Bachelor Thesis

Finding Small Node Separators

Michael Wegner

Submission date: October 2, 2014

Supervisors: Prof. Dr. Peter Sanders
Dr. Christian Schulz

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Errata

The experiments in this thesis are not described correctly. More precisely, the node separator algorithm presented in this thesis has been repeated five times (instead of one single time) and the best result has been taken.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 2. Oktober 2014

Abstract

We present a new algorithm to compute small node separators on large, undirected graphs ensuring a user defined balance on the induced connected components. Our algorithm uses edge separators computed by the open source graph partitioning package KaHIP [42] and a new technique to transform them into node separators. Given an undirected graph G, our algorithm constructs a flow problem on a subgraph of G containing the edge separator so that the induced minimum cut in this subgraph corresponds to a node separator in G. Experiments show that our algorithm finds significantly better node separators than other popular tools. The speed of our algorithm is some orders of magnitude slower than other tools, however, parallelizing the algorithm and optimizing some crucial parts should decrease the running time without too much effort. To further evaluate the quality of our node separators we implemented a node ordering algorithm based on *nested dissection* [12, 13, 25] which uses our separators to compute high quality node orderings.

Zusammenfassung

Wir stellen einen neuen Algorithmus zum Finden von Knotenseparatoren auf großen, ungerichteten Graphen vor, der eine vom Benutzer geforderte Balance zwischen den induzierten Partitionen sicherstellt. Unser Algorithmus verwendet vom Open Source Graphpartitionierungspaket KaHIP [42] berechnete Kantenseparatoren und ein neues Verfahren um diese in Knotenseparatoren umzuwandeln. Für einen ungerichteten Graphen G konstruiert unser Algorithmus ein Flussproblem auf einem Subgraphen von G der den Kantenseparator enthält, sodass der induzierte minimale Schnitt in diesem Subgraphen einem Knotenseparator in G entspricht. Experimente zeigen, dass unser Algorithmus im Vergleich zu anderen bekannten Algorithmen für dieses Problem signifikant bessere Knotenseparatoren berechnet. Die Laufzeit unseres Algorithmus ist im Vergleich zu den anderen Algorithmen um einige Größenordnungen höher, jedoch sollte sich durch das Parallelisieren und Optimieren einiger kritischer Stellen die Laufzeit ohne großen Aufwand verringern lassen. Um die Qualität der gefundenen Knotenseparatoren noch besser zu beurteilen, haben wir einen Algorithmus zur Berechnung von Knotenordnungen implementiert, der auf dem *nested dissection* Algorithmus basiert [12, 13, 25] und unsere Knotenseparatoren verwendet, um Knotenordnungen hoher Qualität zu berechnen.

Contents

1. Introduction **5**
1.1. Overview . . . 5

2. Preliminaries **7**
2.1. Graph Notation . . . 7
2.2. Graph Properties . . . 7
2.3. Graph Partitioning . . . 8
2.4. Flow Network . . . 9
2.5. Node Separator Problem . . . 10

3. Related Work **11**
3.1. Nested Dissection . . . 13

4. Finding Small Node Separators **17**
4.1. Overview . . . 17
4.2. Defining the Subgraph . . . 19
4.3. Flow Problem . . . 20
4.4. Balancing . . . 23

5. KaHIP Nested Dissection **25**
5.1. Minimum Degree Heuristic . . . 25
5.2. Node Separator Balancing . . . 27
5.3. K-way Version . . . 28
5.4. Connected Components Version . . . 28

6. Experiments **31**
6.1. Test Environment . . . 31
6.2. Test Instances . . . 31
6.3. Parameters of our Algorithms . . . 31
6.4. Competitors . . . 37
6.5. Separator Quality . . . 37
6.6. Nested Dissection . . . 41

7. Conclusions **49**
7.1. Future work . . . 49

A. Appendix **51**
A.1. Detailed Results of our Nested Dissection Experiments . . . 51
A.2. Parameters of our Algorithms . . . 56

1. Introduction

The divide-and-conquer strategy is a well-known pattern in algorithm engineering which is also useful for some popular graph problems [27, 24, 2]. The idea is to divide the original problem into two or more smaller problems which are solved by applying this method recursively. The solutions to the subproblems are combined to result in a solution for the original problem [26]. The reasons to use divide-and-conquer are manifold and include improved algorithm efficiency as well as easy parallelization.

Applying a divide-and-conquer strategy only works efficiently if the subproblems have roughly the same size, are independent of each other and the costs of solving the original problem given the solution of the subproblems are small.

Consider a graph $G = (V, E)$ with $|V| = n$ and we want to apply a divide-and-conquer strategy on the nodes of G. According to the last paragraph, we have to divide G into two or more subgraphs of roughly the same size. Additionally, the subgraphs must be independent of each other, i.e. they must not be connected to each other. To find such subgraphs we search for a set $S \subset V$ such that $G \setminus S$ is a set of two or more connected components which are all smaller than αn for a fixed $\alpha \in (\frac{1}{2}, 1)$. Such a set is called *node separator* or α-separator. A small α guarantees subgraphs of nearly equal sizes. To meet the requirement of solving the original problem with little costs given the solutions on the subgraphs, we want to minimize the size of S. However, finding a node separator which meets the above conditions on general graphs is NP-hard even if the maximum node degree is three [4, 10]. This is why heuristics and approximation algorithms are used on general graphs to find small node separators.

One popular strategy of computing node separators is to first compute an edge separator which is then turned into a node separator [33, 42]. The edge separator can be obtained by applying a graph partitioning heuristic. We use this concept in our new algorithm and introduce a novel technique to transform the edge separator into a node separator. To improve the balance of the obtained separator we apply an additional balancing algorithm which maintains the cardinality of the node separator. Our algorithm is integrated in the open source graph partitioning package KaHIP which is one of the best graph partitioners in terms of solution quality at the moment [42].

1.1. Overview

We start by introducing some basic graph notations and properties as well as some graph problems related to the node separator problem in Chapter 2. We finish the second chapter with the definition of the node separator problem. Chapter 3 first presents related work on node separators before introducing *nested dissection* – a divide-and-conquer algorithm introduced by Alan George in 1973 [12]. Additionally we describe a popular algorithm called *minimum degree heuristic* [43] which is used on the finest level of recursion during nested dissection. In Chapter 4 we describe our own algorithm for finding small node separators in detail. We start by giving an overview and then describe the key components of the algorithm. Chapter 5 describes our nested dissection implementation using our new node separator algorithm. Experimental results on several test instances and a comparison with two other popular algorithms

for nested dissection can be found in Chapter 6. In Chapter 7 we give our conclusion and an outlook on future work on the topic in general and our algorithm in particular.

2. Preliminaries

In this chapter we first want to introduce some basic graph notations and properties as well as the graph partitioning problem. We continue by introducing the notation of flow networks and related problems. In the last part of this chapter we define the node separator problem.

2.1. Graph Notation

A weighted graph G can be described as a 4-tuple $G = (V, E, c, \omega)$ where V is a set of nodes and E is a set of edges. An edge is a tuple (u, v) of two nodes from V and models a relationship between them. Each edge is weighted by a cost function $\omega : E \to \mathbb{R}_{>0}$ and another cost function $c : V \to \mathbb{R}_{>0}$ assigns a weight to each node of the graph. The nodes in the set $\Gamma(u) = \{v \in V \mid \{u, v\} \in E\}$ are called neighbors of u and the degree of u is the cardinality of this set.

An unweighted graph can be described by omitting c and ω, i.e. the definition becomes $G = (V, E)$. A graph is called *undirected* if for each edge $(u, v) \in E$ there exists an edge $(v, u) \in E$ and both edges are weighted equally. The cost functions c and ω can be extended to sets such that $c(V') \coloneqq \sum_{v \in V'} c(v)$ for all $V' \subseteq V$ and $\omega(E') \coloneqq \sum_{e \in E'} \omega(e)$ for all $E' \subseteq E$.

A *subgraph* $G_S = (V_S, E_S)$ of a graph G is a graph where the nodes are a subset of V and the edges a subset of E. We call the subgraph G_S *induced* if the set E_S contains every edge of E on the node set $V_S \subset V$. A set of nodes $C \subset V$ is called *clique* if there is an edge $\{u, v\} \in E$ for every pair of nodes in C. A sequence of nodes $(u, \dots, v)$ is called *u-v path* if each consecutive node pair is connected by an edge. A path where start and end node are equal is called *cycle*.

In this work we omit the cost functions for brevity and $G = (V, E)$ denotes a weighted graph as defined in the last paragraph where the node and edge weights are all set to one if not stated otherwise.

2.2. Graph Properties

A graph is *simple*, if there are no self-loops (u, u) in E and furthermore no multiple edges from a node u to node v exist. An *acyclic* graph contains no cycles. We call a graph *bipartite* if we can split the node set V into two non-empty and disjoint sets U and W such that the nodes within each set are not connected to each other, i.e. if $(u, w) \in E$ then $u \in U$ and $w \in W$ or vice versa.

In case of a directed graph we say the graph is *strongly connected* if there exists a u-v path and a v-u path for every pair of nodes. Many graphs are not strongly connected itself in which case we search for subgraphs which are strongly connected. The maximal found subgraphs in terms of node cardinality are called *strongly connected components*. The terms also apply to undirected graphs but we use the term *connected components* instead of strongly connected components. A linear order $(V, \prec)$ on the nodes of a directed, acyclic graph such that each edge $(u, v) \in E$ implies $u \prec v$ is called a *topological order*.

A set $M \subset E$ is called *matching* if the edges in M do not share any common nodes. The weight of a matching is defined as the weight of all its edges. A *maximum weight matching* (short: maximum matching) is a matching which has maximum weight among all possible matchings in G. A set $X \subset V$ is called *vertex cover* if for each edge $\{u, v\} \in E$ either u or v are part of X. The weight of a vertex cover is defined as the weight of all its nodes. A *minimum vertex cover* is a vertex cover which has minimum weight among all possible vertex covers in G.

2.3. Graph Partitioning

Since our algorithm is based on edge separators, we want to introduce the graph partitioning problem in this section. An edge separator is a set of edges which connect the partitions obtained by partitioning the graph.

Let $G = (V, E, c, \omega)$ be a graph, $n = |V|$ and $m = |E|$.

Definition 2.1. *k-way Partition*
A k-way partition of a graph $G = (V, E, c, \omega)$ is a set $\{V_1, \ldots, V_k\}$ with the properties $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$.

A 2-way partition is also called *bisection*.

Definition 2.2. *Edge cut*
We define the edge cut between the partitions of a k-way partition as $cut := \sum_{i<j} |E_{ij}|$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i \wedge v \in V_j\}$.

In many applications it is necessary that the partitions are *balanced*, i.e. they have equal sizes. The following definition formalizes this constraint.

Definition 2.3. *Balancing Constraint*
The property $\forall i \in \{1, \ldots, k\} : V_i \leq L_{max} := (1 + \epsilon)\lceil |V|/k \rceil$ is called balancing constraint.

With the above definitions in mind we now state the graph partitioning problem as follows:

Definition 2.4. *Balanced Graph Partitioning Problem*
Consider a graph $G = (V, E, c, \omega)$. Let k be the desired number of partitions and ϵ the maximum allowed imbalance between them. Then the balanced graph partitioning problem asks to find a k-way partition of G such that the edge cut is minimized and the balancing constraint is fulfilled for the given ϵ.

The balanced graph partitioning problem is NP-hard which was shown by Hayfil and Rivest [18] as well as Garey et al. [11] and during the last decades of research, many different approaches evolved to solve partitioning problems heuristically.

2.4. Flow Network

Consider a simple, directed graph $G = (V, E)$ with a cost function $\omega : V \times V \to \mathbb{R}$ denoting capacities on the edges. The nodes $s \in V$ and $t \in V$ are two designated notes in G and are also called source and sink. The tuple (G, s, t, ω) is called *flow network*. A *flow* is a function $f : E \to \mathbb{R}_0^+$ which satisfies the *capacity constraint*, the *flow conservation constraint* and the *skew symmetry constraint*. The capacity constraint demands that for each edge $(u, v) \in E$ the inequation $0 \leq f(u, v) \leq \omega(u, v)$ holds. To fulfill the flow conservation constraint each node has to emit the same amount of flow as it receives with the exception of nodes s and t. The skew symmetry constraint requires that $f(u, v) = -f(v, u)\ \forall (u, v) \in V \times V$. The value $val(f)$ of a flow f is defined as $val(f) := \sum_{(s,v)\in E} f(s, v) - \sum_{(v,s)\in E} f(v, s)$ which is equal to the amount of flow routed from source to sink.

Residual Graph & Cuts

The *residual capacity* is defined as $r_f = \omega(u, v) - f(u, v)$. The *residual graph* $G_r = (V, E_r)$ of G is given by $E_r = \{(u, v) \in V \times V \mid r_f(u, v) > 0 \text{ and } (u, v) \in E \text{ or } (v, u) \in E\}$. Let S be a subset of V then the partitioning $(S, V \setminus S)$ is called a cut in G. If $s \in S$ and $t \in V \setminus S$ then S is called s-t cut. The *capacity* of a cut is defined as

$$\omega(S, V \setminus S) = \sum_{\substack{(u,v)\in V\times V \\ u\in S, v\in V\setminus S}} \omega(u, v).$$

A minimum s-t cut is a cut $(S, V \setminus S)$ such that $\omega(S, V \setminus S)$ is the minimum among all s-t cuts.

Max-Flow Problem

Given a flow network (G, s, t, ω) the max-flow problem demands to transfer as much flow as possible from source s to sink t. This means we want to maximize the value $val(f)$ of a flow f in the network.

Max-Flow Min-Cut Theorem

Theorem 2.5. *The maximum value of an (s, t)-flow in a flow network (G, s, t, ω) is equal to the minimum capacity among all s-t cuts.*

This theorem was shown by Ford and Fulkerson as well as Elias, Feinstein and Shannon [8, 6]. This allows us to easily compute a minimum s-t cut given a maximum flow in the network.

2.5. Node Separator Problem

Let $G = (V, E)$ be an undirected graph.

Definition 2.6. *Node Separator*
$S \subset V$ is called a node separator if $V \setminus S = A \cup B$ and $A \cap B = \emptyset$. The two sets A and B do not need to be connected components.

This means removing the node separator S from G partitions the graph into at least two disjoint sets A and B. The definition can be extended to a *k-way node separator* which demands that $V \setminus S = V_1 \cup \cdots \cup V_k$ and $V_i \cap V_j = \emptyset$ for every $i \neq j$. Now we can state the node separator problem as follows:

Definition 2.7. *Node Separator Problem*
Let $\epsilon \in [0, 1)$ be a constant. The problem asks to find a node separator S such that

(i) $G \setminus S = V_1 \cup \cdots \cup V_k$

(ii) $|V_i| \leq (1 + \epsilon)\lceil \frac{|V|}{k} \rceil$ *for every* $i \in \{1, \ldots, k\}$

(iii) $|S|$ *is minimal*

Property (ii) is denoted as balancing constraint of the node separator problem. This is sometimes also written as $|V_i| \leq \alpha|V|$ for a given alpha in $(0, 1)$ which corresponds to our notation for $\alpha = \frac{1+\epsilon}{k}$. It was shown that the problem is NP-hard on general graphs even in case the maximum node degree is three [4, 10].

