node v_i with its neighborhood in the subgraph of G' induced by $v_i, v_{i+1} \dots v_n$. Testing which of these linearly many neighborhoods are maximal cliques results in all maximal cliques being found. The maximal cliques of G' are the bags of the decomposition. We denote their set by B. These cliques are the nodes of the tree backbone T. It remains to construct the edges of the tree backbone T. We therefore consider the weighted undirected graph $T' = (B, E)$, where an edge $\{x, y\}$ exists if the intersection of bags x and y is nonempty. An edge $\{x, y\}$ is weighted by the number of nodes in the intersection of x and y. T' is not necessarily a tree and thus not a valid backbone. However, every maximum spanning tree of T' is a valid tree backbone [6]. Computing a maximum spanning tree thus completes the tree decomposition construction. Given a tree decomposition, we can easily get back to the corresponding chordal graph. The transformation consists of adding edges between all nodes in each bag.

This gives us three views that encode essentially the same information: chordal supergraphs, tree decompositions, and elimination orders. These are interconvertible as follows:

- Given an undirected graph G and an elimination order o, we can get to a chordal supergraph G' by iteratively contracting the nodes.
- By computing a perfect elimination order of G', we can obtain an elimination order o' of G. The orders o and o' are not necessarily the same, but they induce the same chordal supergraph.
- From the chordal supergraph G', we can construct a tree decomposition of G as described above.
- From the tree decomposition of G, we can get back to the chordal supergraph G' by adding edges in each bag.

It is guaranteed that none of these transformations increases the width of the corresponding tree decomposition.

Elimination Tree. With respect to an elimination order, we can define the *elimination tree*.[2] For every node v_i at position i in the order, we define its parent in the elimination tree as the first node

[2]The elimination tree is actually a forest if the input graph is disconnected.

v_j that appears in the order after v_i such that an edge $\{v_i, v_j\}$ exists. If no such v_j exists, the node is a root in the elimination tree.

It can be shown that the nodes in the search space of v are the set of ancestors of v in the elimination tree [5]. In a shortest-path distance query from s to t in a CCH, the ancestors of s and t in the elimination tree are therefore the set of nodes touched by the CCH query algorithm. It is therefore our goal to minimize the depth of the elimination tree.

Two elimination orders that yield the same chordal supergraph do not necessarily yield the same elimination tree. The minimum depth over the elimination trees over all elimination orders is called the *tree depth*.

Computing Elimination Orders. Commonly used algorithms to compute tree decompositions of large graphs in practice are heuristics that try to guess the elimination order. The simplest is the minimum degree heuristic [26, 37]. It consists of iteratively contracting a node of minimum degree. This simple algorithm is usually fast and works well enough on some graphs, but at least on road graphs there is a large gap to the optimal orders [19].

A more sophisticated approach is called *nested dissection* [25, 36]. The idea is precisely the same as the approach we used to compute the CCH contraction orders. It consists of finding a small balanced separator and placing these nodes at the end of the elimination order. The separator is then removed from the graph and the algorithm recursively continues on both sides.

The operation of contracting a separator last can be interpreted as trying to guess a central edge in the tree backbone. Denote by a and b two nonleaf bags and by $\{a, b\}$ an edge in the backbone. The nodes of $a \cap b$ form a separator of the input graph [7]. If the edge $\{a, b\}$ is positioned near the center of the tree backbone, then the sides of the separator $a \cap b$ are likely of roughly the same size. This motivates why nested dissection works in practice.

3.5 Bounding CCH Performance in Terms of Tree Width and Depth

We can express the CCH performance in terms of tree depth, which can be bounded using the tree width.

A tree decomposition of minimum width does not necessarily have an elimination order that results in a minimum elimination tree depth. For example, a path has a tree decomposition of width 1 as it is a tree. However, no elimination order exists that yields a depth smaller than $n/2$. Fortunately, a tree decomposition exists with a width in $O(\log n)$ and a depth in $O(\log n)$. It can be obtained using nested dissection with balanced separators. Besides small bags, a tree decomposition must also have a logarithmic diameter to allow for a low-elimination tree depth. This logarithmic diameter corresponds to the logarithmic recursion depth obtained by recursively bisecting a graph along a balanced separator.

Denote by tw the tree width of the input graph G and by td its tree depth. [10] have shown that there always exists an elimination order of G that yields an elimination tree depth of $O(\text{tw} \log n)$, but the corresponding tree decomposition does not necessarily have a minimum width. Fortunately, the depth of every elimination tree is an upper bound to the width of the corresponding tree decomposition. There is therefore always a tree decomposition with width $O(\text{tw} \log n)$ that admits an elimination tree of depth $O(\text{tw} \log n)$. Assuming that we could construct an elimination order Π of minimum elimination tree depth, which is NP-hard [38], we could bound the performance of the corresponding CCH in terms of td.

A CCH query from s to t explores the search spaces of s and t. The number of nodes in each of them is bounded by td and thus no more than 2td nodes are visited. The running time of this exploration is, however, only bounded by $O(\text{td}^2)$ as the subgraphs can be dense. In practice, the average depth and the average number of arcs over all nodes might be a better performance

indicator than the maximum numbers. Fortunately, using a simple algorithm that proceeds top-down along the elimination tree, we can compute the average and maximum number of nodes and arcs in the subgraph of each node efficiently.

We are further interested in the number of edges and triangles in G' as these correspond to the customization running times and the memory consumption, respectively. The number of edges is equal to $\sum_v d_o(v)$ when directing every edge of G' upward. The number of triangles in which a node v appears as the lowest node is $(d_o(v) \cdot (d_o(v) - 1))/2$ as the upper neighbors of v form a clique. The total number of triangles is therefore $\sum_v (d_o(v) \cdot (d_o(v) - 1))/2$.

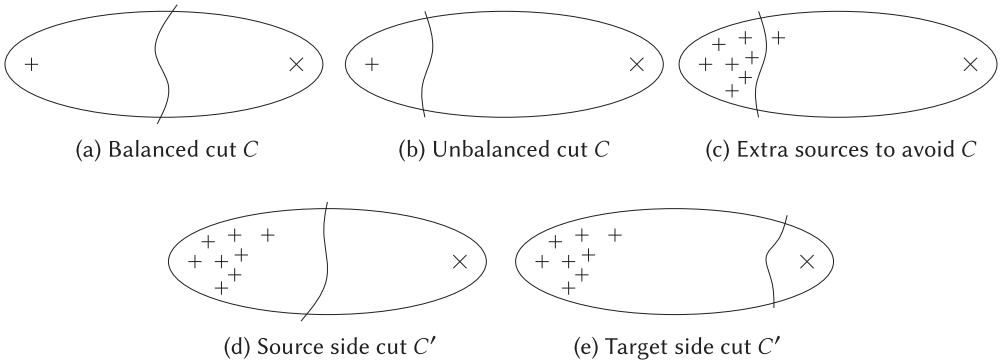
We can bound $d_o(v)$ using the width of the tree decomposition corresponding to Π. Recall that this decomposition does not necessarily have a minimum width, but, as shown by [10], its width can be bounded by td. We thus obtain the bounds of $O(n\text{td})$ for the number of edges and $O(n\text{td}^2)$ for the number of triangles. Fortunately, for planar graphs, [27, 36] have shown that there exist nested dissection orders such that the number of edges in G' is bounded by $O(n \log n)$. This is usually smaller than the $O(n\text{td})$ upper edge count bound. Further, this formula can also be used to bound the number of triangles by $O(n\text{td} \log n)$ as follows: we can bound $\sum_v d_o^2(v)$ by $(\max_v d_o(v)) \cdot \sum_v d_o(v)$. $\max_v d_o(v)$ is at most td and $\sum_v d_o(v)$ is the number of edges, i.e., at most $O(n \log n)$. Unfortunately, [27, 36] do not analyze the corresponding elimination tree heights. It is thus unknown whether they are in $O(\text{td})$. Road graphs are not strictly planar, but often planar enough to make this result relevant.

4 CORE FLOWCUTTER ALGORITHM

In the previous two sections, we described how finding good graph cuts and separators is beneficial to many applications. In this section, we propose our novel algorithm to compute graph cuts, named FlowCutter.

FlowCutter works by computing a sequence of st-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. FlowCutter picks C_S as C if there are fewer or equally as many nodes on the source side of C_S as 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 1(a), then our algorithm is finished. However, most of the time we are unlucky and we either have the situation depicted in Figure 1(b) 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 nonsource nodes into additional source nodes to invalidate C and computes a new, more balanced st-min-cut C', the second cut in the sequence. 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 a 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 the *piercing arc.* The situation is illustrated in Figure 1(c). All nodes on the source's side are marked as source node to ensure that C' does not cut through the source's side. The piercing node is necessary to ensure 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 4.2, we



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

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

Fig. 1. An ellipse represents a graph and the curved lines are cuts. The "+" signs represent source nodes and "×" signs represent target nodes.

describe heuristics to implement such a piercing oracle. For the algorithm to make progress, we need that C' is nondominated. As its size is at 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 1(d). If C' is a target-side cut as in Figure 1(e), 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 grows the sides around the source and target nodes. By doing so, it can guarantee that both sides are connected. In some applications, this is a desired property. In others, it might be an obstacle to finding the smallest possible cuts. Depending on the application, this property is therefore either a feature or a drawback of our algorithm.

Our algorithm computes the st-min-cuts by finding max-flows and using the max-flow-min-cut duality [21]. It assigns unit capacities to every edge and computes 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 nonsaturated xy-path exist. Analogously, *backward-growing* X consists of adding all nodes y for which a nonsaturated yx-path exists. The growing operations are implemented using a graph traversal algorithm (such as a DFS or BFS) that only follows nonsaturated arcs. The algorithm maintains, besides the flow values, four node sets: the set of sources S, the set of targets T, the set of source-reachable nodes S_R, and the set of target-reachable nodes T_R. 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 presented as Algorithm 1.