Node separators are useful in the context of divide-and-conquer algorithms but also in other areas, e.g. on a communication network they tend to be the bottle neck [7]. This observation can be used to provide lower bounds on communication tasks in that network [2, 24].

Quality of Node Separators

There are several measures to assess the quality of node separators. First of all there is the size of the node separator which is what we want to minimize. Regarding the balancing constraint we can differ between the balance of each partition compared to the complete graph and the balance among partitions excluding the node separator. Given a node separator, we can compute the actual ϵ in the notation of Definition 2.7 by

$$\epsilon = max(|V_1|, \ldots, |V_k|) \cdot (\lceil \frac{|V|}{k} \rceil)^{-1} - 1. \tag{2.1}$$

Note that ϵ can be smaller than 0, because the separator partition S is part of V. To assess the balance among the induced partitions after removing the separator we define the balance β of the partitions as follows:

$$\beta = max(|V_1|, \ldots, |V_k|) \cdot (\lceil \frac{|V| - |S|}{k} \rceil)^{-1} - 1 \tag{2.2}$$

where S is the set of separator nodes.

3. Related Work

While the problem of finding the minimal node separator is NP-hard on general graphs there exist several graph classes on which the problem is solvable in polynomial time at least if the balancing constraint is relaxed [23, 26]. The most famous graph class concerning the node separator problem is certainly the class of planar graphs.

Planar Separator Theorem

A *planar graph* is a graph $G = (V, E)$ which can be embedded in the plane, i.e. it can be drawn on a plane such that no pair of edges are crossing each other. Lipton and Tarjan proofed the *Planar Separator Theorem* which states that there always exists a separator S with $|S| \in O(\sqrt{|V|})$ which partitions the planar graph into two sets A and B, each of maximum size $2|V|/3$ and this separator can be obtained in linear time [26]. However, for the perfectly balanced case it was shown that even on planar graphs the node separator problem is NP-hard [9].

Heuristics

For general graphs there exist several heuristics for computing small node separators. We want to emphasize two tools which we used as competitors in our own experiments.

The SCOTCH graph partitioning package [33] computes a node separator by first calculating an edge separator by means of their graph partitioning algorithm and then transforming the edge separator into a node separator [33, 34, 35]. To accomplish that, they use a method first described by Pothen and Fan [37] which calculates a maximum matching on the bipartite graph induced by the edge separator with the Hopcroft-Karp algorithm [17]. Then a minimum vertex cover - the node separator - can be computed from the maximum matching since the problem of finding a maximum matching and finding a minimum vertex cover are equivalent on bipartite graphs (see König's Theorem [22]).

Before we started this work, KaHIP already contained an algorithm to compute node separators on undirected graphs which uses the same approach as SCOTCH. In Section 6.5 we do a comparison between this approach and our new algorithm. KaHIP makes use of the duality between a maximum matching and a maximum flow in a bipartite graph to compute a node separator. Given the maximum flow in the constructed flow problem depicted in Figure 1, this induces an s-t cut $(S, V \setminus S)$ on the bipartite graph which can be turned into a minimum vertex cover $C = (B[V_1] \setminus S) \cup (B[V_2] \cap S)$ where $B[V_1]$ and $B[V_2]$ denote the two sets of nodes forming the bipartite graph [42].

The METIS graph partitioner also uses the idea to obtain a node separator from an edge separator, however, it uses a multilevel approach to compute the node separator [20, 19]. This means, METIS coarsens the graph in several steps. This is done by carefully merging nodes so that the overall structure of the graph is maintained in the coarser levels. On the coarsest level a node separator is computed and during

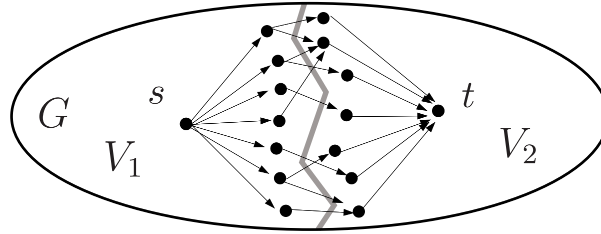

Figure 1: Flow problem constructed by KaHIP to find a minimum vertex cover in the bipartite graph induced by the edge separator [42].

the uncoarsening phase (to get back to the original graph) the separator is projected on the original graph and refined by dropping of nodes which are not needed. This process is depicted in Figure 2 (Karypis et al. [19]).

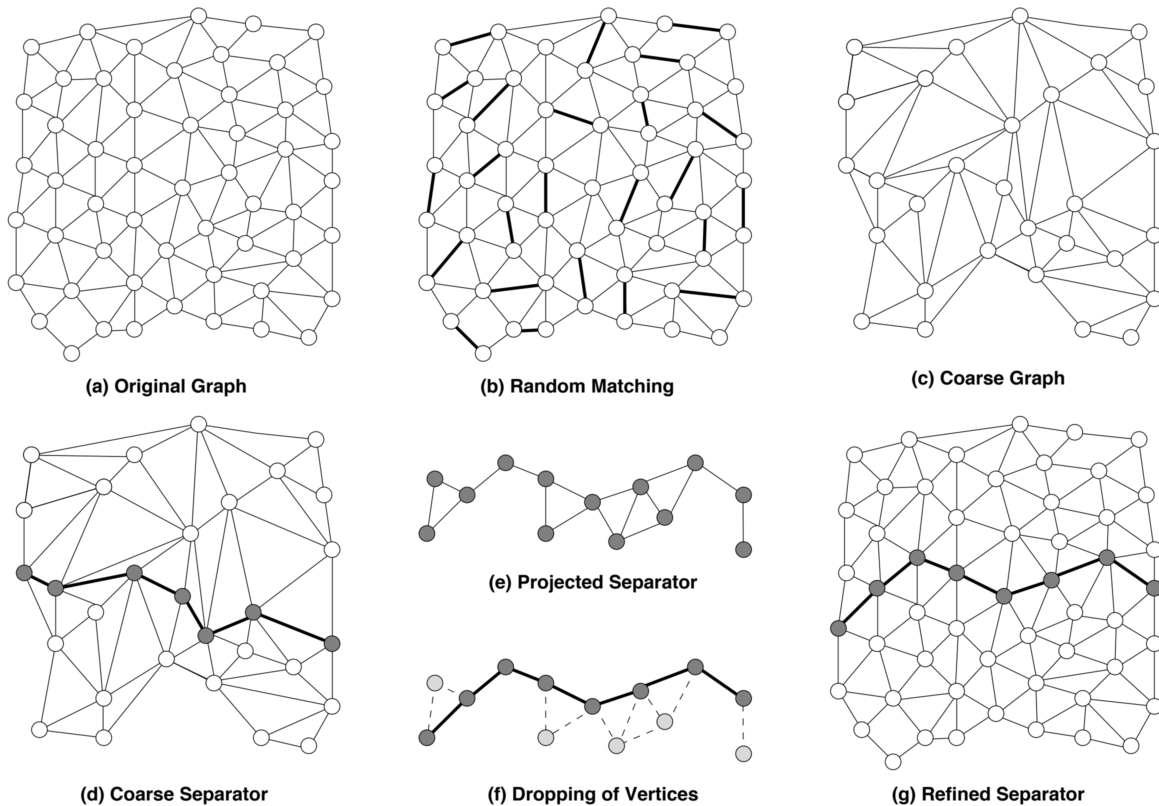

Figure 2: Multilevel approach to obtain a node separator by METIS [19]

3.1. Nested Dissection

One of the most well-known applications of node separators can be found in linear algebra or more precisely the factorization of sparse, symmetric matrices where the matrix is decomposed into a product of matrices. To reduce the number of arithmetic operations of the factorization algorithm and the memory requirements of the actual factorization itself we want to minimize the *fill-in*, i.e. the number of entries of the matrix that are initially zero and change to a non-zero value during the execution of the algorithm. To obtain a fill-in reduced version of the matrix we can exchange the rows and columns and apply the factorization on this altered (permuted) version of the original matrix.

Consider a sparse, symmetric matrix $A \in \mathbb{R}^{n \times n}$, a vector $b \in \mathbb{R}^n$ and a linear equation system

$$Ax = b.$$

From numerical linear algebra it is known that this can be solved efficiently by factorizing A and computing the solution of the above equation system with the factorized version of A. Depending on the structure of A, there exist several factorization methods like LU decomposition or Cholesky factorization. Let P be an $n \times n$ matrix which permutes the rows and columns of A. Instead of factorizing A we can factor the matrix PAP^T which significantly decreases the amount of fill-in during the factorization [14].

For example, A can be factored by LU decomposition by iteratively eliminating the matrix elements below the main diagonal of the i-th column. During this elimination, it is possible that some entries which were zero are turned to non-zero values. As a consequence, this increases the overall number of entries as well as the number of arithmetic operations needed to factorize A. An example of such an elimination is given in Figure 3.

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix} \xrightarrow{\textit{eliminating 1st column}} \begin{pmatrix} 1 & 1 & 2 \\ 0 & -1 & -2 \\ 0 & -2 & -4 \end{pmatrix}$$

Figure 3: Elimination of first column where non-zeros are introduced

The problem of finding a permutation which minimizes the fill-in is NP-hard which was shown by Yannakakis [45]. Alan George developed an algorithm called *nested dissection* [12] which reduces fill-in on regular finite element meshes. Later Leiserson and Lewis [25] as well as George et al. [13] generalized the algorithm for general graphs by using node separators.

The *nested dissection* algorithm makes use of an algorithm called *minimum degree heuristic* which itself is a heuristic to reduce the fill-in. The following paragraph describes this algorithm.

Minimum Degree Heuristic

The minimum degree algorithm [43] is a heuristic to reduce the fill-in generated by matrix factorization. Before we introduce the algorithm we first have to describe how a symmetric matrix can be represented as an undirected graph.

Let $A = (a_{ij})$ be an $n \times n$ symmetric matrix. We can represent this matrix as an undirected graph G by introducing a node i for every row in A. For every $a_{ij} \neq 0$ we introduce an undirected edge $\{i, j\}$.

Now let $G = (V, E)$ be a representation of a sparse, symmetric matrix A. The ith elimination step described in the previous section corresponds to the elimination of the edges from node i to its neighbors and the extension of the neighborhood of i to a clique. The edges which were missing to form a clique represent the generated fill-in.

The algorithm is based on the idea that a node $u \in V$ which has minimum degree among all nodes produces only a small amount of fill-in as u has a small neighborhood. After the elimination we obtain a graph G_u where u and all its edges are removed and its neighbors form a clique. Figure 4 depicts this step. Algorithm 1 shows the minimum degree heuristic in pseudocode.

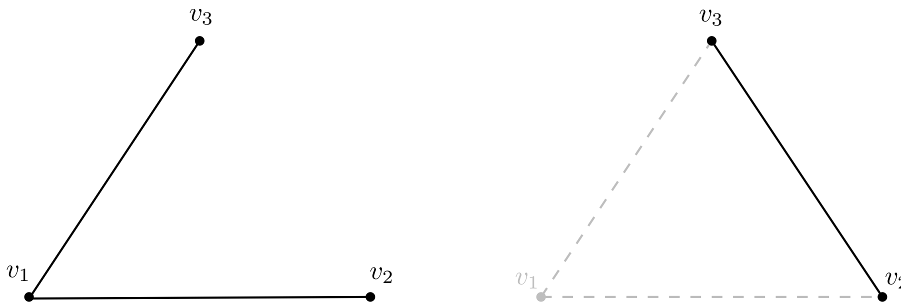

Figure 4: Graph representation of matrix A in Figure 3. Nodes v_1, v_2 and v_3 correspond to rows $1, 2$ and 3 of A. The right image shows the graph after eliminating v_1.

Algorithm 1: Minimum Degree Algorithm

`Input:` $G = (V, E)$: graph representation of symmetric matrix
`Output:` π : node ordering

1. `while` $V \neq \emptyset$ `do`
2. $\quad u \leftarrow$ **MinDegree**(G) *// Select node u of minimum degree in G*
3. $\quad \pi \leftarrow (\pi, u)$ *// append u to the already ordered sequence* π
4. $\quad G \leftarrow G_u$ *// eliminate u and update the graph*
5. `return` π

A priority queue can be used to obtain the node of minimum degree of all remaining nodes efficiently.

Nested Dissection Algorithm

Algorithm 2 shows the nested dissection approach which applies a divide-and-conquer strategy to compute a fill-in reduced ordering of a sparse, symmetric matrix. The node ordering can be transformed into a permutation matrix without any further steps. It recursively calculates a 2-way node separator such that the graph is partitioned into three sets A, B and the separator set S. S is always ordered after the partitions A and B have been ordered. On the coarsest level the minimum degree heuristic is applied to the subgraphs induced by the partitions.

Algorithm 2: Nested Dissection

```
  Input: G = (V, E) : Graph, γ : minimum degree threshold
  Output: π : node ordering
1 if |V| ≤ γ then
2 |  MinimumDegree(G, π)
3 else
  |  // compute 2-partition of G
4 |  P = {G1 = (V1, E1), G2 = (V2, E2)} ← PartitionGraph(G)
  |
  |  // find node separator on G given the partitioning P
5 |  (V1', V2', S) ← ComputeSeparator(G, P)
  |
  |  // continue recursively on G1 and G2
6 |  NestedDissection(G1')
7 |  NestedDissection(G2')
8 |_ OrderSeparator(S, π)                    // order S after G1 and G2
```

The threshold γ is used to control when the recursion should end and the minimum degree ordering should be applied. Typically, a value between 30 and 200 nodes is chosen for γ. As experiments of Pellegrini et al. [34] have shown, the nested dissection algorithm performs better than the minimum degree heuristic on the complete graph. Also, the solution quality is often better if γ is set to a lower value (see Section 6.3). Experiments suggest that small node separators usually tend to improve the quality of the obtained node ordering [21, 32].

Quality of Node Orderings

The quality of a node ordering can be measured by several criteria. The first one is the number of non-zeros (NNZ) of the factorized matrix which have to be stored. The second one is the operation count (OPC) which is equal to the number of arithmetic operations which are required to compute a factorization of the matrix. A third one is the shape of the elimination tree. Parallel factorization should be all the better if the elimination tree is balanced. Depending on the application, the importance of those quality measures will be ranked differently.

Other Implementations

The graph partitioning tools SCOTCH [33] and METIS [21] both implement versions of nested dissection. METIS implements the basic nested dissection algorithm with some improvements to the minimum degree heuristic. The node separators are computed in a multilevel approach (see Section 3) [21, 20].