4.1 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 exact details are slightly more involved but, fortunately, the

ALGORITHM 1: Pseudo-Code Illustrating the Core *st*-Bisection Algorithm

```
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 edges;
13             x ← pierce node;
14             S ← S ∪ {x}; S_R ← S_R ∪ {x};
15             forward-grow S_R;
16         else
               // Analogous for target side
17         end
18     end
19 end
```

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 running time is thus in $O(cm)$. The remainder of this section contains the details necessary to formally show the $O(cm)$ worst-case running time.

Lines 1 to 3 of Algorithm 1, which initialize the data structures, have a running time in $O(m)$ and are therefore unproblematic. The condition in line 4 can be implemented in $O(1)$ as follows: S and T only grow. Using two bit-arrays with n elements, we can store which node is in S and which is in T. When adding a node, we raise the corresponding bit and check whether the bit in the array is set. As S and T only grow, the loop will abort the next time line 4 is reached, once there is one node for which both bits are set.

We can use a similar structure for the test between S_R and T_R in line 5. S_R and T_R only grow as long as the true branch in lines 6 to 8 is not executed. Outside of the true branch, we can therefore use the same bit-vector trick to achieve an $O(1)$ test in line 5. Lines 6 to 8 consist of the code that augments the flow; i.e., they have a running time of $O(m)$ each time that the branch is executed. In $O(m)$ running time, we can reset the bit arrays, i.e., entering the true branch is unproblematic for the running time of the test in line 5. We can therefore account for the running time needed to manage the bit arrays in lines 6 to 8 and have an $O(1)$-test in line 5.

As already stated, lines 6 to 8 augment the flow and need $O(m)$ running time each time they are executed. Fortunately, there can be at most c path augmentations. The total time spent in lines 6 to 8 over the algorithm's execution is therefore in $O(cm)$.

In addition to maintaining the bit arrays for S_R and T_R, we can keep track of the number of elements in the sets. This allows us to implement the test in line 10 in $O(1)$.

Showing that the algorithm spends no more than $O(cm)$ running time in lines 11 to 15 and in the analogous lines 16 and 17 is the tricky part of the algorithm's analysis. Lines 16 and 17 follow directly by symmetry and therefore we focus on lines 11 to 15.

We will first establish that lines 10 to 17 are only executed at most m times. In each iteration, an arc is chosen as a piercing arc. After being chosen, an arc cannot participate in another cut and can therefore not be chosen a second time as a piercing arc. As there are only m arcs, the number of iterations is bounded by m.

For each of S, T, S_R, and T_R, we maintain the data structures of a breadth-first search,[3] i.e., a queue and a bit array of n elements. Growing a set as seen in lines 11 and 15 consists of removing nodes from the corresponding queue and visiting neighboring nodes until the queue is emptied, i.e., executing the regular breadth-first search algorithm. Adding a node to the set as seen in line 14 consists of adding the node to the queue and raising the corresponding in the bit array. It is thus clear that the operation in line 14 is in $O(1)$, and as there are at most m iterations, the total time spent in line 14 is in $O(m)$, which is below the claimed running time of $O(cm)$. The growing of the sets S and T is also in $O(m)$ as we never remove an element from S or T and they therefore consist of standard breadth-first searches. These searches are interrupted from time to time, but this does not change the fact that the total running time spent in them is in $O(m)$. Analyzing the running time required to grow the sets S_R and T_R is more difficult as the states of the associated searches can be reset in line 7. Fortunately, as we have already established, line 7 can only be executed at most c times. There are therefore only $O(c)$ state resets. Between two resets the search consists of a normal breadth-first search with a running time in $O(m)$. The total running time is therefore bounded by $O(cm)$.

We assumed that the piercing oracle requires a running time proportional to the number of arcs in the cut from which it must choose. The number of cut arcs never decreases. Further, there are c cut arcs at the end. We therefore know that c is an upper bound to the size of every intermediate cut. Further, as there are at most m iterations, we have bounded the total running time in line 13 by $O(cm)$.

It remains to show that line 12, which outputs the cuts, does not require more than $O(cm)$ running time. This seems trivial at first, but the details are significantly more involved than one would naively expect. Following the argumentation for line 13, we know that the operation must run in $O(c)$ running time to achieve a total running time of $O(cm)$. The algorithm must therefore output the cuts as a list of cut arcs and not as a bit array that maps each node to a side, as is often done in competitor algorithms. Outputting bit arrays would be too slow. Another problem consists of identifying the cut arcs efficiently. In $O(c)$ running time, the algorithm cannot iterate over all nodes in S or T to determine all outgoing arcs, which is needed to find the cut arcs in a straightforward way. The trick to achieve the required running time consists of maintaining two lists of saturated arcs. The first list consists of saturated arcs that depart in S and could be part of the cut. The second list consists of saturated arcs that enter T and work analogously. If the algorithm encounters a saturated arc when growing S in line 11, it adds the arc to the list of S. It does this regardless of whether the arc is a cut arc. When reaching line 12, this list contains all cut arcs but also possibly additional saturated arcs that are not part of the cut. The algorithm therefore iterates over all arcs before outputting them and removes those that are not cut arcs. This step can have a running time larger than $O(c)$. Fortunately, as every arc can only be added once to the list, it can also only be removed once. The total running time needed for the removal is therefore in $O(m)$ and we do not need to account for it in line 12. Further, after removing superfluous arcs, at

[3] A depth-first search would work too.

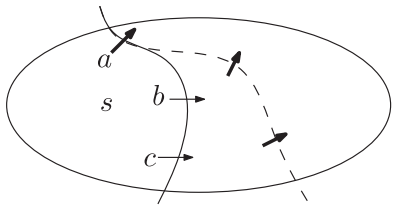


Fig. 2. The curves represent cuts; the current one is solid. The arrows are cut arcs; bold ones result in augmenting paths. The dashed cut is the next cut where piercing any arc results in an augmenting path.

most $O(c)$ remain, which is within the required bounds. This concludes the proof that the running time of our core algorithm is within $O(cm)$.

4.2 Piercing 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 is analogous and we therefore only describe the procedure for the source side. Denote by $a = (q, p)$ the piercing arc with piercing node $p \notin S$.

Our piercing heuristic is composed of two parts: the primary and the secondary heuristic. The primary heuristic first narrows down the set of potential piercing arcs and the secondary heuristic then chooses from this smaller set.

4.3 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 nonsaturated 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, the secondary heuristic is often only relevant in the case that an augmenting path is unavoidable. Consider the situation depicted in Figure 2. 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 therefore not T_R. Which piercing arcs result in larger cuts is therefore left unchanged. No matter whether b or c is picked, picking a in the next iteration results again in an augmenting path. The algorithm will therefore eventually end up with the same cut composed solely of arcs that should be avoided unless perfect balance is achieved first. This cut is represented as a dashed line in Figure 2. 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. It therefore does not matter which of these dominated cuts are enumerated and in which order they are found.

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. In this case, it is possible that perfect balance is achieved before the dashed cut is found. The result consists of a race between source and target sides to claim the last nodes. Not the best side wins, but the first that gets there.

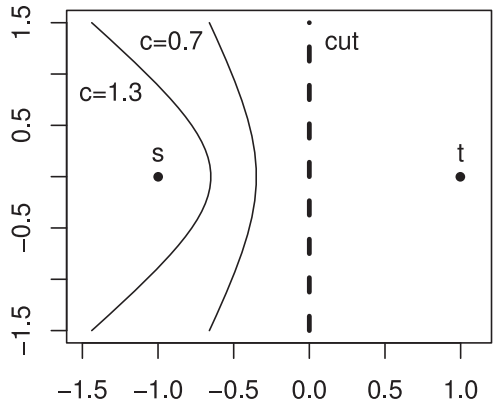


Fig. 3. Geometric interpretation of the distance heuristic.

4.4 Secondary Heuristic: Distance Based

If our primary avoid-augmenting-paths heuristic does not uniquely determine the piercing arc, we use a secondary distance heuristic to tie-break between the remaining choices. 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 the 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.

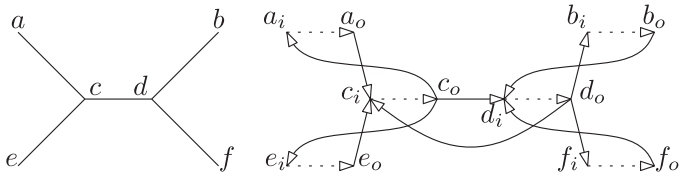
The distance heuristic has a geometric interpretation as depicted in Figure 3. We interpret the nodes as positions in the plane and the distances as being Euclidean. The set of points p for which $\|p-t\|_2 - \|p-s\|_2 = c$ holds for some constant c is one branch of a hyperbola whose two foci are s and t. 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. For example, cuts with a circle-like shape are problematic. This geometric interpretation also works in higher-dimensional spaces.

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 to compute contraction orders.

5.1 General Cuts

Our core algorithm computes balanced st-cuts. In many situations, cuts independent of a specific st-pair are needed. This problem variant can be solved with high probability by running Flow Cutter q times with st-pairs picked uniformly at random. Indeed, suppose that C is a Pareto-optimal cut such that the larger side has $\alpha \cdot n$ 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 number of pairs we recommend in our experiments, the success probability is larger than 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., $\epsilon = 80\%$)



(a) Input graph (b) Expanded graph

Fig. 4. Expansion of an undirected graph G into a directed graph G'. The dotted arrows are internal arcs. The solid arrows are external arcs.

and $q = 20$ the rate is still slightly above 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 $O(qcm)$. As we set q to a constant value of at most 100 in our experiments, the running time is in $O(cm)$.

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 the argument above because of effects induced by the properties of the piercing oracle.

5.2 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, 22]. 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 4. 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.

5.3 Contraction Orders

Our algorithm constructs contraction orders using an approach based on nested dissection [25, 36]. 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 unfortunately highly nontrivial.

The cuts produced by FlowCutter can be represented using a plot such as in Figure 5. Each point represents a nondominated cut. The question is which of the many points to choose. After some experimentation, we went with the following heuristic: pick a separator with minimum expansion and at most 60% imbalance. This choice is not perfect as the experiments of Section 6.4 show but works well in most cases. Picking a cut of minimum expansion given a Pareto cut set is

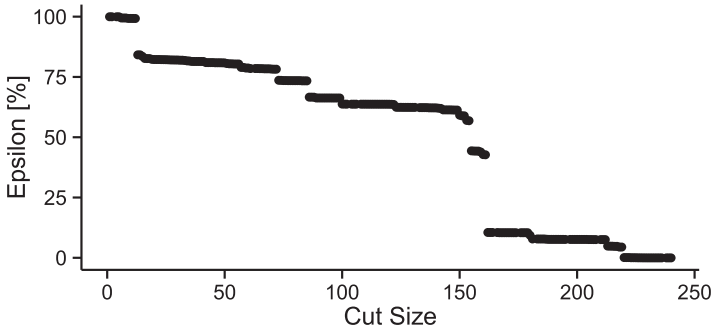


Fig. 5. Edges cuts found by FlowCutter with 20 random source-target pairs for the Central Europe graph used in the experiments.

easy. However, we know of no easy way to do this using an algorithm that computes a single small cut of bounded imbalance, as all the competitor algorithms do. It is therefore not easy to swap FlowCutter out for any of the competitors without also dropping expansion as an optimization criterion.

We continue the recursion until we obtain trees and cliques. On cliques, any order is optimal. On trees, an order can be derived from a so-called optimal node ranking as introduced in [33]. A node ranking of a tree is a labeling of the nodes with integers $1, 2 \dots k$ such that on the unique path between two nodes x and y with the same label, there exists a node z with a larger label. An optimal node ranking is one with minimum k. Contracting the nodes by increasing the label yields an elimination tree of minimum depth. In [41], it has been shown that these rankings can be computed in linear running time.

Special Preprocessing for Road Graphs. Road graphs have many nodes of degree 1 or 2. We exploit this in a fast preprocessing step similar to [18] to significantly reduce the graph size.

Our algorithm first determines the largest biconnected component B using [31] in linear time. It then removes all edges from G that leave B. It continues independently on every connected component of G as described in the next paragraph. The set of connected components usually consists of B and many tiny, often tree-like graphs. The resulting orders are concatenated such that the order of B is at the end. The other orders can be concatenated arbitrarily.

For each connected component, our algorithm starts by marking the nodes with a degree of 3 or higher. For each degree-2 chain $x, y_1 \dots y_k, z$ between two marked nodes x and z with $x \neq z$, it adds an edge $\{x, z\}$. It then splits the graph into the graph $G_{\geq 3}$ induced by the marked nodes and the graph $G_{\leq 2}$ induced by the unmarked nodes. Edges between marked and unmarked nodes are dropped. $G_{\leq 2}$ consists of the disjoint union of paths. As paths are trees, we can therefore employ the node-ranking-based tree-ordering algorithm described above to determine a node order. For $G_{\geq 3}$, we determine an order using FlowCutter and nested dissection as described above. We position the nodes of $G_{\leq 2}$ before those of $G_{\geq 3}$ and obtain the node order of the connected component.

6 EXPERIMENTS

We compare Flowcutter to the state-of-the-art partitioners KaHip [39], Metis [34], and InertialFlow [42]. There is a superficial comparison with PUNCH [16] in Section 6.3. We present three experiments: (1) we compare the produced contraction orders in terms of CCH performance in Section 6.2 on road graphs made available during the DIMACS challenge on shortest paths [17], (2) we compare the Pareto cut sets in more detail in Section 6.3 on the same road graphs, and (3) we evaluate

FlowCutter on nonroad graphs using the Walshaw benchmark set [44] in Section 6.5. Section 6.1 describes the experimental setup common to all experiments. All experiments were run on a Xeon E5-1630 v3 @ 3.70GHz with 128GB DDR4-2133 RAM.

6.1 Algorithm Implementations Used and Their Configurations

Edge Cut Algorithm. We use FlowCutter in three variants denoted by F3, F20, and F100, with three, 20, and 100 random source-target pairs, respectively. InertialFlow was introduced in [42], but no code was published. Fortunately, the idea is simple and we therefore were able to reimplement the algorithm. We refer to it using the letter I. Metis is a well-known general graph partitioner based on a multilevel scheme. The original authors published source code, which we used in our experiments. We compare against Metis 5.1.0, which is the newest version at the time of writing, and refer to it as M. KaHip also uses a multilevel scheme but adds a lot of optimizations compared to Metis that can drastically decrease the cut sizes. The KaHip source code is also available and thus we use it for our experiments. At the time of writing, the current version of KaHip is 1.00. Unfortunately, we have observed several regressions compared to earlier versions. These regressions are due to a bug being fixed that caused certain expensive flow-based refinement steps not being run for higher imbalances. The newer version achieves smaller cuts at the expense of higher running times for higher imbalances. Because of these regressions and for comparability with previous works, we also include comparisons with the earlier versions KaHip 0.61 and KaHip 0.73, which were the current versions and therefore used when we performed the experiments for [19] and [29], respectively. We use KaHip in the strong preconfiguration and add `--enforce_balance` to the commandline for $\max \epsilon = 0$. We refer to the three variants as K0.61, K0.73, and K1.00.

Node-Ordering Algorithms. Metis provides its own node-ordering tool called `ndmetis`, which we use. Unfortunately, no other package provides a similar tool. We have therefore implemented a nested dissection algorithm on top of them. For KaHip 1.00 and InertialFlow, we use the same straightforward nested dissection implementation that computes one edge cut at each level and recurses until either cliques or trees are reached. Edge cuts are transformed into node separators by picking the nodes on one side incident to the cut edges. For KaHip, we use a maximum imbalance of 20%, and for InertialFlow, we use 60%. For KaHip 0.61, we use an older nested dissection implementation originally written for [19]. It is not optimized for running time, only for quality. At each level, it invokes KaHip several times with different random seeds and picks the smallest cut found. It calls KaHip repeatedly on every level until for 10 consecutive calls no smaller cut is found. We do this to reliably get rid of variations in achieved cut sizes that are due to randomization. However, this setup is unfavorable to KaHip as it results in large running times. We decided to stick with the old ordering routine for K0.61 for comparability with [19] and use an ordering scheme for K1.0 that only computes one cut per level. The FlowCutter nested dissection implementation is based on the same code as used for KaHip 1.00 and InertialFlow but uses the separator variant of FlowCutter and performs the low-degree node optimizations described in Section 5.3.

KaHip v1.00 includes a more sophisticated tool to transform edge cuts into node separators using the algorithm of [40]. We tried using it in combination with the newer nested dissection scheme with one separator per level, but we needed 19 hours to compute orders for the small California and Nevada graph used in our experiments. We were not able to compute orders on the larger instances in reasonable time and therefore omit this algorithm from our comparison.

6.2 Order Experiments

We computed contraction orders for four DIMACS road graphs [17]. Our results are summarized in Table 1.

Table 1. Contraction Order Experiments

		Search Space				#Arcs		Up.	Running Times		
		Nodes		Arcs [$\cdot 10^3$]		in CCH	#Tri.	Tw.	Order	Cust.	Query
		Avg.	Max.	Avg.	Max.	[$\cdot 10^6$]	[$\cdot 10^6$]	Bd.	[s]	[ms]	[μs]
Col	M	155.6	354	6.1	22	1.4	6.4	102	**2.0**	18	26
	K0.61	135.1	357	4.6	22	1.7	7.2	103	3,837.1	21	20
	K1.00	136.4	357	4.8	22	1.5	6.9	99	1,052.4	20	20
	I	151.2	542	6.2	38	1.5	7.4	119	7.4	21	24
	F3	126.3	280	4.1	15	1.3	4.8	91	10.3	15	18
	F20	**122.4**	**262**	**3.8**	**14**	**1.3**	**4.4**	**87**	61.0	**14**	**17**
	F100	122.5	264	**3.8**	**14**	**1.3**	**4.4**	**87**	285.9	**14**	18
Cal	M	275.5	543	17.3	53	6.5	36.4	180	**9.9**	88	60
	K0.61	187.7	483	7.0	37	7.5	34.2	160	18,659.3	89	30
	K1.00	184.9	471	6.8	38	7.0	33.4	143	6,023.6	86	30
	I	191.4	605	7.1	53	6.9	34.1	161	42.6	84	31
	F3	177.5	**356**	6.2	**24**	5.9	23.4	**127**	64.1	69	27
	F20	170.0	380	**5.6**	26	**5.8**	**21.8**	132	386.8	68	**26**
	F100	**169.5**	380	**5.6**	26	**5.8**	**21.8**	132	1,831.8	**65**	**26**
Eur	M	1,223.4	1,983	441.4	933	69.9	1,390.4	926	**125.9**	2,241	1,164
	K0.61	638.6	1,224	114.3	284	73.9	578.2	482	213,091.1	971	303
	K1.00	652.5	1,279	113.4	287	68.3	574.5	451	242,680.5	934	297
	I	732.9	1,569	149.7	414	67.4	589.7	516	1,017.2	935	385
	F3	734.1	1,159	140.2	312	60.3	519.4	531	2,531.6	853	365
	F20	**616.0**	**1,102**	**102.8**	268	**58.8**	459.6	455	16,841.5	784	**270**
	F100	622.6	1,105	104.8	**239**	**58.8**	**459.4**	**449**	85,312.8	**766**	278
USA	M	990.9	1,685	249.1	633	86.0	1,241.1	676	**170.8**	2,111	651
	K0.61	575.5	1,041	71.3	185	97.9	737.1	366	265,567.3	1,250	202
	K1.00	540.3	1,063	62.3	208	88.7	648.3	439	315,942.6	1,097	179
	I	533.6	1,371	62.0	291	88.8	682.0	384	1,076.8	1,125	177
	F3	562.7	906	66.4	159	75.9	478.4	321	2,108.7	858	191
	F20	**490.6**	868	**52.7**	**154**	**74.3**	**440.5**	312	12,379.2	**812**	156
	F100	490.9	**863**	52.8	**154**	**74.3**	442.6	**311**	59,744.6	886	**155**