SCOTCH provides several ordering methods for the nested dissection algorithm which can be combined with boolean algebra [31]. Node separators are calculated with the approach described in Section 3.

4. Finding Small Node Separators

This chapter presents our new algorithm for finding small separators on large, undirected graphs with a guaranteed balancing constraint. We first give a general overview of the algorithm including some high level pseudocode and then describe the key components in more detail in the following sections.

4.1. Overview

The goal of our algorithm is to find small node separators which fulfill a user defined balancing constraint. There exists a natural trade-off between the size of the node separator and the resulting balance between the induced connected components which we cannot overcome. But our algorithm guarantees to hold a maximum allowed imbalance which can be set as a parameter. An overview of the available parameters is given in Appendix A.2.

There are several ways to compute a node separator on a graph, some of the more popular ones are described in Chapter 3. We make use of the strategy which first computes an edge separator and then transforms that separator into a node separator.

By partitioning a graph $G = (V, E)$ into a k-way partition $V_1, \dots, V_k$, the cut edges $E_{cut} := \{\{u, v\} \in E : u \in V_i \wedge v \in V_j, i \neq j, i, j \in \{1, \dots, k\}\}$ induce an edge separator on graph G. Reducing the amount of cut edges can also reduce the number of involved cut nodes $\{u \in V : \exists v \in V : \{u, v\} \in E_{cut}\}$ as the edge separator is an upper bound for the cut nodes. Since the minimum vertex cover on the set of cut nodes is an upper bound for the computed node separator of our algorithm we are interested in finding a small set of cut nodes and therefore a small set of cut edges.

This implies that the quality of the k-way partition in terms of total edge cut is a crucial component of our node separator algorithm. The graph partitionings computed by the graph partitioning package KaHIP [16, 38] have one of the highest qualities among current partitioning tools which is the reason why we use KaHIP for obtaining small edge separators.

Given an edge separator, the standard approach is to calculate a node separator from the set of involved cut nodes. The idea of our approach is to extend the search area between each pair of partitions for finding a smaller node separator. The area must be carefully selected so that the balancing guarantee still holds. A detailed description on how this is done can be found in Section 4.2.

Once we have extended our search area, that is, we selected a proper subgraph representing the search area, we still have to find a node separator. We construct a special maximum flow problem on the subgraph which is solved by our implementation of the push-relabel algorithm introduced by Goldberg and Tarjan [15]. A detailed description of the flow problem can be found in Section 4.3. The result of this step is a node separator which is part of the defined subgraph meaning that it still holds the balancing constraint, assuming the subgraph is selected properly. As the set of cut nodes are a subset of the subgraph nodes and we can guarantee that we find the minimum node separator on this subgraph, the upper bound of the node separator size is indeed equal to a minimum vertex cover on the set of cut nodes.

An important feature which sets our algorithm apart from other implementations is the ability to compute a k-way node separator without the need for recursion. That is mainly due to KaHIP which enables us to compute a k-way partitioning without recursive bipartitioning. To obtain a k-way node separator we compute a 2-way separator between every pair of partitions. The union of these separators is a k-way node separator. Since the direct computation of a k-way partitioning is usually of higher quality than a recursive bisectioning approach this can further improve our node separators (see Section 6.3).

Algorithm 3 shows the complete pseudocode of our new node separator approach and also summarizes the steps we take to find small node separators.

Algorithm 3: Separator Algorithm

```
  Input: G = (V, E) : Graph, P = {V_1, ..., V_k} : Partition
  Output: S : Separator
1 foreach (V_i, V_j) ∈ P, i ≠ j do
2   H ← CreateSubgraphForFlowProblem()          // defines the search area
3   H' ← CreateFlowProblem(H)
4   MaxFlowMinCut(H')                            // solve flow problem

    // try to increase the balance of the minimum cut in H'
5   T ← IncreaseBalance(H')
6   S ← S ∪ BoundaryNodes(T)                     // Add boundary nodes of T to S
7 return S
```

We already discussed lines two to four and further information on each of these steps can be found in the subsequent sections of this chapter. On line five we apply a balance increasing method to the solved flow network. The reason for this is a rather technical one concerning the flow problem. Consider a maximum flow on a flow network. The source set is usually computed by doing a breadth-first search from the source node s which stops at each edge which is saturated. This way, we obtain a minimum cut in the network which is probably quite near the source node s. Since we are interested in balanced node separators we would like to have a more balanced minimum cut which is why we apply this balancing step to obtain a more balanced source set T. Further details about this method can be found in Section 4.4.

Line 6 of Algorithm 3 finally adds the obtained node separator between the current pair of partitions to the overall node separator. The *BoundaryNodes* function returns the set $\{u \in T : \exists \{u, v\} \in E_{H'} : v \in H' \setminus T\}$ which is exactly the set of separator nodes. This step also includes the adaptation of partitions V_i and V_j according to the found node separator. The separator nodes are assigned to a special separator partition V_{k+1}.

After node separators between all pairs of partitions have been calculated we can return the overall separator at line seven. Since the graph partitioning algorithm in KaHIP contains randomized parts [30, 39] and our algorithm depends on the edge separator induced by a partitioning of KaHIP we can do multiple runs of Algorithm 3

with different partitionings and return the best separator found for another increase in quality (see Section 6.3).

In the following sections we assume a 2-way node separator for simplicity but it is obvious that the definitions and arguments also apply to a k-way node separator which can be obtained by calculating pair-wise 2-way node separators on the k partitions and combine them to an overall node separator of the complete graph.

4.2. Defining the Subgraph

Assume that we want to find a 2-way node separator S of a graph $G = (V, E)$ and the induced partitions should fulfill the balancing constraint $|V_i| \leq (1+\epsilon)\lceil |V|/2 \rceil$ with $i = 1, 2$ and $0 < \epsilon < 1$. Our algorithm extends the search area for finding a small node separator in G.

We start with identifying the set of boundary nodes B_1 and B_2 for partition V_1 and V_2 which are defined as $B_1 := \{u \in V_1 \mid \exists \{u, v\} \in E : v \in V_2\}$ and $B_2 := \{u \in V_2 \mid \exists \{u, v\} \in E : v \in V_1\}$. From these boundary sets we perform breadth-first searches into the respective partitions to determine sets V_1^* and V_2^* which together define our new search area. Since the found node separator S induces a new partitioning of G into $G = V_1' \cup V_2' \cup S$ the balance of V_1' and V_2' is also altered. Therefore we have to choose the search area carefully because we still want to guarantee the balancing constraint for V_1' and V_2'.

Figure 5 shows G partitioned into V_1 and V_2. The gray line represents the edge cut between the two partitions. The blue lines mark the boundaries of the subsets $V_1^* \subset V_1$ and $V_2^* \subset V_2$ which denote the search area used by our algorithm.

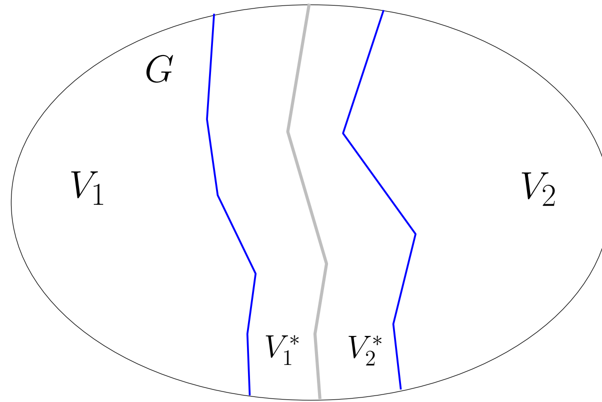

Figure 5: Extended search area $V_1^* \cup V_2^*$ in graph G

To meet the balancing constraint we define the size of V_1^* as $|V_1^*| \leq (1+\epsilon)\lceil \frac{|V|}{2} \rceil - |V_2|$ and the size of V_2^* as $|V_2^*| \leq (1+\epsilon)\lceil \frac{|V|}{2} \rceil - |V_1|$.

This way the maximum cardinality of V_1' is

$$|V_1'| \leq |V_1| + |V_2^*| \leq |V_1| + (1+\epsilon)\lceil\frac{|V|}{2}\rceil - |V_1| = (1+\epsilon)\lceil\frac{|V|}{2}\rceil \tag{4.1}$$

which proofs that the balancing constraint is still guaranteed if we choose V_1^* and V_2^* like that. Equation 4.1 also analogously applies for V_2'.

To incorporate these size constraints in our algorithm the breadth-first searches from the boundaries B_1 and B_2 stop if the resulting subgraph V_1^* or V_2^* exceed the defined sizes. It is possible that V_1^* or V_2^* is empty. This happens if the initial imbalance obtained from the partitioning algorithm is close to the maximum allowed imbalance or even larger. In this case we still have the set of boundary nodes $B_1 \subseteq V_1^*$ or $B_2 \subseteq V_2^*$ which are always part of the search area.

Once we have determined the sets V_1^* and V_2^* we can define the subgraph H which is induced by the set of nodes from $V_1^* \cup V_2^*$.

4.3. Flow Problem

Given the subgraph $H = (V_H, E_H)$ which is built according to the description in Section 4.2 we want to find a minimum node separator $S \subset V_H$ on H. The size of a node separator S is defined as $\sum_{u \in S} c(u)$. In this work, we only use graphs with $c(u) = 1$ for each node u and therefore the size is equal to the number of nodes in S, however, the concept also works for graphs with other node weights.

The idea is to use flow techniques to find S. Consider a flow network with designated source and sink node s and t. A flow on that network implies an s-t cut partitioning the network into a source and sink partition. This way we have two sets of boundary nodes which are the end nodes of the cut edges. It is obvious that both sets themselves are node separators and we could simply use the smaller set to get a node separator. A better approach used by KaHIP before we introduced our new algorithm is to compute a minimum vertex cover on the bipartite graph induced by the two boundary node sets [42]. According to the max-flow min-cut theorem (see Section 2.4), the maximum flow on the network induces a minimum s-t cut, i.e. the capacity of the cut is the minimum among all s-t cuts. However, this approach minimizes the summed capacities of the cut edges and not the summed node weights and therefore neither of the above approaches is able to always find a minimum node separator on the network. We would like to have a flow network such that a maximum flow on that network induces a minimum set of nodes which separates source and sink. Therefore the flow limiting capacities have to be the node weights instead of edge weights such that each node can only carry as much flow as its weight.

To model this flow network on nodes we construct a normal flow network $H' = (V_H', E_H')$ from H on which we compute a maximum flow with our implementation of the push-relabel algorithm introduced by Goldberg and Tarjan [15]. The minimum cut in H' induces a minimum node separator S in H as described above. The following paragraph describes the construction of H'.

For each node $u \in V_H$ we introduce two nodes u_1 and u_2 in V_H' which are connected through a directed edge $(u_1, u_2) \in E_H'$ with an edge capacity of one - the node weight.

We denote this edge as duplicate edge of node u. Each undirected edge $\{u, v\} \in E_H$ can also be seen as two directed edges (u, v) and (v, u). In H' we separate the incoming edges from the outgoing edges of u (see Figure 6) and introduce the directed edges (u_2, v_1) and (v_2, u_1) in E'_H for each neighbor v of u. Each edge gets a capacity of ∞. Figure 7 illustrates this construction.

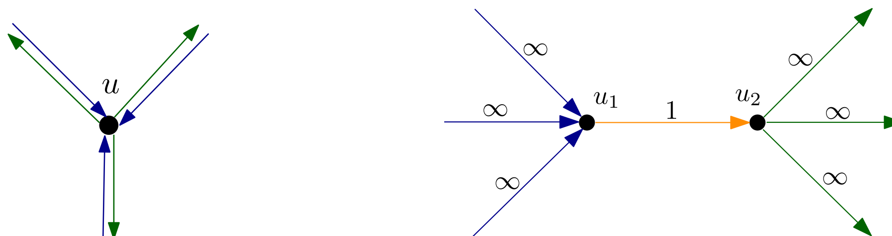

Figure 6: Node u with undirected edges interpreted as directed ones on the left and representation of u as u_1 and u_2 in H' on the right.

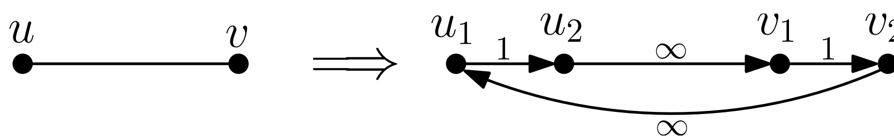

Figure 7: Edge construction for each neighbor v of u

For a valid flow network we need to specify a source and sink node. Therefore we add two additional nodes to V'_H which model source and sink. To connect those nodes to the rest of the network we determine the set of boundary nodes between H and the partitions V_1 and V_2 of the original graph G which we want to denote by A_1 and A_2 respectively. Figure 8 shows the boundary node sets A_1 and A_2.

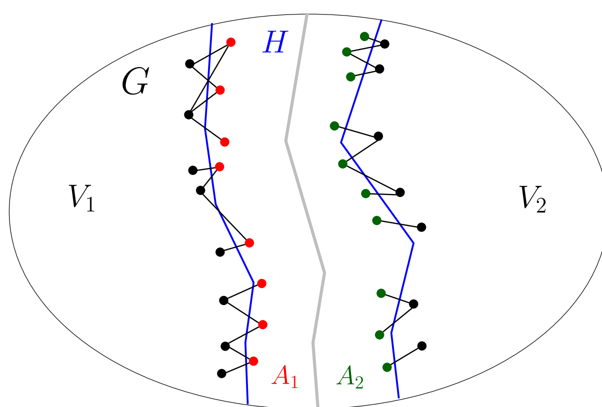

Figure 8: Boundaries A_1 and A_2 of subgraph H

According to the construction, a node $u \in A_1$ corresponds to u_1 and u_2 in the flow network H'. We connect the source s to u_1 for each node $u \in A_1$ with a directed edge of capacity ∞. Analogously a node $v \in A_2$ corresponds to v_1 and v_2 in H'. Connecting v_2 to t with a directed edge of capacity ∞ for each node $v \in A_2$ finishes the construction of the flow network.

Solving the Flow Problem

Lemma 4.1. *The minimum s-t cut induced by a maximum flow in H' is equivalent to a minimum node separator S in H.*

Proof. Since every edge in H' has an integer capacity, the maximum s-t flow is also integral. As there are at least two edges of capacity one on each s-t path in H' (the duplicate edges of the boundary nodes), the maximum flow on each path is one. Therefore, the cut edges of a minimum s-t cut in H' are a subset of duplicate edges which we denote by E_d and which induce a node separator $S = \{u \in V_H \mid (u_1, u_2) \in E_d\}$ in H with $|S| = |E_d|$. It remains to show that S is the smallest separator among all node separators of H.