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 four cores. The customization times are the median over nine runs to eliminate variance. The query running times are averaged over 10^6 st-queries with s and t picked uniformly at random. Several CCH customization variants exist. The times reported are for a nonamortized, nonperfect customization, with SSE and using precomputed triangles.

Instances. The smallest test instance is the DIMACS Colorado graph 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 United States with $n = 24$M and $m = 57$M.

Relations between Columns. Table 1 contains a lot of data. However, some columns are related. We therefore first point these relations out and then limit our discussion to the remaining nonrelated columns. 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. These correlations are nonsurprising 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 therefore say that the search space is small or large without indicating whether we refer to nodes or arcs as one implies the other.

Search Spaces. One of the FlowCutter variants always produces the smallest search spaces. KaHip produces the next smaller search spaces, followed by InertialFlow. Metis is last by a large margin. It is interesting that the USA graph has a smaller search space than the Europe graph. The ratio between the average and the maximum search space sizes 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 countryside. However, compared to the competitors, the ratio is higher for InertialFlow. This illustrates that its geography-based heuristic is effective most of the time but in few cases fails noticeably at finding a good cut.

Number of Arcs. A small search space 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 size. InertialFlow seems to be comparable to Metis in terms of CCH size, as their CCH arc counts are never significantly different. However, FlowCutter beats all competitors and clearly achieves the smallest CCH sizes.

Number of 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 on all but the USA graph. On the USA graph, K1.0 is ahead of both InertialFlow and K0.61. FlowCutter also wins with respect to this quality metric, producing between 20% and 30% fewer triangles than the closest competitor.

Treewidth. As the CCH is essentially a chordal graph that is closely tied to tree decompositions, we can easily obtain upper bounds on the tree width 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 the worst, followed by InertialFlow, followed by KaHip, and FlowCutter has the best bounds. Analogous to the search space sizes, we observe that the USA graph has a significantly lower tree width than the Europe graph, assuming that our upper treewidth bounds are not completely off.

Running Time. Quality comes at a price, and thus the computation times of the orders follow nearly the opposite trend: KaHip is the slowest, followed by FlowCutter, followed by InertialFlow, while Metis is astonishingly fast.

K1.00 versus K0.61. The two KaHip versions seem to be very similar. Sometimes the newer version K1.00 is ahead and sometimes the older version K0.61 wins in terms of order quality. We explain this effect by differences in implementation in our nested dissection code. Recall that K0.61 takes the best cut of at least 10 iterations on each level, whereas K1.00 only computes a single cut. This means that K1.00 is more sensible to random fluctuations coming from bad random seeds

than K0.61. On average, one run of K1.00 is better than one run of K0.61. However, the best of at least 10 K0.61 runs wins against one K1.00 run with a bad seed. This effect explains the observed variance. Both the running times of K1.0 and K0.61 are very high, but for different reasons. K0.61 is slow because of the numerous repetitions on each level. However, K1.00 is slow because the newer KaHip version is significantly slower for $\epsilon = 20\%$ than the older versions. We will see this effect in greater detail in Section 6.3.

F3 versus F20 versus F100. It is not always clear which of F3, F20, or F100 gives the best results. F3 is most of the time slightly worse. This suggests that three source-target pairs are enough to get good separators most of the time but not enough to be fully reliable. A bad random seed can result in good separators being missed. The difference between F20 and F100 in terms of order quality is nearly negligible. This means that F20 and F100 find nearly always at least very similar separators. We conclude that there is no real advantage of going from 20 source-target pairs to 100 on road graphs. Twenty source-target pairs are enough to be quality-wise nearly independent of the random seed used.

6.3 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 five graphs in more detail. These are the DIMACS California and Nevada, Colorado, USA, and Europe graphs and a Central European subgraph.

Experimental Setup. For each of these graphs, we report the results in a table similar to Table 2. With the exception of FlowCutter, we ran each of the algorithms for various maximum imbalance input parameters (max ϵ column), effectively sampling the Pareto set computed by each partitioner. We report the imbalance of the produced cut. This achieved imbalance can be smaller than the input parameter, which is only a maximum. We further report the size of each cut and indicate whether both sides of the cut form connected subgraphs. Finally, we report the running time needed to compute each cut. To compute all reported cuts, i.e., the sampled Pareto set, all partitioners except FlowCutter need the sum over all reported running times.

For FlowCutter, we use a slightly different setup. We compute a set of Pareto cuts using FlowCutter and then pick the best cut from this set that has an imbalance below the requested maximum. This means that for FlowCutter, one can compute all reported cuts within the time needed to compute the cut for the input parameter max $\epsilon = 0$.

PUNCH. In [16], a competing algorithm named PUNCH was introduced. Unfortunately, we do not have access to an implementation of it. We therefore cannot perform experiments with this algorithm. However, for the USA and Europe graphs, the original authors [16] report cut sizes for an imbalance of 3%. In their experimental setup, PUNCH is run 100 times with varying random seeds. On average, PUNCH finds a cut with 130 edges for the Europe graph and 70 edges for the USA graph. The minimum cut size over 100 runs is 129 and 69 edges, respectively. K1.0 finds a cut with 130 edges for the Europe graph and 69 edges for the USA graph. We conclude that the performance of PUNCH is comparable in terms of quality to K1.0.

Instance Selection. Selecting meaningful and representative testing instances is difficult as can be seen from Table 2. For the imbalance between 20% and 50%, all partitioners with the exception of Metis find a cut of the same size. One can argue that this imbalance range is the most relevant for our application. It is therefore hard to argue, based on this experiment, whether one partitioner is better than another in terms of cut quality because they are all quasi the same. All cuts with 61 edges divide the United States along the Mississippi River into east and west. This cut is so

Table 2. Results for the DIMACS USA Graph

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.001	0.000	136	119	1,342	1,344	245	1,579
1	0.374	0.594	0.545	0.991	0.000	0.388	87	86	109	106	216	406
3	2.333	2.333	2.334	2.944	0.001	0.071	76	76	76	69	204	257
5	3.844	3.844	3.845	3.846	0.001	0.102	61	61	61	61	255	186
10	3.844	3.844	3.846	3.845	0.000	3.169	61	61	61	61	196	81
20	3.844	3.844	3.850	3.846	0.001	3.866	61	61	61	61	138	61
30	3.844	3.844	3.850	3.845	0.001	3.866	61	61	61	61	232	61
50	3.844	3.844	3.850	3.845	0.001	3.866	61	61	61	61	198	61
70	69.575	69.575	3.850	3.846	41.178	66.537	46	46	61	61	64,414	61
90	89.350	89.350	3.850	69.598	47.370	70.315	42	42	61	46	60,071	46
max ϵ	Running Time [s]						Are Sides Connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	297.7	1,902.0	2,489.1	2,560.7	12.2	15.7	●	●	○	○	●	○
1	264.1	1,717.6	274.7	279.0	12.1	23.6	●	●	●	○	○	○
3	240.9	1,584.2	720.8	665.0	12.2	31.7	●	●	●	○	●	○
5	208.0	1,377.5	1,262.3	1,251.1	12.4	35.5	●	●	●	●	○	○
10	208.0	1,377.5	2,073.7	2,715.5	12.4	29.7	●	●	●	●	●	●
20	208.0	1,377.5	249.0	3,463.3	12.2	45.6	●	●	●	●	●	●
30	208.0	1,377.5	249.1	4,176.1	12.3	64.8	●	●	●	●	●	●
50	208.0	1,377.5	248.8	3,702.3	12.4	100.7	●	●	●	●	●	●
70	156.8	1,056.2	249.6	4,047.7	12.9	158.7	●	●	●	●	○	●
90	144.3	965.2	249.2	6,359.3	12.8	201.1	●	●	●	●	○	●

pronounced that nearly all partitioners manage to find it. However, we cannot conclude from this experiment that all partitioners are interchangeable in terms of quality. This experiment only illustrates that the USA graph is in some sense an easy instance and therefore not a good testing instance. We therefore need to look at subgraphs of the United States to be able to observe the differences in quality, which definitely exist given the difference in contraction order qualities. We provide results for the DIMACS California and Nevada graph and the DIMACS Colorado graph in Tables 3 and 4. We also ran experiments on the DIMACS Europe graph. However, because of the special geographical topology of Europe, which we discuss in detail in Section 6.4, this graph is also nonrepresentative. We therefore evaluate the algorithms on a Central European subgraph induced by nodes with a latitude $\in [45, 52]$ and a longitude $\in [-2, 11]$. This subgraph has about $n = 7$M nodes and $m = 18$M arcs.

6.3.1 Discussion for USA. As already outlined, we cannot deduce much from the experimental results for the USA graph. However, there are a few observations that are interesting nonetheless. Most of these observations are also valid for all other test instances. We will therefore refrain from repeating these observations when discussing the other graphs.

Limitations of Metis. Metis is clearly dominated as it is the only partitioner unable to find the Mississippi. We can further observe that for imbalances of 70% and above, Metis finds huge cuts.

Table 3. Results for the DIMACS California and Nevada

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.000	0.000	42	39	157	174	51	306
1	0.169	0.169	0.184	1.000	0.000	0.566	31	31	31	36	52	93
3	2.293	2.293	2.300	2.303	0.001	1.112	29	29	29	29	61	64
5	2.293	2.293	2.293	2.329	0.005	1.571	29	29	29	29	42	62
10	2.293	2.293	2.304	2.294	0.001	0.642	29	29	29	29	43	37
20	2.293	16.706	2.756	2.293	0.000	2.656	29	28	30	29	41	29
30	2.293	16.706	2.768	2.293	13.936	5.484	29	28	29	29	51	29
50	2.293	49.058	2.768	2.296	0.000	40.833	29	24	29	29	39	27
70	64.522	49.058	2.768	2.296	41.178	42.591	27	24	29	29	4,310	26
90	87.953	89.838	2.768	82.592	47.370	85.555	20	14	29	19	3,711	18
max ϵ	Running Time [s]						Are Sides Connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	9.5	59.8	30.8	31.6	0.8	1.1	●	●	○	○	●	○
1	8.0	53.2	14.6	14.8	0.8	1.4	●	●	●	●	●	○
3	7.7	51.0	24.0	23.5	0.8	1.7	●	●	●	●	●	○
5	7.7	51.0	36.4	35.8	0.8	2.3	●	●	●	●	●	○
10	7.7	51.0	76.2	70.9	0.8	2.2	●	●	●	●	●	○
20	7.7	49.6	15.0	94.5	0.9	2.4	●	●	●	●	○	●
30	7.7	49.6	15.5	109.7	0.8	2.9	●	●	●	●	●	●
50	7.7	43.2	15.5	137.2	0.8	3.7	●	●	●	●	●	○
70	7.1	43.2	15.4	159.6	0.8	4.9	●	●	●	●	○	○
90	5.3	25.4	15.6	125.3	0.9	5.2	●	●	●	●	○	●

This is most likely a bug in the implementation. Further, while Metis does find a highly balanced cut, it is not perfectly balanced and therefore formally not a valid output for the case $\max \epsilon = 0$.

Limitations of KaHip. The running times of K0.73 are comparatively small for imbalances of 20% and higher. This is not the behavior that one would expect from the algorithm description. The running time is expected to grow with increasing imbalance as it does for K1.00. The reason for this behavior is the bug that was fixed in version 1.00. Before this version, KaHip would not do the flow-based refinement steps correctly. KaHip was therefore faster, but the achieved cuts can be very strange. This fixed bug is also the reason computing contraction orders with K1.00 is so slow.

Different Mississippi Cuts. Another interesting observation is that while nearly all partitioners are able to find a Mississippi cut, they find different variants of it. All cuts have size 61, but the achieved imbalances vary. FlowCutter finds slightly smaller imbalances than KaHip and InertialFlow. The cuts found by FlowCutter are therefore marginally better.

Connected Sides. FlowCutter guarantees by construction that both sides of each reported cut are connected. The other partitioners give no such guarantees. This means that the exact problem variants that they solve are slightly different. We therefore report for each of the other partitioners whether the cut they find happens to have connected sides. It is interesting that this is nearly

Table 4. Results for the DIMACS Colorado

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.001	0.000	37	37	74	49	40	259
1	0.308	0.277	0.970	0.023	0.002	0.088	31	29	34	29	39	96
3	0.308	0.277	2.999	0.553	0.000	0.748	31	29	29	28	51	70
5	0.308	4.263	4.290	0.553	0.025	0.897	31	28	27	28	40	60
10	0.308	9.073	9.467	0.550	0.001	1.413	31	23	23	28	47	46
20	17.664	19.995	11.761	18.842	16.671	13.984	22	19	22	19	376	27
30	22.784	27.606	12.249	27.737	23.080	23.125	18	14	20	14	521	21
50	22.784	40.630	9.772	40.630	42.409	36.365	18	12	23	12	14	14
70	22.784	57.602	12.000	40.630	41.177	48.771	18	11	23	12	1124	12
90	88.080	87.330	12.084	81.224	47.362	81.495	17	8	20	9	856	9
max ϵ	Running Time [s]						Are Sides Connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	2.0	12.1	4.5	3.6	0.1	0.2	●	●	○	○	●	○
1	1.7	9.9	2.8	2.7	0.2	0.3	●	●	●	●	●	○
3	1.7	9.9	4.0	4.4	0.2	0.3	●	●	●	●	●	○
5	1.7	9.6	5.1	7.1	0.1	0.3	●	●	●	●	●	○
10	1.7	8.1	9.1	15.7	0.2	0.3	●	●	●	●	○	○
20	1.3	6.8	3.2	16.5	0.2	0.3	●	●	●	●	○	●
30	1.1	5.2	3.0	23.3	0.2	0.4	●	●	●	●	○	●
50	1.1	4.5	3.4	35.2	0.1	0.4	●	●	●	●	●	○
70	1.1	4.2	3.5	40.8	0.2	0.5	●	●	●	●	○	●
90	1.1	3.1	3.5	24.4	0.2	0.6	●	●	●	●	○	●

always the case. One of the exceptions is, for example, the 3% imbalance of K1.00 with 69 edges. This cut is also the only situation where FlowCutter is outperformed in terms of cut size on the USA graph. However, the sides of this cut are not connected. The cut is therefore not a valid solution with respect to the exact problem setting solved by FlowCutter. This explains why it is not found.

Perfectly Balanced Cuts. Even though it is not useful for our application, it is interesting to compare the algorithms in terms of perfectly balanced cuts. This is the case when $\max \epsilon = 0$, or formulated differently: the number of nodes on each side must not differ by more than one node. Past research has partially focused on this special case. KaHip even includes a special postprocessing step called cycle refinement to reduce the sizes of perfectly balanced cuts [39]. The results are surprising. Metis is not able to find perfectly balanced cuts as the balance of the achieved cut is larger than required. For this border case, the achieved cuts are thus formally not valid. Even though KaHip includes special code, the achieved cut sizes are large. They even rival those of InertialFlow, a heuristic that in the case of perfect balance degenerates to sorting the nodes by longitude and cutting along the median. KaHip's cycle refinement clearly does not work on this kind of graph. Even though FlowCutter was not designed to compute perfectly balanced cuts, it is capable of doing so. Further, these cuts found turn out to be the smallest among all competitors by a significant margin.

6.3.2 Discussion for California and Nevada.

Perfectly Balanced Cuts. We include the DIMACS California and Nevada graph in our benchmark because [14] were able to determine the optimal size of a perfectly balanced cut for this graph. The optimum is 32 edges. The best cut found by the partitioners evaluated in Table 3 contains 39 edges and was found by FlowCutter. It is therefore off by seven edges. However, even with a slight imbalance, i.e., $\max \epsilon = 1\%$, F3, F20, and K0.73 are able to find a cut with 31 edges. As this cut is smaller than the smallest balanced cut, it is possible that this 31-edge cut is optimal.

Cut Sizes. The sizes of the cuts on California seem to be similar to those of the USA graph. There is one small and very pronounced cut, the one with 29 edges, which is found by all partitioners. However, F20 is able to find a 28- and 24-edge cut for higher imbalances. KaHip misses these cuts and sticks with the 29-edge cut. It is also interesting that InertialFlow is able to find a good 29-edge cut with a 2.7% imbalance. Unfortunately, it does not find it when the input parameter is at $\max \epsilon = 3\%$ but at $\max \epsilon = 20\%$. This means InertialFlow is capable of finding good cuts, but $\max \epsilon$ parameters that significantly differ from the desired ϵ have to be tried.

6.3.3 Discussion for Colorado.

Perfectly Balanced Cuts. The authors of [14] were also able to determine the minimum size of a perfectly balanced cut on the Colorado graph. It has 29 edges. While FlowCutter comes closest among all the evaluated partitioners, the cut found is again significantly larger by eight edges. For imbalances in the range of 1% to 3%, F20, K0.73, and K1.00 manage to achieve cut sizes of 29 edges, but no cut is perfectly balanced. All of them are therefore suboptimal. For $\epsilon = 5\%$, cuts smaller than 29 edges are found.

Cut Sizes. In contrast to the USA graph, we observe different cut sizes for the different partitioners on this instance for the relevant imbalances. We can therefore better deduce from this experiment whether a partitioner is better than another for our specific application. We observe that F20 wins with respect to every imbalance except for $\max \epsilon = 3\%$ and $\max \epsilon = 5\%$, where K1.00 and K0.73 respectively win by one edge. This demonstrates that FlowCutter is indeed a heuristic and does not always achieve the optimum. Comparing K1.00 and K0.73 is interesting. One could expect K1.00 to always win because it is the newer version, but this is not the case. For $\max \epsilon = 1\%$, K1.00 is five edges ahead, but for $\max \epsilon = 10\%$, K0.73 wins by five edges. This can mean that K1.00 is not always superior to K0.73. Another explanation is that both do not make enough iterations in their standard configuration to produce results that are reliable, i.e, with high probability insensitive to the random seed used. Rerunning K1.00 and K0.73 with different random seeds could change the outcome. The cut sizes of Metis are far from the competitors. InertialFlow is better than Metis but also clearly dominated.