Assume that S^* is a node separator in H for which $|S^*| < |S|$ holds. The set of duplicate edges E_d^* of nodes in H' corresponding to the separator nodes of S^* in H are the cut edges of a minimum s-t cut. $|S^*| < |S|$ implies $|E_d^*| < |E_d|$ which means that the s-t cut induced by E_d^* is smaller than the one induced by E_d. Therefore, the s-t cut induced by E_d is no minimum s-t cut which is a contradiction. □

Figure 9 shows an example of the flow network construction and the interpretation of the minimum cut obtained by the max-flow min-cut theorem.

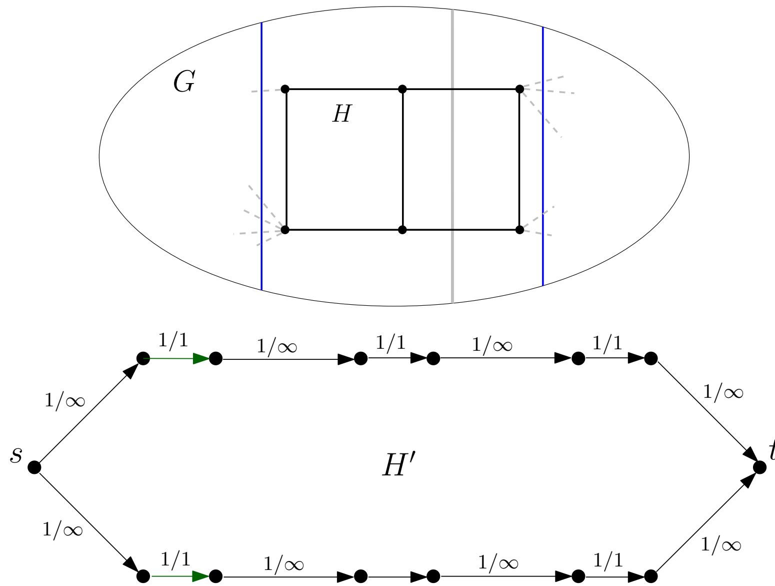

Figure 9: Construction of flow network H' from H (omitting edges with no flow for clarity). Green edges mark the minimum cut.

Boundary Cases

It is possible that A_1 or A_2 is empty. This can happen if the allowed imbalance is high enough so that V_1^* or V_2^* are equal to V_1 or V_2 in the subgraph construction process described in section 4.2. Another possibility is that V_1^* or V_2^* are not connected to $V_1 \setminus V_1^*$ or $V_2 \setminus V_2^*$ respectively. The following paragraph describes how we handle this phenomenon. We only describe the case $A_1 = \emptyset$, the other case is handled analogously. If A_1 is empty we have to differentiate two cases:

Case 1. $V_1^* = V_1$
We set A_1 to the boundary nodes B_1 and if A_2 is also empty we set A_2 to B_2. This way, the imbalance is decreased and we can compute a separator.

Case 2. *V_1^* is not connected to $V_1 \setminus V_1^*$*
We check if we can put V_1^* to the right partition V_2 without violating the balancing constraint. If this is possible we found a partitioning with an empty node separator. Otherwise we construct the flow problem according to case 1.

Another possibility to avoid the boundary cases is to detect them in the course of breadth-first search and stop the search before one of the cases occurs. This approach should result in larger subgraphs than the above one and is therefore an alternative we want to implement in the future.

4.4. Balancing

In the previous section we defined how we construct the flow problem. After we calculated the maximum flow we still have to compute the minimum cut which finally implies our node separator.

Our first approach made use of the property that the minimum cut divides H' into two sets - the source set and the sink set. We calculate the source set by doing a breadth-first search from s including all nodes which are reachable via an unsaturated edge. In a last step we get the corresponding separator nodes by scanning the source set and adding each node to the separator which is connected to the sink set. However, this approach often leads to unbalanced minimum cuts and therefore unbalanced node separators. Figure 9 shows the minimum cut in H' which was computed with this approach. It is obvious that there exists a more balanced minimum cut.

Most Balanced Minimum Cuts

Our second approach uses an algorithm called *most balanced minimum cut* which is based on the knowledge that we have information about all minimum s-t cuts as soon as we have calculated one maximum s-t flow [42]. A set $C \subset V$ of a graph $G = (V, E)$ is called *closed node set* if there are no connections from C to $V \setminus C$, i.e. for every node $u \in C$ an edge $(u, v) \in E$ implies that $v \in C$ as well. Picard and Queyrenne showed that each closed node set containing the source s in the residual graph of a maximum (s, t)-flow induces a minimum s-t cut [36].

The idea of the algorithm is to enumerate all possible minimum cuts of a graph by contracting the strongly connected components of the residual graph and find a better balanced minimum cut concerning the partitions of the original graph. However, it is still NP-hard to find the most balanced minimum cut [3].

The algorithm of Sanders and Schulz [38] applies a random topological order to the strongly connected components and scans them in reverse order. Subsequently adding further strongly connected components yields several closed node sets, each inducing a minimum s-t cut [42]. The closed node set with the best occurred balance among multiple runs of the algorithm with different random topological orders is returned. Figure 10 shows the flow network from Figure 9 after running the most balanced minimum cuts algorithm to improve the balance between the induced partitions.

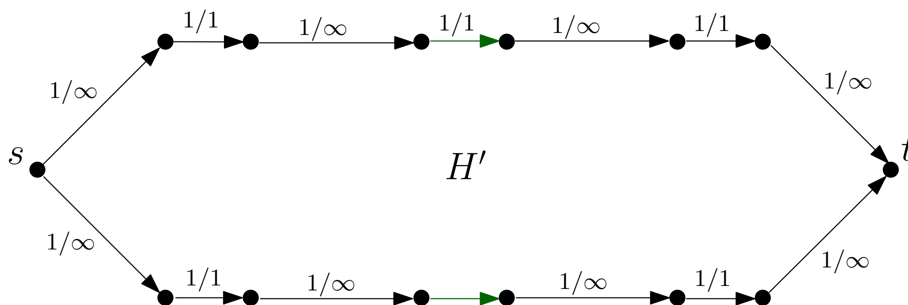

Figure 10: Flow network H' from Figure 9 after applying the balancing algorithm. Green edges mark the minimum cut which is now perfectly balanced in this example.

5. KaHIP Nested Dissection

In this chapter we describe our implementation of the nested dissection algorithm. General information about nested dissection can be found in Section 3.1.

We basically implemented two nested dissection algorithms – a k-way version and a version which searches for connected components in the graph and recursively orders those components instead of calculating a node separator. In Section 5.3 we present the k-way version and in Section 5.4 we describe the differences of the connected components version. But before, we want to describe some changes we have made to the original version of the minimum degree heuristic as it is described in Section 3.1.

5.1. Minimum Degree Heuristic

Since the first version of the minimum degree algorithm was introduced many proposals to improve it were made [1, 14, 28]. We implemented some of the improvements and describe them in this section. However, we did not examine all approaches which would be definitely worth doing so in the future. Algorithm 4 shows our extended minimum degree heuristic.

Algorithm 4: KaHIP Minimum Degree Heuristic

```
   Input: G = (V, E) : Graph, S ⊂ V : Separator, π : ordered node set
   Output: π : node ordering
1  G′ ← IncludeSeparator(G, S)     // add separator nodes to prevent halo nodes
2  M ← {u ∈ V | Γ(u) = 0 or Γ(u) = 1}            // isolated and degree-1 nodes
3  π ← (π, M)
4  while V ≠ ∅ do
5  |  Q ← MinDegree(V)          // Select all nodes of of minimum degree in G
6  |  u ← TieBreaking(Q)          // Select a node u with a tie-breaking strategy
7  |  I ← IndistinguishableNodes(u)
8  |  π ← (π, I)                   // append I to the already ordered sequence π
9  |_ G ← G − I                          // eliminate I and update the graph
10 return π
```

Halo Nodes

As the input graph $G' = (V', E')$ of the minimum degree algorithm is a subgraph representing a partition of the original graph G in the nested dissection algorithm, there exist nodes in V' which lie at the boundary to the node separator and originally were connected to the separator nodes. However, our subgraph G' does not include those edges as it is induced by the set V' which does not contain the node separator. Therefore the degrees of these nodes at the boundary are not the same as in the original graph and as a consequence the minimum degree algorithm will tend to order them first which can lead to more fill-in between separator nodes and the boundary nodes [35].

In literature, those nodes are also called *halo nodes* [35]. Figure 11 illustrates the halo nodes in G' which represents the right partition of the original graph G in this example.

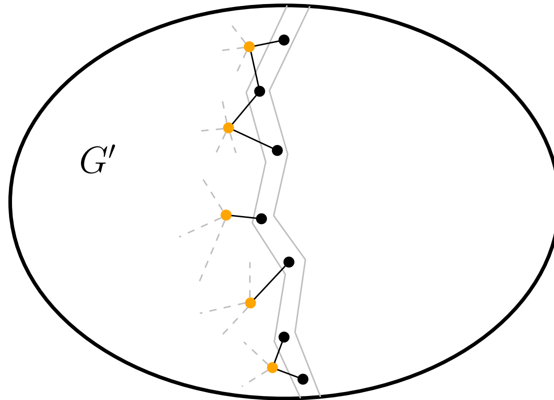

Figure 11: Partition G' on the left with halo nodes marked orange. The black nodes represent the node separator.

Since the original minimum degree heuristic was not designed for nested dissection and therefore G is always the original graph, this problem does only occur in nested dissection. We circumvent this problem by adding the separator nodes to the subgraph G' to form the extended subgraph G^*. Since we do not want to order the separator nodes in this step we exclude them from being ordered by the algorithm. This way, running our minimum degree algorithm works as before but we prevent the wrong ordering of halo nodes.

Isolated Nodes and Degree-1 Nodes

Isolated nodes, i.e. nodes with degree zero, and nodes with degree one can be eliminated without the need to exercise a degree update on the neighbors. Therefore we order all those nodes first and exclude them from the minimum degree process.

Mass Elimination

The idea behind mass elimination is the following. Assume that during the execution of the minimum degree heuristic we choose a node u of minimum degree. Now it occurs that one or multiple neighbors of u have the same incident nodes as u itself. Assume that there exists such a neighbor v of u. It was shown by George and Liu [14] that we can always order u and v simultaneously because they insert the same edges between their neighbors if there are missing any. Node u and the neighbors which fulfill this property are called *indistinguishable nodes* [14]. By ordering all indistinguishable nodes at once in each iteration the running time of the algorithm is reduced. In fact, the number of indistinguishable nodes at each iteration should increase over time because we extend the neighborhood of the eliminated nodes to a clique at each elimination.

Figure 12 shows node u and a neighbor v which share the same neighbors and Figure 13 shows the subgraph after eliminating u and v at once.

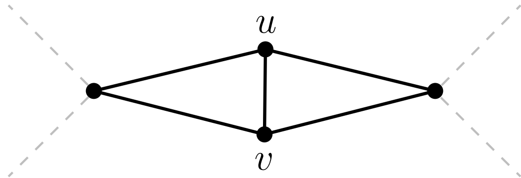

Figure 12: u and v are indistinguishable

Figure 13: The graph after removing u and v in figure 12 simultaneously. The green edge has been added.

Tie-breaking Strategy

At each iteration of the algorithm we choose a node of minimum degree. However, there may be several nodes with minimum degree and experiments have shown that an appropriate tie-breaking strategy for selecting a node of minimum degree can lead to gains in both storage and arithmetic operations [14].

Therefore we implemented two different tie-breaking strategies. The first strategy draws a random node from the set of nodes of minimum degree. The second approach is based on the idea of preordering by George and Liu [14]. Before the graph is passed to the nested dissection algorithm we first run a preordering algorithm which computes an initial ordering used by the actual minimum degree algorithm to break ties. We followed the suggestion of George and Liu and used the *reverse Cuthill-McKee* algorithm for pre-ordering [5]. The complete ordering process can be viewed as

$$A \xrightarrow{RCM} \tilde{A} = P_r A P_r^T \xrightarrow{MD} P \tilde{A} P^T$$

where P_r is the reverse CutHill-McKee (RCM) order on the adjacency matrix A of G and P is the minimum degree (MD) ordering on the permuted matrix $\tilde{A} = P_r A P_r^T$ [14]. However, in our experiments a random selection of nodes was always at least as good as a selection using a preorder.

5.2. Node Separator Balancing

In addition to the improvements of the basic minimum degree heuristic we also extended our node separator algorithm. During our experiments on nested dissection, we noticed that the balance of the node separators is not as important as their size (see Section 6.3). Therefore we allowed more imbalance for our node separators by enlarging the search area described in Section 4.2. We introduced a new parameter ξ which is multiplied with the original balance parameter ϵ to get a higher imbalance. We then define the size of V_1^* to be smaller than $(1 + \epsilon \cdot \xi)\lceil \frac{|V|}{2} \rceil$ with the definitions

from Section 4.2. $|V_2^*|$ is defined symmetrically. Choosing $\xi > 1$, we relax the balancing constraint defined by ϵ and get a larger search area on which our algorithm finds a minimum node separator. Despite the larger search area it turned out that the obtained imbalance was often far less than the possible maximum which shows the importance of the balancing step in our nested dissection algorithm. However, this also leads to an increased imbalance in the recursion tree which increases the overall running time and choosing a too large ξ reduces the node ordering quality. This relaxation of the balancing constraint is only used in nested dissection. For our node separator algorithm we still guarantee the balance defined by the user by setting ξ to one.

5.3. K-way Version

Algorithm 5 shows our k-way version of the nested dissection algorithm which first computes a k-way node separator and then in each further recursion we compute a 2-way node separator as it is known from the basic algorithm. The parameter k can be specified by the user. If k is set to 2 the basic nested dissection algorithm is run.

As we already mentioned in Section 4.1, KaHIP contains randomized algorithms [39, 41] and running our separator algorithm multiple times can improve the overall quality (see Section 6.3). Therefore we provide the possibility to compute several node separators in each recursion step and take the best one found.

If the partition size reaches the minimum degree threshold we stop the recursion and apply our altered version of the minimum degree heuristic as described in Section 5.1. Separator nodes are sorted by their degree in increasing order before we add them to the node ordering.

5.4. Connected Components Version

In this section we want to describe our connected components version of the nested dissection algorithm. The difference to our k-way version is that we search for connected components in the graph in every recursion step. If there are multiple connected components we continue the recursion on those connected components. As they are not connected, we do not need a node separator to separate them. Algorithm 6 shows the connected components version. If G is connected, i.e. there is only one connected component, we calculate a node separator and take the induced partitions as connected components.

During our experiments the connected components version did not show any improvements over the normal version which is why we focused on the latter. A reason might be that the natural connected components found in our test graphs are not very balanced.