Running Times. Metis and InertialFlow are by an order of magnitude faster but also compute worse cuts. The comparison between FlowCutter and KaHip is interesting. FlowCutter gets slower with a decreasing maximum imbalance. However, KaHip gets slower with an increasing maximum imbalance, i.e., the other way round. A clear ranking is therefore not possible, but the tendency for max imbalances above 10% is that F3 is the fastest, followed by K0.73, followed by F20, and finally K1.00.

6.3.4 Discussion for Central Europe.

Cut Sizes. In Table 5, we report the results of our experiments for the Central Europe graph. The most striking observation is that the cut sizes in this graph are larger than those in any of the USA graphs. This explains why the Europe graph has a higher tree width and larger search spaces than the USA graph. It is not immediately clear which cut is the best for our application; however, the cuts with sizes 180, 162, and 155 seem to offer a good tradeoff

Table 5. Results for the DIMACS Central Europe Graph

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.000	0.000	292	240	716	674	369	1,180
1	0.232	0.132	0.998	0.916	0.000	0.089	275	220	245	216	360	391
3	0.232	0.132	0.457	2.086	0.000	0.008	275	220	227	207	372	319
5	4.963	4.894	0.464	1.470	0.000	0.857	271	213	227	208	369	276
10	6.914	9.330	0.043	8.862	0.000	0.375	243	180	228	207	375	241
20	19.419	10.542	3.139	10.546	0.000	0.132	225	162	250	162	375	220
30	19.419	10.542	3.139	10.543	0.017	7.384	225	162	250	162	369	203
50	19.419	44.386	3.139	10.547	33.336	10.542	225	155	250	162	9,881	162
70	63.775	66.655	3.139	10.547	41.178	44.386	100	86	250	162	14,375	155
90	84.199	84.199	3.139	10.544	83.087	84.257	13	13	250	162	28	17
max ϵ	Running Time [s]						Are Sides Connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	231.4	1,390.3	369.1	315.9	3.3	4.3	●	●	○	○	●	○
1	230.2	1,342.9	80.2	72.2	3.3	7.9	●	●	●	●	●	○
3	230.2	1,342.9	112.5	111.7	3.1	10.2	●	●	●	●	●	○
5	229.9	1,319.0	158.3	206.5	3.3	12.3	●	●	●	●	●	●
10	225.0	1,181.5	338.1	455.1	3.1	16.8	●	●	●	●	●	○
20	215.7	1,089.5	75.5	355.4	3.1	25.6	●	●	●	●	○	●
30	215.7	1,089.5	75.4	395.9	3.1	34.9	●	●	●	●	●	●
50	215.7	1,047.8	75.3	467.4	3.2	47.5	●	●	●	●	○	●
70	101.8	591.6	75.5	560.4	3.2	82.8	●	●	●	●	○	●
90	13.8	92.8	75.4	633.0	3.3	17.1	●	●	●	●	●	○

between cut size and imbalance. F20 manages to find all of them. K1.00 finds a variant of the 162-edge cut with a marginally higher imbalance. InertialFlow is able to find the 162- and the 155-edge cuts. Unfortunately, as already previously observed, we need to set the max ϵ parameter significantly higher than the imbalance of the cuts for InertialFlow to find them.

Running Times. On all of the USA graphs, F20 was at least on par with K1.00 in terms of running time and often even faster. On this graph, we see a significant gap of at least a factor 2 for all imbalances below 70%. The explanation is that the running time of FlowCutter depends not only on the graph size but also on the cut size. As this graph has larger cuts than the USA graphs, FlowCutter is slower. KaHip's running time is not or is at least less affected by cut size and therefore comes out ahead on this graph. However, for our particular application, i.e., nested dissection, a running time sensitive to the cut size is a good thing. We have only a few top-level cuts with large cuts but many more low-level cuts that have tiny cuts. A partitioner that gets faster the smaller the cuts become is therefore useful in this scenario as it gets faster on the lower levels. This observation also explains why F20 wins against K1.00 in terms of running time on the Europe graph in the CCH experiment.

Table 6. Results for the DIMACS Europe

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.003	0.000	369	276	1,296	1,299	402	1,579
1	0.930	0.930	1.000	0.984	0.003	0.337	234	234	169	154	398	417
3	2.244	2.244	2.717	2.654	0.003	0.357	221	221	130	130	306	340
5	2.244	4.918	2.976	2.985	0.003	0.171	221	216	129	129	276	299
10	9.453	9.453	8.092	7.875	0.003	0.174	188	188	112	112	460	284
20	9.453	9.453	9.405	7.888	0.003	7.539	188	188	126	112	483	229
30	9.453	9.453	9.232	8.216	0.003	9.060	188	188	128	111	465	202
50	9.453	42.080	9.232	8.214	33.336	9.453	188	58	128	111	31,127	188
70	64.477	67.497	9.232	32.079	41.178	64.724	58	22	128	86	53,365	38
90	72.753	72.753	9.232	72.753	70.741	72.753	2	2	128	2	44	2

max ϵ	Running Time [s]						Are Sides Connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	508.4	3,475.5	1,887.5	1,893.9	8.9	11.3	●	●	○	○	○	○
1	468.6	3,292.7	224.7	196.9	8.9	19.0	●	●	○	●	●	○
3	455.7	3,215.0	317.5	303.3	8.9	28.0	●	●	●	●	○	○
5	455.7	3,181.7	510.2	524.8	8.9	33.6	●	●	●	●	○	○
10	411.5	2,913.3	934.0	1,419.1	9.0	49.3	●	●	○	○	○	●
20	411.5	2,913.3	198.8	1,646.2	8.9	69.9	●	●	○	○	○	○
30	411.5	2,913.3	193.6	1,569.3	8.9	94.0	●	●	○	○	●	●
50	411.5	949.4	194.1	1,727.7	9.1	172.9	●	●	○	○	○	●
70	134.4	371.9	193.9	1,642.2	9.4	79.0	●	●	○	○	○	●
90	7.6	51.9	194.1	3,411.3	9.0	18.9	●	●	○	●	○	●

(a) K0.76 (b) F with guidance (c) F20

Fig. 6. Various top-level Europe cuts.

6.4 Special Structure of the Europe Graph

We present the results for the Europe graph in Table 6. The reported cut sizes do not follow the pattern observed in the other graphs. The cuts of F20 are significantly larger than those of KaHip. Another observation is that most of the cuts found by partitioners except FlowCutter do not have connected sides. This already hints at the root of the problem. In Figure 6, we visualized the cuts found. Figure 6(a) depicts the cut found by KaHip with 112 edges, and Figure 6(c) depicts the cut

found by F20 with 188 edges. Visually these two cuts look very different. To explain the effect in detail, we must first describe some properties of the Europe graph.

Unique Geography. Top-level Europe has a unique geographic topology. There is a well-connected center formed by France, Germany, Belgium, Luxembourg, and the Netherlands. Further, there are four peninsulas. Spain and Portugal are only connected by a comparatively small piece of land with France. Italy is separated from the rest of Europe by the Alps. Sweden and Norway are separated by the Baltic Sea from Central Europe. They are only connected to Denmark by a highway bridge in Copenhagen. This bridge is also the cut with two edges with 72% imbalance found by several partitioners. Great Britain is separated by the North Sea and is only connected to the continent using ferries, which are treated as roads in the benchmark dataset. There are further ferries between Spain and England, and between Spain and Italy. However, there are no ferries from or to Scandinavia.

Structure of KaHip Cuts. The KaHip cut with 112 edges separates Central Europe from its peninsulas. The sides are not connected because there is no path from Great Britain to Scandinavia. The KaHip cut with 129 edges with connected sides further separates Denmark from Germany. The sides of the cut are connected, as there is a ferry from England to Denmark and a bridge from Denmark to Sweden.

Structure of FlowCutter Cuts. The FlowCutter cut is structurally very different. FlowCutter separates Central Europe along the Rhine River and the Alps. FlowCutter cannot find the 112-edge cut because its sides are not connected. Further, it does not find the 129-edge cut because the shape of this cut is very different from what the employed piercing heuristic expects.

KaHip versus FlowCutter. At first glance, KaHip seems to be better than FlowCutter on this instance. However, this is not consistent with our observation that FlowCutter produces better contraction orders. The explanation is that, as we consider a recursive bisection, the question is not whether Central Europe must be cut along the Rhine River, but at which recursion level we do it. FlowCutter does it at the top level, whereas KaHip does it at a lower level. It is unclear which approach is better. We will investigate this question in detail below. However, before we answer this question, we explore how we can modify FlowCutter to find a cut similar to the one found by KaHip.

Adapting FlowCutter. One can regard the balanced 112-edge cut of KaHip as a union of four smaller edge cuts with higher imbalances. There is one cut for each peninsula. Repeatedly cutting each peninsula on consecutive levels of the recursion is equivalent with cutting them all in one level. The question is therefore whether FlowCutter is able to find one of the peninsula cuts, and this is indeed the case. FlowCutter finds the cut with two edges that separates Scandinavia from the rest. However, FlowCutter refrains from choosing this cut from the Pareto set because we have a hard bound on a maximum imbalance of at most 60%.