Algorithm 5: K-Way Nested Dissection

```
Input: G = (V, E) : Graph, γ : minimum degree threshold
Output: π : node ordering
1  if recursionRoot then
       // compute k-partition of G
2      P = {G_1 = (V_1, E_1), ..., G_k = (V_k, E_k)} ← PartitionGraph(G)
3      (V'_1, ..., V'_k, S) ← ComputeSeparator(G, P)
4  else
       // compute 2-partition of G
5      P = {G_1 = (V_1, E_1), G_2 = (V_2, E_2)} ← PartitionGraph(G)
       (V'_1, V'_2, S) ← ComputeSeparator(G, P)

6  for G'_i = (V'_i, E'_i) in P do
7      if |V'_i| > γ then
8          NestedDissection(G'_i)          // continue recursively if G'_i is too big
9      else
10         MinimumDegree(G'_i, S, π)
11 OrderSeparator(S, π)
```

Algorithm 6: Connected Components Nested Dissection

```
Input: G = (V, E) : Graph, γ : minimum degree threshold
Output: π : node ordering
// Compute connected components of G
1  𝒞 ← ConnectedComponents(G)
2  if |𝒞| == 1 then
       // compute 2-partition of G
3      P = {G_1 = (V_1, E_1), G_2 = (V_2, E_2)} ← PartitionGraph(G)

       // find node separator on G given the partitioning P
4      (V'_1, V'_2, S) ← ComputeSeparator(G, P)
5      𝒞 ← {V'_1, V'_2}
6  else
7      S ← ∅

8  for C in 𝒞 do
9      if |C| > γ then
10         NestedDissection(C)          // continue recursively if C is too big
11     else
12         MinimumDegree(C, S, π)
13 OrderSeparator(S, π)
```

6. Experiments

To test our separator algorithm we have done several experiments. In this chapter we present and discuss the results. The chapter is structured into two main parts. In the first part we assess the quality of our computed separators and in the second part we test our nested dissection algorithm. In general, small separators lead to less fill-in and therefore to good node orderings [21]. As a consequence, we can obtain further insight into the quality of node separators by evaluating the performance in nested dissection.

6.1. Test Environment

All our experiments were done on a system equipped with two Quad-core Intel Xeon processors (X5355) which run at a clock speed of 2.667 GHz, have 2x4 MB of level 2 cache and are equipped with 16 GB of main memory. Our algorithm is completely written in C++ and compiled with gcc 4.6.4.

6.2. Test Instances

We tested our algorithms on two test sets. The first set contains the graphs from the Walshaw Graph Partitioning Archive [44] and consists of several sparse matrices and graphs from numerical simulations in different sizes. The second set is a collection of social network graphs. All test instances and their properties can be found in Table 1.

In all our experiments we did five runs on each graph and took arithmetic means of the results to reduce the deviation due to the random parts of KaHIP. If not stated otherwise, the experiments were done with a set of five test graphs {*add20*, *crack*, *bcsstk30*, *finan512*, *auto*}) which comprise a good mixture of small and large graphs. To get a global result over all five test graphs we computed a geometric mean over the arithmetic means.

6.3. Parameters of our Algorithms

Our node separator and nested dissection algorithm can be configured by several parameters (see Appendix A.2 for a detailed list). The default parameters for our algorithms were determined by tuning the parameters on a small subset of graphs. Since KaHIP offers several different configurations for general graphs and social network graphs, we did one parameter tuning for the general graphs and one for social network graphs to maintain this distinction. The parameter tunings for both the Walshaw and the social network graphs were done on a small subset of our test graphs in Table 1. The values found during the parameter tuning are used as default values in our experiments if not stated otherwise. The exact values are noted in Appendix A.2. The following paragraphs of this section describe the available parameters of our algorithms and motivate our choice of default values found during our parameter tuning.

Graph	Nodes	Edges	Graph	Nodes	Edges
Walshaw Graphs			Social Network Graphs		
add20	2 395	7 462	p2p-Gnutella04	6 405	29 215
data	2 851	15 093	wordassociation-2011	10 617	63 788
3elt	4 720	13 722	PGPgiantcompo	10 680	24 316
uk	4 824	6 837	email-EuAll	16 805	60 260
add32	4 960	9 462	as-22july06	22 963	48 436
bcsstk33	8 738	291 583	loc-brightkite_edges	56 739	212 945
whitaker3	9 800	28 989	loc-gowalla_edges	196 591	950 327
crack	10 240	30 380	coAuthorsCiteseer	227 320	814 134
wing_nodal	10 937	75 488	citationCiteseer	268 495	1 156 647
fe_4elt2	11 143	32 818			
vibrobox	12 328	165 250			
bcsstk29	13 992	302 748			
4elt	15 606	45 878			
fe_sphere	16 386	49 152			
cti	16 840	48 232			
memplus	17 758	54 196			
cs4	22 499	43 858			
bcsstk30	28 924	1 007 284			
bcsstk31	35 588	572 914			
fe_pwt	36 519	144 794			
bcsstk32	44 609	985 046			
fe_body	45 087	163 734			
t60k	60 005	89 440			
wing	62 032	121 544			
brack2	62 631	366 559			
finan512	74 752	261 120			
fe_tooth	78 136	452 591			
fe_rotor	99 617	662 431			
598a	110 971	741 934			
fe_ocean	143 437	409 593			
144	144 649	1 074 393			
wave	156 317	1 059 331			
m14b	214 765	1 679 018			
auto	448 695	3 314 611			

Table 1: Test instances used in our experiments

Node Separator Algorithm

Our node separator algorithm has four parameters: k, *imbalance*, *nSeps* and *preconfiguration*. The k parameter tells our algorithm to find a k-way node separator and the *imbalance* parameter controls the balance of the induced partitions. The *imbalance* parameter is specified in percent, i.e. $\epsilon = imbalance/100$ in the notation of Section 4.2. The number of computed k-way separators can be specified by *nSeps*. If more than one separator is calculated, the algorithm returns the best one found in terms of size and balance. The *preconfiguration* parameter can be one of {*fast*, *fastsocial*, *eco*, *ecosocial*, *strong*, *strongsocial*} and sets the values of many parameters of the graph partitioner KaHIP. Configurations with the suffix *social* are used on social network graphs and the others are dedicated to all other graph classes. While the *strong* configurations tend to produce the best results in terms of solution quality, they consume a lot of time. The *fast* configurations offer faster execution with reduced quality and *eco* configurations provide a trade-off between solution quality and runtime. For more detailed information about this parameter we refer to the manual of KaHIP [40].

K-Way Node Separators

KaHIP is able to compute k-way partitions without recursive partitioning which enables us to directly compute k-way node separators by combining the node separators of each pair of partitions. Since the direct computation of k-way partitions is often better than the recursive approach [42], we wanted to examine if this also applies to k-way node separators. Therefore we computed node separators which induce 2, 4 and 8 partitions with both approaches for our five test graphs. Our nested dissection experiments revealed that an imbalance of 20% leads to good node orderings which is why we also chose a maximum imbalance of 20% for this experiment, i.e. for each node separator computation we allowed this imbalance.

As expected, the obtained 4- and 8-way node separators outperform the recursive 2-way node separators. While the sizes of the separators computed by the k-way approach are only marginally smaller than the recursive approach (1.3% and 1.5%), the actual ϵ and β values as defined in Equations 2.1 and 2.2 of the k-way node separators are far better. This is because we allowed a maximum ϵ imbalance of 20% for each computation and the recursive approach has more separator computations. In practice, the maximum ϵ imbalance for the recursive approach is often chosen to be less on the first level to overcome the high imbalances in subsequent recursion levels. However, this would further increase the size of the separators compared to the k-way approach. Table 2 shows the complete results of this experiment. In our implementation of the nested dissection algorithm we are able to compute a k-way node separator on the first level of recursion. During our experiments, it turned out that a 3-way node separator on the first level leads to better node orderings on the Walshaw graphs. On social networks, a normal 2-way node separator seems to be the better choice.

Nested Dissection Algorithm

The nested dissection algorithm for reducing fill-in during sparse matrix factorization makes use of node separators. As experiments have shown [21, 32], small node separators seem to produce better orderings. Therefore we implemented a nested dissection algorithm (see Section 5) to further evaluate the quality of our new node separator algorithm. We first present some results concerning the choice of parameters before we do a comparison between our nested dissection version and two competitors in Section 6.6.

In addition to the parameters of the node separator algorithm, our nested dissection algorithm offers four more parameters: *final_partition_size*, *balance_multiplier*, *initial_order* and *cc_order*. In the following paragraphs we describe some of the parameters in more detail and show some ordering results which motivated our choice of default values. To asses the quality of node orderings, we used the SCOTCH node ordering benchmark tool which computes a Cholesky factorization of the given matrix based on a specified node ordering. It outputs several quality measures like the number of non-zeros (NNZ) of the factorized matrix and the operation count (OPC) which is the total amount of all arithmetic operations (additions, subtractions, multiplications, divisions) [34]. Additionally, it outputs information about the elimination tree like the minimum and maximum height, the average height and the variance in height. Those numbers can be used to examine the balance of the elimination tree which is important if parallel factorization algorithms should be used [34].

Size Threshold

The parameter *final_partition_size* specifies the minimum degree threshold γ as defined in Section 5.3. The recursion of the nested dissection algorithm stops when the number of nodes in a partition is lower than the threshold value in which case the minimum degree algorithm is run on that partition. Typical values for the threshold are between 30 and 200 nodes. In our experiments we tested several threshold values on our five test graphs. The results are shown in Table 3. It turned out that the number of non-zeros (NNZ) present in the factorized matrix decreases as the threshold is lowered which confirms that nested dissection outperforms the minimum degree heuristic in terms of quality. The lowest NNZ and OPC has been observed at a size threshold of 50. A reason for the increase in NNZ below 50 nodes might be that the overall number of separator nodes increases and we order those nodes by increasing node degree and not with the minimum degree algorithm. We further found out that a small minimum degree threshold leads to an increase in runtime because our nested dissection algorithm has to compute more node separators which is a time consuming task. As a consequence, we use a default size threshold of 80 in our nested dissection algorithm since this value results in a good balance between minimizing NNZ and OPC and is a reasonable trade-off between quality and runtime.

Imbalance and Balance Multiplier

The *balance_multiplier* parameter corresponds to the ξ described in Section 5.2. It allows us to relax the demanded balance between the partitions defined by the *imbalance* parameter. While the edge separator is still computed with the normal balancing constraint defined by *imbalance* we allow a higher imbalance value for our node separator algorithm. This way, we get a larger search area on which our algorithm finds a minimum node separator. We tested the impacts of different *imbalance* and *balance_multiplier* combinations on our five test graphs. The results of Table 4 suggest that an increased search area for computing the node separators reflects in better node orderings but the higher allowed imbalance also leads to a more imbalanced recursion tree which means that the algorithm takes longer. However, we found out that despite the larger search area and the resulting possibility of a higher imbalance between the partitions the actual imbalance is often significantly less than the maximum allowed imbalance which is due to the balancing step of our nested dissection algorithm. The experiments also reveal that the use of the *balance_multiplier* parameter results in better node orderings compared to the same imbalance with a *balance_multiplier* of one. For example, the node ordering with *balance_multiplier*=2 and an *imbalance* of 20% produces on average 2.3% less non-zeros and needs 5.5% less operations than an *imbalance* of 40% and a *balance_multiplier* of one in Table 4. While both parameter choices allow a maximum imbalance of 40% for the node separator, the maximum imbalance for the edge separator is 20% in the case of a *balance_multiplier* of 2 and 40% in the other case which seems to be the reason for the difference. As default values, we chose an *imbalance* of 20% and a *balanace_multiplier* of 3 which produced good node orderings during our parameter tuning.

Number of Separators

As the graph partitioner KaHIP uses randomized algorithms we tested if we could improve our found node separators and therefore the computed node orderings by using multiple edge separators with different random seeds for computing our node separators. We tested several numbers of node separators and the results in Table 5 show that our assumption was right. Increasing the number of computed node separators often leads to smaller node separators and improved node orderings as a consequence. However, the time needed to compute the node ordering also increases and from a certain number of separators there is only few or no improvement at all. Therefore we use 5 node separator computations as default value for the *nSeps* parameter in our algorithms.

k	k-way			recursive 2-way		
	sep. size	ϵ	β	sep. size	ϵ	β
2	128	1.8%	0.8%	128	1.8%	0.8%
4	294	5.9%	1.5%	298	10.4%	2.6%
8	556	7.0%	1.7%	564	17.1%	3.8%

Table 2: Comparison of direct k-way and recursive 2-way node separator computation on our five test graphs. Results show geometric means. Maximum allowed imbalance for each separator computation was 20%.

	Size Threshold				
	25	50	75	100	200
NNZ	+0.6%	best	+0.3%	+0.7%	+1.6%
OPC	+2.1%	best	+0.6%	+2.1%	+3.0%

Table 3: Nested dissection results for our five test graphs. NNZ and OPC results are normalized and show the increase in percent over the best result. For this test we used the *fast* configuration.

	ξ with *imb.* = 20%				*imb.* with $\xi = 1$		
	1	2	3	4	40%	60%	80%
NNZ	+2.9%	best	+0.8%	+9.6%	+2.3%	+4.8%	+25.4%
OPC	+7%	best	+1.9%	+22.6%	+5.5%	+10.2%	+70.4%

Table 4: Ordering results for our five test graphs. On the left we used a constant *imbalance* of 20% with varying *balance_multiplier* values and on the right a constant *balance_multiplier* of 1 and varying *imbalance*. Results are normalized and show the increase in percent over the best result.

	Number of Separators				
	1	2	4	8	10
NNZ	+1.7%	+1.0%	+0.5%	best	+0.1%
OPC	+7.4%	+5.2%	+2.9%	best	+0.4%
Avg. Time [s]	3.97	6.84	12.79	25.00	31.02

Table 5: Nested dissection results for our five test graphs with varying number of separators calculated at each level of recursion. NNZ and OPC results are normalized and show the increase in percent over the best result. For this test we used the *fast* configuration.

6.4. Competitors

We compared our algorithms with three other tools. First, we did a comparison of our new algorithm and the node separator algorithm which was already implemented in KaHIP [42]. Both our new and the previous algorithm rely on the graph partitionings computed by KaHIP and therefore have the same setting. We were interested in the size of the obtained node separators as well as the balance of the induced partitions after removing the separator. Additionally, we compared the quality of our node separators on the test instances of Table 1 to the ones obtained by METIS [21]. The results are shown in Section 6.5.

Our nested dissection algorithm was also tested on both the Walshaw graphs and the social network graphs. In the previous section we already presented some experiments regarding the parameter choice of our nested dissection algorithm and in Section 6.6 we compare it to the nested dissection algorithms of METIS [21] and SCOTCH [33]. Table 6 shows the competitors and the versions we used during our tests.