Another option to help FlowCutter is to handpick source and target nodes. We selected the nodes that are closed to the coordinates given in Table 7 and used these as input to FlowCutter. These coordinates are not magic numbers. They represent positions chosen at the extremities of the peninsulas and in the center of Central Europe. Most humans are able to deduce this information from looking at a Europe map. With this setup, we were not able to find the 112-edge cut with 7.9% imbalance found by KaHip. However, we were able to find another cut with a seemingly better tradeoff. This new cut is depicted in Figure 6(b). It has 87 edges and a 15% imbalance. The smaller cut results from placing Austria on the other side of the cut compared to the 112-edge cut of KaHip and from some minor improvements along the other borders. KaHip is incapable of finding this cut.

Table 7. Handpicked Source and Target Nodes

	Lat	Lon	Place
Source	49.0	8.4	Karlsruhe
Target	41.0	16.9	Bari
	38.7	−9.1	Lisbon
	53.5	−2.8	Liverpool
	59.2	18.0	Stockholm

Table 8. Contraction Order Experiments on the DIMACS Europe Graph

	Search Space		#Arcs			Up.	
	Nodes		Arcs [$\cdot 10^3$]		in CCH	#Tri.	Tw.
	Avg.	Max.	Avg.	Max.	[$\cdot 10^6$]	[$\cdot 10^6$]	Bd.
F3	734.1	1,159	140.2	312	60.3	519.4	531
F20	616.0	1,102	102.8	268	**58.8**	459.6	455
F100	622.6	1,105	104.8	**239**	**58.8**	459.4	449
F3+H	625.1	1,151	106.2	262	60.2	509.2	**439**
F20+H	601.2	**1,064**	98.9	261	**58.8**	456.9	444
F100+H	**600.6**	1,065	**98.6**	250	**58.8**	**454.4**	444

F3, F20, and F100 are the default FlowCutter variants that use a top-level cut along the Rhine River. F3+H, F20+H, and F100+H use a handpicked top-level cut separating Central Europe from the peninsulas.

The Best Top-Level Cut. We have shown that with a bit of help, it is possible to push FlowCutter towards computing a small cut that separates the peninsulas. Now, we will answer the question whether this a better top-level cut than the 188-edge cut found by the default FlowCutter configuration. We derive an 87-node separator from the 87-edge cut and place these nodes at the end of the contraction order manually. We then run FlowCutter on the resulting sides recursively without any further manual guidance. In Table 8, we report the characteristics of the so-obtained orders. The new orders are marked with "+H", indicating human interaction. We compare them with the default FlowCutter orders. The new orders seem to be slightly superior with respect to every criterion except the maximum number of arcs in the search space, where the default FlowCutter orders seem to win. Further, the orders seem to produce a similar number of edges in the CCH regardless of the top-level cut used. However, the differences in order quality are very minor. We observe with respect to no criterion a difference that is larger than 2%. This difference can be due to a peninsula top-level cut being slightly better. However, another explanation is that FlowCutter finds better cuts on the lower levels because a difficult-to-find peninsula cut was eliminated manually. In either case, the differences are so small that we decided that it is not worthwhile to automatize the selection of a top-level peninsula cut.

6.5 Walshaw Benchmark Set

A popular set of graph partitioning benchmark instances is maintained by Walshaw [44]. The data contains 34 graphs and solutions to the edge-bisection problem with nonconnected sides and maximum imbalance values of $\epsilon = 0\%$, $\epsilon = 1\%$, $\epsilon = 3\%$, and $\epsilon = 5\%$. These archived solutions are the best cuts that any partitioner has found so far. A few of them were even proven to be optimal [14]. Comparing against these archived solutions allows us to compare FlowCutter quality-wise against the state of the art. We want to stress that this state of the art was computed by a large mixture of algorithms with an even larger set of parameters that may have been chosen in instance-dependent

Table 9. Performance on the Walshaw Benchmark Set, Part 1

Graph	Algorithm	Minimum Edges in Cut For				Running Time [s]
		$\epsilon = 0\%$	$\epsilon = 1\%$	$\epsilon = 3\%$	$\epsilon = 5\%$	
144	F20	6,649	6,608	6,514	6,472	2,423.82
144K nodes	F100	6,515	6,479	6,456	6,366	10,437.91
1,074K edges	Reference	**6,486**	**6,478**	**6,432**	**6,345**	
3elt	F20	**90***	**89**	**87**	**87**	0.36
4720 nodes	F100	**90***	**89**	**87**	**87**	1.87
13K edges	Reference	**90***	**89**	**87**	**87**	
4elt	F20	149	**138**	**137**	**137**	1.97
15K nodes	F100	**139***	**138**	**137**	**137**	9.50
45K edges	Reference	**139***	**138**	**137**	**137**	
598a	F20	2,417	2,390	**2,367**	**2,336**	545.69
110K nodes	F100	2,400	**2,388**	**2,367**	**2,336**	2,675.32
741K edges	Reference	**2,398**	**2,388**	**2,367**	**2,336**	
auto	F20	10,609	10,283	9,890	**9,450**	13,445.66
448K nodes	F100	10,549	10,283	9,823	**9,450**	66,249.82
3314K edges	Reference	**10,103**	**9,949**	**9,673**	**9,450**	
bcsstk30	F20	6,454	6,347	**6,251**	**6,251**	245.65
28K nodes	F100	6,408	6,347	**6,251**	**6,251**	1,230.27
1007K edges	Reference	**6,394**	**6,335**	**6,251**	**6,251**	
bcsstk33	F20	10,220	**10,097**	**10,064**	**9,914**	118.38
8738 nodes	F100	10,177	**10,097**	**10,064**	**9,914**	573.02
291K edges	Reference	**10,171**	**10,097**	**10,064**	**9,914**	
brack2	F20	742	**708**	**684**	**660**	58.13
62K nodes	F100	742	**708**	**684**	**660**	283.99
366K edges	Reference	**731***	**708**	**684**	**660**	
crack	F20	**184**	**183**	**182**	**182**	2.17
10K nodes	F100	**184**	**183**	**182**	**182**	10.97
30K edges	Reference	**184**	**183**	**182**	**182**	
cs4	F20	381	371	367	360	11.68
22K nodes	F100	372	370	365	357	58.11
43K edges	Reference	**369**	**366**	**360**	**353**	
cti	F20	342	**318**	**318**	**318**	6.10
16K nodes	F100	339	**318**	**318**	**318**	30.55
48K edges	Reference	334	**318**	**318**	**318**	
fe_4elt2	F20	**130***	**130**	**130**	**130**	1.86
11K nodes	F100	**130***	**130**	**130**	**130**	9.19
32K edges	Reference	**130***	**130**	**130**	**130**	

"Reference" is the best-known bisection for the graph as maintained by Walshaw. A "*" marks solutions for which optimality has been shown.

ways. We compare this against a single algorithm with a single set of parameters. Further, FlowCutter was designed for higher imbalances than 5%. It was not tuned for the cases with a lower imbalance. FlowCutter only computes cuts with connected sides. We therefore filter out all graphs that are either not connected or where the archived $\epsilon = 0$-solution has nonconnected sides. Of the 34 graphs, only 24 remain. The results are reported in Tables 9 and 10.

Table 10. Performance on the Walshaw Benchmark Set, Part 2

Graph	Algorithm	Minimum Edges in Cut For				Running
		$\epsilon = 0\%$	$\epsilon = 1\%$	$\epsilon = 3\%$	$\epsilon = 5\%$	Time [s]
fe_ocean	FlowCutter 20	504	431	**311**	**311**	89.70
143K nodes	FlowCutter 100	483	408	**311**	**311**	418.60
409K edges	Reference	**464**	**387**	**311**	**311**	
fe_rotor	FlowCutter 20	2,115	2,091	**1,959**	1,948	334.58
99K nodes	FlowCutter 100	2,106	2,067	**1,959**	**1,940**	1,636.78
662K edges	Reference	**2,098**	**2,031**	**1,959**	**1,940**	
fe_sphere	FlowCutter 20	**386**	**386**	**384**	**384**	5.98
16K nodes	FlowCutter 100	**386**	**386**	**384**	**384**	30.84
49K edges	Reference	**386**	**386**	**384**	**384**	
fe_tooth	FlowCutter 20	3,852	3,841	3,814	**3,773**	413.48
78K nodes	FlowCutter 100	3,836	3,832	3,790	**3,773**	2,067.54
452K edges	Reference	**3,816**	**3,814**	**3,788**	**3,773**	
finan512	FlowCutter 20	**162***	**162**	**162**	**162**	8.11
74K nodes	FlowCutter 100	**162***	**162**	**162**	**162**	39.01
261K edges	Reference	**162***	**162**	**162**	**162**	
m14b	FlowCutter 20	3,858	**3,826**	**3,823**	3,805	2,115.07
214K nodes	FlowCutter 100	**3,836**	**3,826**	**3,823**	3,804	10,512.24
1679K edges	Reference	**3,836**	**3,826**	**3,823**	**3,802**	
t60k	FlowCutter 20	80	79	73	**65**	2.98
60K nodes	FlowCutter 100	80	77	**71**	**65**	14.55
89K edges	Reference	**79**	**75**	**71**	**65**	
vibrobox	FlowCutter 20	10,614	10,356	10,356	10,356	139.90
12K nodes	FlowCutter 100	10,365	**10,310**	**10,310**	**10,310**	680.76
165K edges	Reference	**10,343**	**10,310**	**10,310**	**10,310**	
wave	FlowCutter 20	8,734	8,734	8,734	8,724	2,723.12
156K nodes	FlowCutter 100	8,716	8,673	8,650	8,590	13,583.59
1059K edges	Reference	**8,677**	**8,657**	**8,591**	**8,524**	
whitaker3	FlowCutter 20	**127***	**126**	**126**	**126**	1.49
9800 nodes	FlowCutter 100	**127***	**126**	**126**	**126**	7.00
28K edges	Reference	**127***	**126**	**126**	**126**	
wing	FlowCutter 20	790	790	790	790	80.11
62K nodes	FlowCutter 100	790	790	781	773	401.82
121K edges	Reference	**789**	**784**	**773**	**770**	
wing_nodal	FlowCutter 20	1,767	1,764	1,715	1,691	27.02
10K nodes	FlowCutter 100	1,743	1,740	1,710	1,688	134.05
75K edges	Reference	**1,707**	**1,695**	**1,678**	**1,668**	

For $\epsilon = 5\%$, there are only six graphs where FlowCutter does not match the best-known cut quality. These are "144," "cs4," "m14b," "wave," "wing," and "wing_nodal." For three of these graphs, FlowCutter finds cuts that are larger by a negligible amount of at most five edges. For the other three, the cuts found are larger but are still close to the best-known solutions. For lower imbalances, the results are not quite as good but still very close to the best-known solutions.