Tool	Version
KaHIP	0.6
SCOTCH	6.0.0
METIS	5.1.0

Table 6: Versions of the competing tools

6.5. Separator Quality

To asses the quality of a node separator, we use three quality measures. The first one is the size of the node separator and the other two rate the balance of the partitions as described in Section 2.5. Given a user specified maximum imbalance (ϵ in percent), the computed separator should be small and the actual ϵ (see Equation 2.1) should not exceed the defined maximum imbalance. We also evaluated the balance among the partitions after removing the separator from the graph (β in Equation 2.2). To break ties if two separators are equal in size, the actual ϵ imbalance is used to rank the separators.

Comparison of KaHIP and METIS Node Separators

We compared the node separators of our new algorithm with the separators of the old algorithm implemented in KaHIP and the METIS node separator algorithm. Both METIS and SCOTCH do not offer a separate program for calculating node separators. However, we were able to get the size of the partitions and the separators obtained by the METIS nested dissection algorithm. On the first level of recursion the obtained node separator and partitions are equal to a solution of the node separator problem on the complete graph.

The KaHIP algorithm computes a minimum vertex cover on the nodes induced by the cut edges of the edge separator [42] (see Section 2.5). Since both the KaHIP node

separator and our new algorithm use the same edge separator computed by the graph partitioner of KaHIP the differences in quality only result from the different node separator algorithms which makes it very easy to compare the results. In contrast, the METIS node separator algorithm computes the separators with a multilevel approach (see Section 2.5).

Walshaw Graphs

We computed 2-way node separators of our test instances in Table 1 with an allowed maximum imbalance of 20%. Both our own and the METIS nested dissection algorithm use a default imbalance of 20% on the Walshaw graphs which is why we also chose that value for this test. The KaHIP algorithms were run with the *strong* configuration.

Compared to the old node separator algorithm in KaHIP, we had an improvement in node separator size in more than 50% of all runs, on average the size was more than 3% less. The largest improvement was 25%. Although our new algorithm computed smaller node separators, the actual imbalance was equal or lower than the one induced by the KaHIP algorithm in more than 60% of all runs.

Our new algorithm outperforms METIS in more than 70% of the runs and in another 20% it has similar quality in terms of size. On average, our node separators are smaller by 6% while maintaining the balancing constraint. The largest improvement was a factor of two. As the sizes of our node separators are smaller on most instances it is reasonable that the imbalance is higher compared to the larger node separators obtained by METIS. However, on some graphs we got smaller node separators even though the imbalance was lower than the METIS separator imbalance.

It turned out that the multilevel computation is the crucial part of the METIS node separator algorithm. We also did some tests with the METIS algorithm with multilevel computation turned off and the computed separators were all larger than the ones computed by our algorithm and the normal METIS node separator algorithm.

The node separators computed by the former KaHIP node separator algorithm also outperformed the METIS separators in more than 40% of the runs which shows that small edge separators lead to better node separators, as the KaHIP partitioning routines outperform the METIS routines in terms of solution quality [42]. Table 8 shows the complete results of this comparison. Bold values mark the best result in terms of separator size for each test graph.

Social Network Graphs

Next, we tested our node separator algorithm on the set of social network graphs listed in Table 1. Again, we computed 2-way separators with an allowed maximum imbalance of 20% with our new algorithm, the former KaHIP node separator algorithm and the one provided by METIS. The KaHIP algorithms were run with the *strongsocial* configuration.

It turned our that the KaHIP algorithms were significantly worse on social networks compared to the METIS algorithm. In 80% of the runs the METIS separators outperformed the ones computed by our new algorithm. However, in 90% of all runs our

algorithm computed separators with significantly better balance values (on average by a factor of 15). This is expected, because the METIS nested dissection algorithm uses a multilevel approach to compute node separators which is better than our approach on social networks. The reason for this can be found in the structure of social network graphs. They consist of dense clusters which have a lot of intra-cluster edges and relatively few connections to other clusters. Therefore the breadth-first searches of our algorithm are not very effective because the search area is often dense which leads to larger node separators. In this setting, the multilevel approach of METIS to compute node separators is highly beneficial because it is able to coarsen the dense parts of clusters and can therefore find smaller node separators in regions of the cluster which are less dense.

Comparing the old algorithm to our new approach, the results show that we are able to reduce the size of the computed node separators significantly in 80% of the runs. On average, our new algorithm computes 93% smaller separators. Despite the separators are smaller, the balance values are often better in comparison with the old algorithm. Table 7 shows the complete results of this comparison. Bold values mark the best result in terms of separator size for each test graph.

Graph	New Algorithm			KaHIP			METIS		
	size	ϵ	β	size	ϵ	β	size	ϵ	β
as-22july06	**155**	-0.2	0.5	294	-0.6	0.7	178	19.7	20.6
citationCiteseer	9 121	-3.2	0.2	9 872	-2.0	1.7	**7 574**	18.4	21.8
coAuthorsCiteseer	5 047	17.7	20.4	5 190	16.8	19.5	**3 875**	19.5	21.6
email-EuAll	**5**	16.7	16.7	46	14.2	14.5	**5**	16.1	16.2
enron	1 078	18.3	20.1	1 136	17.3	19.2	**563**	20.0	20.9
loc-brightk.	3 395	14.1	21.3	3 474	17.2	24.9	**2 336**	19.9	25.1
loc-gow.	7 542	16.2	20.8	7 785	18.3	20.9	**5 852**	20.0	23.7
p2p-Gnutella04	2 143	-3.6	44.9	2 143	16.6	75.3	**1 639**	19.9	61.1
PGPgiantcompo	121	6.4	7.6	128	6.6	7.9	**97**	19.1	20.2
wordassoc.	1 805	2.9	24.0	1 805	-0.9	24.0	**1 515**	19.9	39.9
geom. mean	926	5.8	8.8	1266	6.5	11.8	**739**	19.2	25.0

Table 7: Separator Quality of a 2-way node separator with an allowed imbalance of 20% of the social network graphs. Bold values are best in the size column. For this experiment we used the *strongsocial* configuration of our algorithm.

Graph	New Algorithm			KaHIP			METIS		
	size	ϵ	β	size	ϵ	β	size	ϵ	β
144	**1 464**	16.3	17.5	1 496	18.7	19.9	1 520	14.8	16.0
3elt	**42**	13.5	14.5	**42**	13.3	14.3	**42**	11.4	12.4
4elt	**68**	2.1	2.6	**68**	1.8	2.3	**68**	1.2	1.6
598a	**596**	6.7	7.2	603	6.5	7.1	597	3.3	3.9
add20	24	12.2	13.3	30	14.6	16.1	**23**	6.9	7.9
add32	**1**	12.5	12.5	**1**	12.5	12.5	2	6.2	6.3
auto	2 241	4.9	5.4	2 289	4.4	4.9	**2 087**	18.1	18.6
bcsstk29	**180**	10.3	11.7	**180**	11.6	13.0	**180**	-0.9	0.4
bcsstk30	**206**	0.1	0.8	**206**	0.0	0.8	218	16.1	17.0
bcsstk31	**270**	9.6	10.4	291	9.5	10.4	294	7.39	8.3
bcsstk32	**243**	18.4	19.1	258	19.2	19.9	263	19.4	20.1
bcsstk33	**421**	13.9	19.6	**421**	13.9	19.6	**421**	-4.2	0.6
brack2	**181**	8.0	8.3	202	9.3	9.6	183	2.9	3.2
crack	**70**	15.2	16.0	82	19.8	20.7	75	-0.7	0.0
cs4	**281**	14.2	15.6	311	17.0	18.7	296	9.9	11.3
cti	**266**	8.6	10.4	**266**	10.2	12.0	275	7.4	9.2
data	**43**	16.8	18.6	47	16.8	18.8	56	14.3	16.6
fe_4elt2	**66**	2.0	2.6	**66**	1.4	2.0	**66**	-0.6	0.0
fe_body	**91**	-0.2	0.0	95	-0.1	0.1	99	15.3	15.6
fe_ocean	**263**	2.1	2.2	311	1.8	2.0	267	1.8	2.0
fe_pwt	**116**	19.1	19.5	**116**	19.4	19.8	120	3.2	3.5
fe_rotor	**439**	5.6	6.0	443	4.9	5.3	447	4.7	5.1
fe_sphere	**192**	10.6	11.9	**192**	10.6	11.9	**192**	-1.2	0.0
fe_tooth	**877**	17.8	19.2	900	18.1	19.5	888	16.7	18.0
finan512	**50**	-0.1	0.0	**50**	-0.1	0.0	**50**	12.4	12.5
m14b	**835**	4.5	4.9	854	5.9	6.3	887	3.6	4.0
memplus	134	19.1	20.0	135	18.8	19.7	**92**	19.9	20.5
t60k	**56**	9.7	9.8	**56**	9.8	9.9	65	4.6	4.7
uk	**14**	19.4	19.7	**14**	19.1	19.4	15	17.3	17.6
vibrobox	666	-5.4	0.0	666	-5.4	0.0	**598**	10.1	15.7
wave	**2 111**	16.2	17.8	2 120	17.2	18.8	2 363	-1.5	0.0
whitaker3	**62**	19.3	20.0	**62**	18.6	19.4	64	-4.3	0.2
wing	**613**	15.5	16.7	681	17.7	19.0	658	7.2	8.3
wing_nodal	387	16.3	20.6	387	18.8	23.1	**385**	17.2	21.4
geom. mean	**170**	6.6	4.3	176	6.6	4.7	177	5.6	1.9

Table 8: Separator Quality of a 2-way node separator with an allowed imbalance of 20% of the Walshaw graphs. Bold values are best in the size column. For this experiment we used the *strong* configuration of our algorithm.

6.6. Nested Dissection

Here, we compare our nested dissection algorithm which makes use of the node separators computed with our new algorithm to SCOTCH [33] and METIS [20] which are popular fill-in optimization tools and implement the nested dissection algorithm. As we integrated our algorithms in the open source graph partitioning package KaHIP, we denote our nested dissection algorithm by KaHIP in the following paragraphs.

Walshaw Graphs

We start with the Walshaw graphs of Table 1. We did three test runs of our own algorithm with the different configurations offered by the graph partitioner KaHIP. The complete results can be found in appendix A.1. Here we compare the best results of our algorithm computed with the *strong* configuration of KaHIP. Figures 14 and 16 show the ratio of the NNZ and OPC of the SCOTCH orderings to our node orderings and Figures 15 and 17 the same ratios with the METIS orderings.

Our new algorithm outperforms SCOTCH in all runs we have made on the Walshaw graphs. On average, the node orderings computed with KaHIP had 31% less non-zeros than the SCOTCH orderings. In terms of operation count, the KaHIP node orderings produce less operations in more than 90% of the runs, on average our orderings reduce the operation count by 50% compared to the SCOTCH orderings. For example, our node ordering for matrix *auto* produces more than 10 million less non-zero entries in the factorized matrix than the SCOTCH ordering and we save more than 9 billion operations during the factorization of matrix *144*. Reasons for this huge difference are the better edge separators computed by KaHIP [42] and our new node separator algorithm.

Comparing our node orderings with METIS, we are able to produce better orderings in more than 70% of the runs. However, the difference in NNZ and OPC is a lot smaller. On average, the NNZ of our node orderings is smaller by 3% and the OPC is smaller by 4%. Despite the difference is rather small, the gains are still in the region of millions. For example, our node ordering produces more than 1 million non-zeros less on matrix *m14b* and we save over 8 billion operations on matrix *wave* compared to the METIS orderings.

Concerning the elimination trees, our node orderings tend to produce slightly less balanced trees in most runs compared to SCOTCH and METIS. This is a consequence of the higher allowed maximum imbalance during nested dissection. However, on average the balance difference is less than 10% and as the experiments show, our node orderings are better in most runs concerning the number of non-zeros and the operation count which is why we decided to allow a high imbalance. If parallel matrix factorization algorithms are used, the imbalance of our nested dissection algorithm can be lowered to increase the balance of the elimination trees.

The quality of our node orderings comes at a price: our algorithm in its best configuration consumes a lot of time compared to METIS and SCOTCH, on average our algorithm is slower by a factor of about 1000 with the *strong* configuration. The reason for that difference is the time consuming graph partitioning of KaHIP and our min-

imum degree algorithm which is not the fastest compared to other implementations. Since nested dissection is a divide-and-conquer algorithm, it should be easy to parallelize which should lower the time difference effectively. Also, a better implementation of the minimum degree algorithm can further reduce the running time. However, we think that, depending on the application, the gains of our node orderings outweigh the needed amount of time, as factorized matrices are often used multiple times.

If running time is more important, our algorithm offers the two configurations *eco* and *fast* which reduce the time consumption but also the solution quality (see Appendix A.1 for detailed results).

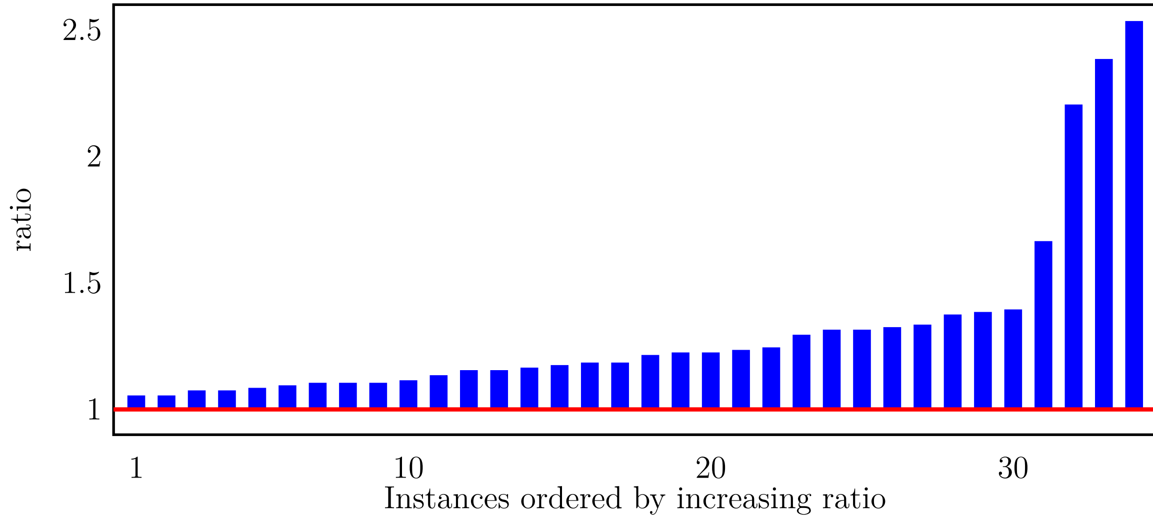

Figure 14: $\frac{SCOTCH}{KaHIP}$ NNZ ratio of the Walshaw graphs.

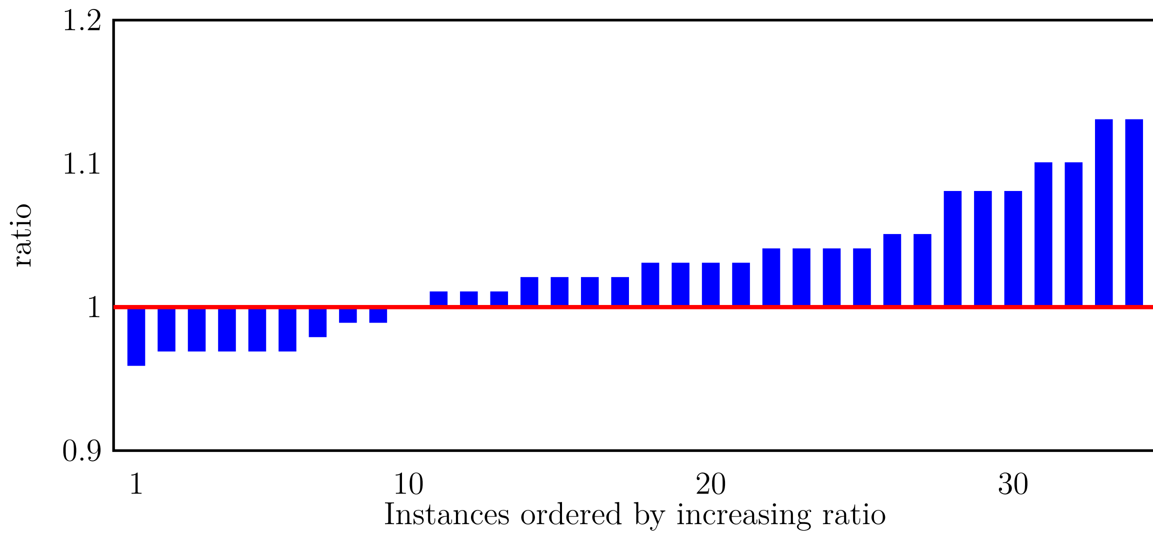

Figure 15: $\frac{METIS}{KaHIP}$ NNZ ratio of the Walshaw graphs.

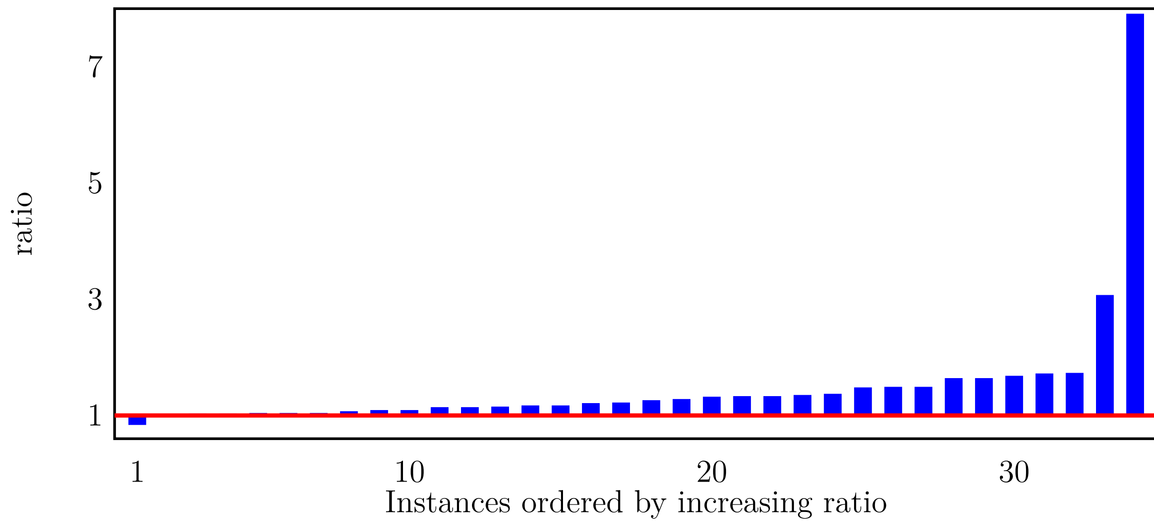

Figure 16: $\frac{SCOTCH}{KaHIP}$ OPC ratio of the Walshaw graphs.

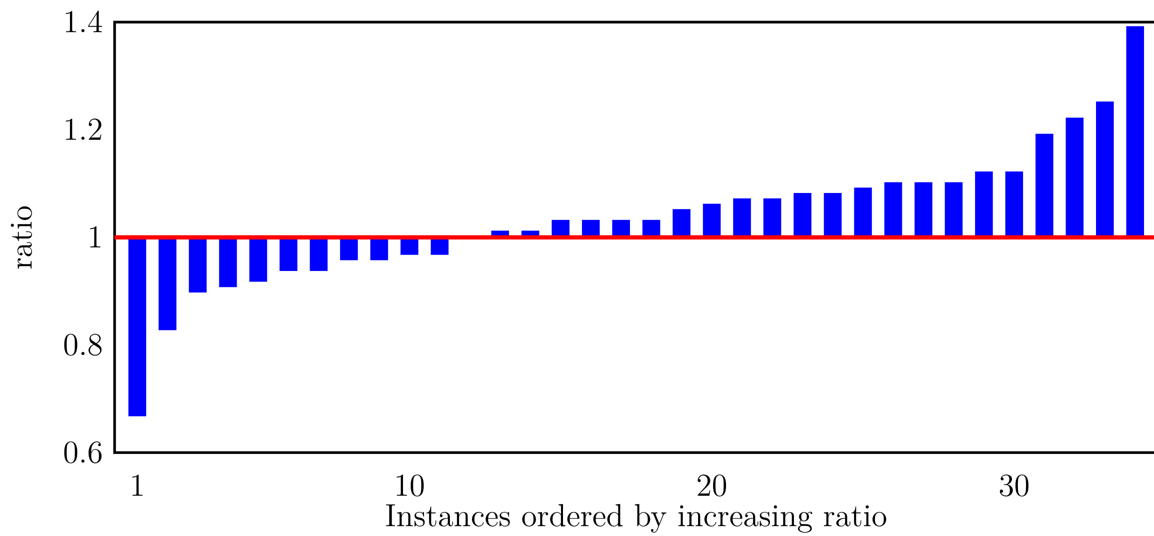

Figure 17: $\frac{METIS}{KaHIP}$ OPC ratio of the Walshaw graphs.

Social Network Graphs

In this section we want to present the results of our nested dissection tests on the set of social network graphs found in Table 1. Again, we compared the node orderings of our new Algorithm in KaHIP with the orderings produced by METIS and SCOTCH. As KaHIP offers three parameter configurations for partitioning social network graphs ($fastsocial$, *ecosocial* and *strongsocial*), we also offer those three configurations for our nested dissection algorithm. The complete results can be found in Appendix A.1. Here we compare the results of our algorithm with the *strongsocial* configuration as it produced the best results among all three configurations. Figures 18 and 20 show the ratio of the NNZ and OPC of the SCOTCH orderings to our node orderings and Figures 19 and 21 the same ratios with the METIS orderings.

It turned out that on social network graphs, our algorithm performs worse in 80% of the runs compared to METIS. The NNZ is higher by 8% and the OPC by 11% on average. The reason for this is again the multilevel approach as it was during the evaluation of the separator quality on social network graphs in Section 6.5. The breadth-first searches of our node separator algorithm often find dense search areas and therefore larger node separators compared to the ones obtained by the multilevel node separator algorithm of METIS. The SCOTCH nested dissection algorithm does not use a multilevel approach and in comparison, we obtain better node orderings in 80% of the runs. The difference in NNZ and OPC is even higher than on the Walshaw graph collection. On average, we get 28% less NNZ and 69% less operations. This shows that our algorithm outperforms the SCOTCH algorithm which both do not use multilevel techniques to compute node separators.

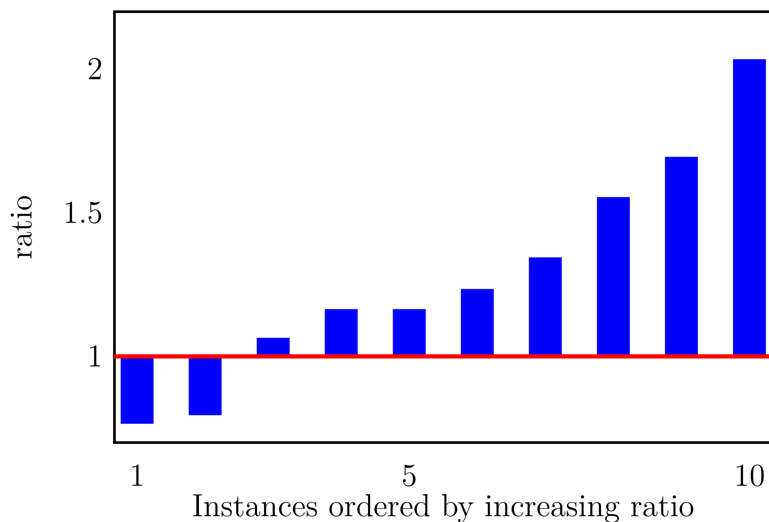

Figure 18: $\frac{SCOTCH}{KaHIP}$ NNZ ratio of the social network graphs.

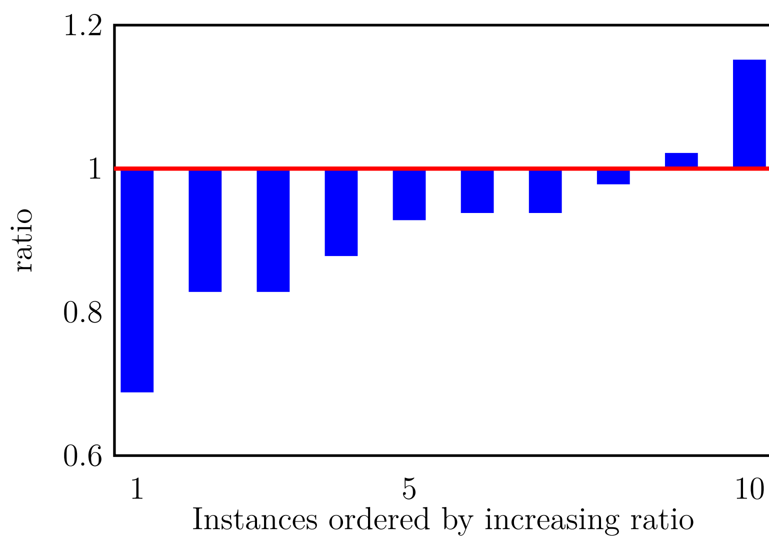

Figure 19: $\frac{METIS}{KaHIP}$ NNZ ratio of the social network graphs.

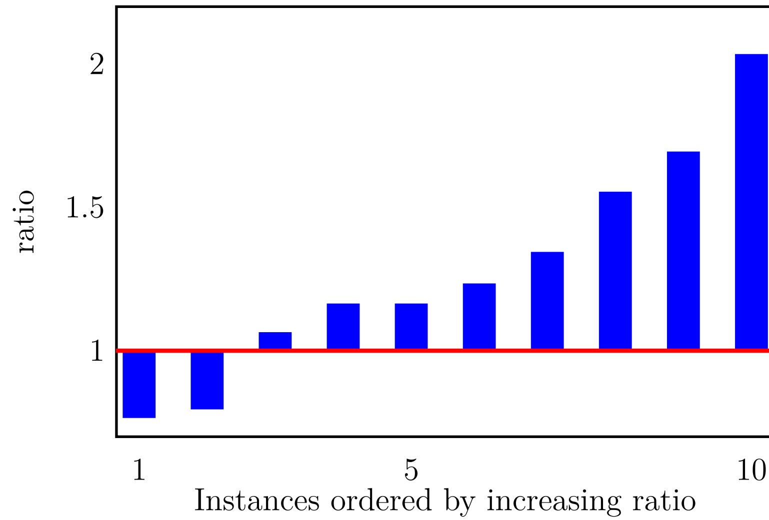

Figure 20: $\frac{SCOTCH}{KaHIP}$ OPC ratio of the social network graphs.

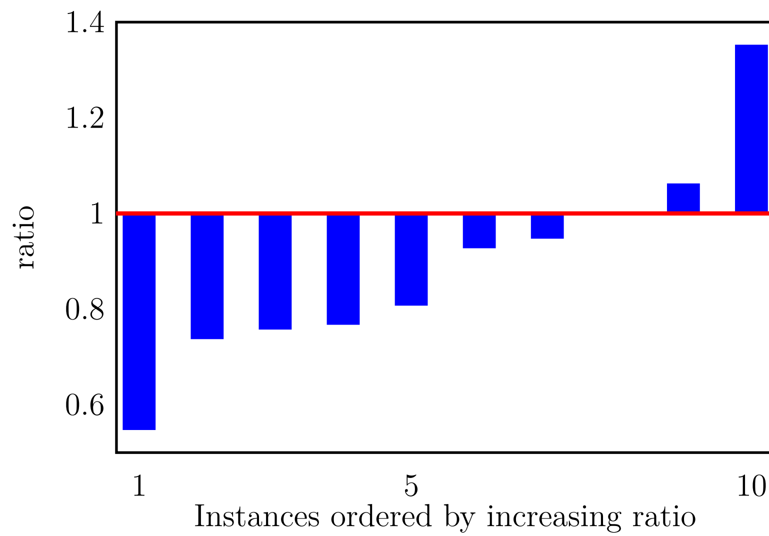

Figure 21: $\frac{METIS}{KaHIP}$ OPC ratio of the social network graphs.

7. Conclusions

The goal of this work was to design and implement a new algorithm to find small node separators on large, undirected graphs which fulfill a user defined balancing constraint. The basic idea of our new algorithm is to transform a given edge separator into a node separator. As shown in our experiments, the quality of node separators computed by this approach relies on high quality edge separators which is why we used the open source graph partitioning package KaHIP to compute the edge separators as it is one of the best graph partitioners in terms of solution quality at the moment [42]. We use an extended search area around the edge separator on which our algorithm finds a minimum node separator by means of flow techniques.

Assessing the quality of node separators can be done by two categories of quality measures: the size of the obtained separator and the balance of the induced partitions. The results of our experiments show that our algorithm is able to find significantly smaller node separators than popular competitors. On some test instances, the balance of the induced partitions is even lower than the ones of our competitors although the separator size is substantially smaller.

One of the most popular applications of node separators is nested dissection [12, 25, 13], a divide-and-conquer algorithm used to reduce the fill-in generated during sparse matrix factorization by computing a permutation matrix which is factored instead of the original one. The original matrix is modeled as a graph and an ordering on the nodes of the graph induces a permutation matrix. It turns out that small node separators tend to produce good node orderings [21, 32]. Therefore we implemented a nested dissection algorithm to further evaluate our node separators.

Our experiments show that we are able to compute better node orderings than our two competitors SCOTCH [33] and METIS [20] on most of the Walshaw graphs. However, on social network graphs our algorithm performs worse than the METIS algorithm. METIS makes use of multilevel techniques to compute node separators which turned out to be the reason for the better node orderings.