In terms of running time, the results are more mixed. Some cuts are found very quickly, while FlowCutter needs a significant amount of time on others. This is due to the fact that its running

time is in $O(cm)$. If both the cut size c and the edge count m are large, then $O(cm)$ is large. However, for graphs with small cuts, the algorithm scales nearly linearly in the graph size.

7 CONCLUSION

We introduce FlowCutter, a graph bisection algorithm that optimizes balance and cut size in the Pareto sense. The core algorithm computes small, balanced edge cuts separating two input nodes s and t. Upon this core algorithm, we build algorithms to compute overall small, balanced edges cuts independent of an st-pair specified in the input. We further extend our algorithm to compute small balanced node separators. By combining FlowCutter with a nested dissection-based strategy, we compute contraction orders (also called elimination or minimum fill-in orders). We show that our orders beat the state of the art in terms of quality on road graphs. We evaluate the quality of our orders by directly applying them in the context of Customizable Contraction Hierarchies, a speedup technique for shortest paths. Further, we show that FlowCutter manages to equate the best-known cuts for many instances of the Walshaw benchmark set, demonstrating that FlowCutter is applicable beyond just bisecting road graphs. Finally, we use FlowCutter to compute tree decompositions of small width. To evaluate the performance of our method, we submitted FlowCutter to the PACE2016 challenge [13], where it won first place in the corresponding sequential track. This demonstrates that FlowCutter works well on a broad class of graphs. The source code of the PACE 2016 submission is available at [45].

Future Work. We show that FlowCutter is an excellent tool to be used within shortest-path acceleration techniques on road graphs. Luckily, the developed techniques seem also useful in other domains. Experimental evaluations that take domain-specific requirements into account would be an interesting venue for future research. For example, our secondary piercing heuristic could be swapped out with one that uses information that is only available in certain applications.

Another open question is how to adapt FlowCutter to graphs that have weighted nodes or weighted edges or even both.

We use a nested dissection scheme to compute contraction orders with FlowCutter as a bisection algorithm. This seems somewhat wasteful as in this setup FlowCutter computes a large set of cuts, but we only use one of them and then discard the others. In the deeper levels of the recursion, FlowCutter will then likely recompute some of the discarded cuts. Adapting the nested dissection scheme in a way that utilizes several cuts from each set could significantly improve the running time.

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.

ACKNOWLEDGMENTS

We thank Roland Glantz for helpful discussions. This work is partially supported by the DFG under grants WA654/19-1 and WA654/22-1.

REFERENCES

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

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

[3] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2016. *Route Planning in Transportation Networks*. Technical Report abs/1504.05140. ArXiv e-prints. https://link.springer.com/chapter/10.1007/978-3-319-49487-6_2 To appear in LNCS Volume on Algorithm Engineering, Lasse Kliemann and Peter Sanders (eds.).

[4] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. 2010. Preprocessing speed-up techniques is hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10) (Lecture Notes in Computer Science)*, Vol. 6078. Springer, 359–370.

[5] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. 2013. Search-space size in contraction hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13) (Lecture Notes in Computer Science)*, Vol. 7965. Springer, 93–104.

[6] Philip A. Bernstein and Nathan Goodman. 1981. Power of natural semijoins. *SIAM Journal on Computing* 10, 4 (1981), 751–771.

[7] Jean Blair and Barry Peyton. 1993. An introduction to chordal graphs and clique trees. In *Graph Theory and Sparse Matrix Computation*. The IMA Volumes in Mathematics and Its Applications, Vol. 56. Springer, 1–29.

[8] Hans L. Bodlaender. 1993. A tourist guide through treewidth. *Acta Cybernetica* 11 (1993), 1–21.

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

[10] Hans L. Bodlaender, John R. Gilbert, Hjalmtyr Hafsteinsson, and Ton Kloks. 1995. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms* 18, 2 (March 1995), 238–255.

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

[12] Bruno Courcelle. 1990. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation* 85, 1 (March 1990), 12–75.

[13] Holger Dell, Thore Husfeldt, Bart M. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances Rosamond. 2016. The first parameterized algorithms and computational experiments challenge. In *11th International Symposium on Parameterized and Exact Computation (Leibniz International Proceedings in Informatics)*. 30:1–30:9. DOI: http://dx.doi.org/10.4230/LIPIcs.IPEC.2016.30

[14] Daniel Delling, Daniel Fleischman, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. 2015. An exact combinatorial algorithm for minimum graph bisection. *Mathematical Programming* 153, 2 (November 2015), 417–458.

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

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

[17] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (Eds.). 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Book, Vol. 74. American Mathematical Society.

[18] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2015. Fast exact shortest path and distance queries on road networks with parametrized costs. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 66:1–66:4.

[19] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics* 21, 1 (April 2016), 1.5:1–1.5:49. DOI: http://doi.acm.org/10.1145/2886843

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

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

[22] Lester R. Ford, Jr. and Delbert R. Fulkerson. 1962. *Flows in Networks*. Princeton University Press.

[23] Delbert R. Fulkerson and O. A. Gross. 1965. Incidence matrices and interval graphs. *Pacific Journal of Mathematics* 15, 3 (1965), 835–855.

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

[25] Alan George. 1973. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis* 10, 2 (1973), 345–363.

[26] Alan George and Joseph W. Liu. 1989. The evolution of the minimum degree ordering algorithm. *SIAM Reviews* 31, 1 (1989), 1–19.

[27] John R. Gilbert and Robert Tarjan. 1986. The analysis of a nested dissection algorithm. *Numerische Mathematik* 50, 4 (July 1986), 377–404.

[28] Michael Hamann and Ben Strasser. 2015. *Graph Bisection with Pareto-Optimization*. Technical Report. ArXiv e-prints. http://arxiv.org/abs/1504.03812

[29] Michael Hamann and Ben Strasser. 2016. Graph bisection with Pareto-optimization. In *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*. SIAM, 90–102.

[30] Martin Holzer, Frank Schulz, and Dorothea Wagner. 2008. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics* 13, 2.5 (December 2008), 1–26.
[31] John E. Hopcroft and Robert E. Tarjan. 1973. Efficient algorithms for graph manipulation. *Communications of the ACM* 16, 6 (June 1973), 372–378.
[32] Cecil Huang and Adnan Darwiche. 1996. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning* 15, 3 (October 1996), 225–263.
[33] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. 1988. Optimal node ranking of trees. *Information Processing Letters* 28, 5 (August 1988), 225–229.
[34] George Karypis and Vipin Kumar. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1999), 359–392. http://dx.doi.org/10.1137/S1064827595287997
[35] Frank Thomson Leighton and Satish Rao. 1999. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM* 46, 6 (1999), 787–832. http://portal.acm.org/citation.cfm?doid=331524.331526
[36] Richard J. Lipton, Donald J. Rose, and Robert Tarjan. 1979. Generalized nested dissection. *SIAM Journal on Numerical Analysis* 16, 2 (April 1979), 346–358.
[37] Harry M. Markowitz. 1957. The elimination form of the inverse and its application to linear programming. *Management Science* 3, 3 (April 1957), 255–269.
[38] Alex Pothen. 1988. *The Complexity of Optimal Elimination Trees.* Technical Report. Pennsylvania State University.
[39] Peter Sanders and Christian Schulz. 2013. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13) (Lecture Notes in Computer Science)*, Vol. 7933. Springer, 164–175.
[40] Peter Sanders and Christian Schulz. 2016. Advanced multilevel node separator algorithms. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16) (Lecture Notes in Computer Science)*, Vol. 9685. Springer, 294–309.
[41] Alejandro A. Schæffer. 1989. Optimal node ranking of trees in linear time. *Information Processing Letters* 33 (November 1989), 91–96.
[42] Aaron Schild and Christian Sommer. 2015. On balanced separators in road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15) (Lecture Notes in Computer Science)*. Springer, 286–297.
[43] Frank Schulz, Dorothea Wagner, and Karsten Weihe. 2000. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics* 5, 12 (2000), 1–23.
[44] A. J. Soper, Chris Walshaw, and Mark Cross. 2004. A combined evolutionary search and multilevel optimisation approach to graph partitioning. *Journal of Global Optimization* 29, 2 (2004), 225–241. Retrieved from http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/#SoperJoGO04.
[45] Ben Strasser. 2016. Source code of PACE 2016 FlowCutter submission. Uploaded to github at https://github.com/ben-strasser/flow-cutter-pace16.
[46] Dorothea Wagner and Frank Wagner. 1993. Between min cut and graph bisection. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS'93) (Lecture Notes in Computer Science)*, Vol. 711. Springer, London, 744–750.
[47] Michael Wegner. 2014. *Finding Small Node Separators.* Bachelor's thesis. Karlsruhe Institute of Technology.

Received September 2016; revised October 2017; accepted December 2017