The runtime of our algorithms is substantially longer than that of our competitors, however, we think that the gains in nested dissection often outweigh the increased running time since most factorized matrices are used in more than one computation and thus can profit from less fill-in.

7.1. Future work

We have several ideas for improving both the node separator algorithm as well as our nested dissection algorithm in terms of quality and runtime. Parallelizing the computation of multiple node separators should reduce the runtime of our node separator algorithm. Also, nested dissection is a divide-and-conquer algorithm and relies on recursion which should make it easy to parallelize our node ordering algorithm as well. Reimplementing the minimum degree algorithm with suitable data structures and the improvements which have been proposed since its first version should also increase the speed of nested dissection.

According to the results of our experiments, a multilevel approach to compute node separators is a promising improvement concerning the quality. Moreover, a *minimum fill-in algorithm* developed by Ng and Peyton [29] seems to outperform the minimum degree algorithm in terms of quality which could be a good replacement for the minimum degree heuristic in our nested dissection algorithm.

A. Appendix

A.1. Detailed Results of our Nested Dissection Experiments

Graph	NNZ	OPC	Time [s]
144	4.60972e+07	5.15101e+10	654.68
3elt	8.73180e+04	2.57038e+06	4.34
4elt	3.35472e+05	1.26842e+07	16.56
598a	2.65131e+07	1.96528e+10	425.96
add20	1.12520e+04	1.54654e+05	1.94
add32	1.45880e+04	4.46690e+04	2.75
auto	2.26263e+08	4.81905e+11	2 182.71
bcsstk29	1.55896e+06	3.16512e+08	40.35
bcsstk30	3.97327e+06	1.01215e+09	163.17
bcsstk31	3.99568e+06	1.01070e+09	109.44
bcsstk32	4.93026e+06	9.60662e+08	196.73
bcsstk33	2.19729e+06	9.31301e+08	33.54
brack2	5.79289e+06	1.71422e+09	144.60
crack	1.64491e+05	6.65377e+06	12.47
cs4	1.35567e+06	3.71936e+08	23.99
cti	1.65387e+06	4.99533e+08	16.40
data	8.19520e+04	4.02415e+06	2.63
fe_4elt2	2.48037e+05	1.08184e+07	10.77
fe_body	8.66595e+05	4.30356e+07	53.11
fe_ocean	1.96785e+07	1.12193e+10	260.37
fe_pwt	1.31222e+06	1.01344e+08	51.05
fe_rotor	1.58556e+07	9.73760e+09	304.34
fe_sphere	6.06621e+05	6.10906e+07	18.97
fe_tooth	1.05646e+07	6.52269e+09	293.03
finan512	1.74690e+06	1.38799e+08	86.86
m14b	6.16940e+07	5.76066e+10	779.96
memplus	7.99440e+04	2.09841e+06	5.82
t60k	9.60169e+05	5.23068e+07	52.77
uk	3.30130e+04	4.07655e+05	3.46
vibrobox	2.41332e+06	1.27633e+09	34.56
wave	6.22756e+07	9.77027e+10	575.28
whitaker3	2.61568e+05	1.50020e+07	9.87
wing	5.16558e+06	2.42268e+10	96.17
wing_nodal	1.78994e+06	6.29961e+08	20.42

Table 9: Node orderings of Walshaw Graphs computed by KaHIP with *eco* configuration.

Graph	NNZ	OPC	Time [s]
144	4.57928e+07	5.08702e+10	152.22
3elt	8.69390e+04	2.54948e+06	1.37
4elt	3.36708e+05	1.29354e+07	6.45
598a	2.66481e+07	2.00712e+10	108.50
add20	1.13800e+04	1.61628e+05	0.55
add32	1.45780e+04	4.45990e+04	0.80
auto	2.30622e+08	5.11669e+11	523.12
bcsstk29	1.54006e+06	3.03537e+08	17.38
bcsstk30	3.95055e+06	1.00096e+09	50.65
bcsstk31	4.00583e+06	1.02562e+09	45.92
bcsstk32	5.04040e+06	1.03933e+09	68.56
bcsstk33	2.12169e+06	8.14897e+08	14.35
brack2	5.76260e+06	1.70536e+09	42.34
crack	1.61898e+05	6.27768e+06	4.64
cs4	1.36080e+06	3.71636e+08	7.11
cti	1.62706e+06	4.92407e+08	5.97
data	8.06970e+04	3.88169e+06	0.84
fe_4elt2	2.49993e+05	1.11725e+07	3.26
fe_body	8.68867e+05	4.33624e+07	18.35
fe_ocean	1.94782e+07	1.12660e+10	64.99
fe_pwt	1.31206e+06	1.00937e+08	17.01
fe_rotor	1.60244e+07	9.87893e+09	88.17
fe_sphere	6.03205e+05	5.99980e+07	6.12
fe_tooth	1.05157e+07	6.61378e+09	63.53
finan512	1.73707e+06	1.36267e+08	29.84
m14b	6.16879e+07	5.71156e+10	211.19
memplus	7.96940e+04	1.99505e+06	2.38
t60k	9.64384e+05	5.38882e+07	18.05
uk	3.28190e+04	3.93750e+05	0.90
vibrobox	2.30358e+06	1.10993e+09	9.77
wave	6.05240e+07	9.17521e+10	132.24
whitaker3	2.61255e+05	1.49919e+07	3.10
wing	5.18022e+06	2.45405e+09	25.81
wing_nodal	1.79220e+06	6.33325e+08	7.06

Table 10: Node orderings of Walshaw Graphs computed by KaHIP with *fast* configuration.

Graph	KaHIP			METIS			SCOTCH		
	NNZ	OPC	Time [s]	NNZ	OPC	Time [s]	NNZ	OPC	Time [s]
144	**4.62389e+07**	**5.28265e+10**	2805.81	4.70048e+07	5.46315e+10	2.76	5.09426e+07	6.27831e+10	4.58
3elt	**8.73630e+04**	**2.58212e+06**	59.84	9.02880e+04	2.70243e+06	0.02	1.06118e+05	3.39470e+06	0.03
4elt	**3.34440e+05**	**1.26122e+07**	194.00	3.46903e+05	1.34470e+07	0.10	4.07393e+05	1.63542e+07	0.12
598a	2.62539e+07	1.90887e+10	1955.12	**2.57324e+07**	**1.79867e+10**	1.88	2.75243e+07	1.92858e+10	3.37
add20	**1.12200e+04**	1.50428e+05	24.04	1.12420e+04	**1.24621e+05**	0.01	2.66860e+04	4.58954e+05	0.02
add32	**1.45710e+04**	**4.44980e+04**	39.81	1.51520e+04	4.91680e+04	0.02	3.68720e+04	3.51002e+05	0.03
auto	2.27705e+08	4.99245e+11	9527.93	**2.21083e+08**	**4.51632e+11**	9.43	2.38122e+08	5.03724e+11	16.90
bcsstk29	**1.56715e+06**	**3.16974e+08**	420.61	1.64491e+06	3.26282e+08	0.20	1.72685e+06	3.39743e+08	0.46
bcsstk30	**3.93775e+06**	**9.86995e+08**	1320.00	4.34326e+06	1.10213e+09	0.36	5.42439e+06	1.44991e+09	0.52
bcsstk31	**3.94860e+06**	**9.96661e+08**	957.80	4.28251e+06	1.11319e+09	0.47	5.16593e+06	1.30433e+09	0.57
bcsstk32	**4.95826e+06**	**9.90494e+08**	1398.22	5.35016e+06	1.068550e+08	0.45	6.90921e+06	1.68973e+09	0.53
bcsstk33	**1.89920e+06**	**6.05767e+08**	322.48	2.14865e+06	7.58336e+08	0.14	2.51055e+06	1.00792e+09	0.25
brack2	**5.77089e+06**	**1.70874e+09**	1022.07	5.83648e+06	1.73326e+09	0.86	7.05663e+06	1.92917e+09	1.53
crack	**1.62508e+05**	**6.32520e+06**	120.95	1.71228e+05	6.81889e+06	0.07	2.69911e+05	1.076303e+06	0.10
cs4	**1.34059e+06**	**3.54646e+08**	251.07	1.38185e+06	3.75781e+08	0.21	1.55917e+06	3.98093e+08	0.31
cti	1.62046e+06	4.96026e+08	197.14	**1.56973e+06**	**4.76789e+08**	0.15	2.08448e+06	6.58545e+08	0.27
data	**7.93110e+04**	**3.62930e+06**	35.99	8.07540e+04	3.66389e+06	0.02	9.80770e+04	4.90268e+06	0.02
fe_4elt2	**2.48919e+05**	**1.08271e+07**	131.00	2.56907e+05	1.15996e+07	0.07	3.06653e+05	1.36879e+07	0.09
fe_body	**8.64606e+05**	**4.25276e+07**	567.87	9.47133e+05	5.17752e+07	0.39	1.18653e+06	6.20731e+07	0.74
fe_ocean	1.92321e+07	1.05269e+10	1928.35	**1.84019e+07**	**1.01500e+10**	1.79	2.13803e+07	1.07598e+10	2.61
fe_pwt	**1.31073e+06**	**1.00699e+08**	437.29	1.34526e+06	1.03926e+08	0.30	1.54060e+06	1.15456e+08	0.42
fe_rotor	**1.55138e+07**	8.91062e+09	1908.06	1.56614e+07	**8.67648e+09**	1.66	1.66146e+07	8.86402e+09	2.87
fe_sphere	**6.00558e+05**	**5.91518e+07**	195.78	6.23539e+05	6.52619e+07	0.11	7.09075e+05	7.33413e+07	0.14
fe_tooth	1.05187e+07	6.36099e+09	1352.14	**1.04085e+07**	**6.18692e+09**	1.15	1.20802e+07	6.68905e+09	1.98
finan512	1.79253e+06	1.46955e+08	803.04	**1.73893e+06**	**1.38213e+08**	0.80	2.10400e+06	1.25863e+08	1.68
m14b	**6.15161e+07**	**5.74206e+10**	4238.82	6.29036e+07	5.93735e+10	4.14	6.69060e+07	6.44360e+10	7.28
memplus	7.99740e+04	2.18767e+06	53.05	**7.75590e+04**	**1.45931e+06**	0.10	1.75681e+05	3.54197e+06	0.18
t60k	**9.43860e+05**	**4.95295e+07**	603.36	9.82389e+05	5.43881e+07	0.38	1.23491e+06	7.26268e+07	0.48
uk	**3.24850e+04**	**3.77700e+05**	44.03	3.49900e+04	4.50987e+05	0.02	4.31230e+04	6.13313e+06	0.02
vibrobox	**1.90292e+06**	**6.92730e+08**	257.81	2.14998e+06	9.61143e+08	0.23	2.14664e+06	8.28242e+08	0.65
wave	**6.10884e+07**	**9.32576e+10**	2778.68	6.19391e+07	1.01337e+11	2.62	6.60146e+07	1.07125e+11	4.59
whitaker3	2.61772e+05	1.50479e+07	113.31	**2.58575e+05**	**1.36711e+07**	0.06	3.02161e+05	1.60937e+07	0.08
wing	**5.15960e+06**	**2.44252e+09**	713.10	5.24867e+06	2.45411e+09	0.65	5.69084e+06	2.48707e+09	1.01
wing_nodal	1.79341e+06	6.36212e+08	168.10	**1.73209e+06**	**5.84666e+08**	0.14	1.91018e+06	6.47859e+08	0.27

Table 11: Walshaw graph node orderings computed by KaHIP (*strong*), METIS and SCOTCH.

Graph	NNZ	OPC	Time [s]
as-22july06	1.33493e+05	7.94151e+06	5.91
citationCiteseer	2.94849e+08	2.40317e+12	166.71
coAuthorsCiteseer	5.40321e+07	2.58449e+11	98.79
email-euAll	9.97715e+05	7.99041e+08	7.37
enron	3.38040e+06	4.13800e+09	21.33
loc-brightkite_edges	1.72366e+07	5.29463e+10	22.03
loc-gowalla_edges	1.52429e+08	1.03092e+12	109.70
p2p-Gnutella04	4.64651e+06	8.83766e+09	5.10
PGPgiantcompo	8.62960e+04	6.02773e+06	2.55
wordassociation-2011	3.92888e+06	5.85472e+09	5.42

Table 12: Node orderings of social network graphs computed by KaHIP with *ecosocial* configuration.

Graph	NNZ	OPC	Time [s]
as-22july06	1.34856e+05	8.10778e+06	3.48
citationCiteseer	3.02807e+08	2.51377e+12	90.27
coAuthorsCiteseer	5.58985e+07	2.75480e+11	49.83
email-euAll	9.54509e+05	7.45058e+08	5.48
enron	3.40638e+06	4.15943e+09	13.62
loc-brightkite_edges	1.75623e+07	5.44263e+10	12.74
loc-gowalla_edges	1.60170e+08	1.12828e+12	57.43
p2p-Gnutella04	4.71070e+06	9.00252e+09	4.26
PGPgiantcompo	8.66800e+04	6.19643e+06	1.54
wordassociation-2011	3.93881e+06	5.93158e+09	3.25

Table 13: Node orderings of social network graphs computed by KaHIP with *fastsocial* configuration.

Graph	KaHIP			METIS			SCOTCH		
	NNZ	OPC	Time [s]	NNZ	OPC	Time [s]	NNZ	OPC	Time [s]
as-22july06	1.33158e+05	7.77049e+06	133.25	**1.31037e+05**	**7.74214e+06**	0.22	2.70429e+05	3.65583e+07	0.31
citationCiteseer	2.91065e+08	2.36219e+12	3071.30	**2.71990e+08**	**2.19115e+12**	7.50	3.09662e+08	2.66782e+12	9.89
coAuthorsCiteseer	**5.08472e+07**	**2.29987e+11**	1435.00	5.16872e+07	2.43575e+11	2.50	7.86049e+07	5.20687e+11	6.09
email-euAll	1.05103e+06	8.60948e+08	175.20	**7.24254e+05**	**4.73813e+08**	0.24	8.40551e+05	5.47432e+08	0.25
enron	3.33978e+06	3.98273e+09	400.33	**2.78381e+06**	**2.92861e+09**	0.98	4.09511e+06	5.20986e+09	1.30
loc-brightkite_edges	1.66284e+07	4.93627e+10	420.50	**1.56861e+07**	**4.67420e+10**	0.91	2.23329e+07	8.27869e+10	1.15
loc-gowalla_edges	1.58194e+08	1.11998e+12	2310.00	**1.39353e+08**	**9.01695e+11**	5.11	1.84228e+08	1.53688e+12	6.74
p2p-Gnutella04	4.73049e+06	9.06288e+09	91.20	3.92199e+06	6.94123e+09	0.15	**3.63750e+06**	**5.59615e+09**	0.17
PGPgiantcompo	8.22000e+04	5.23026e+06	43.72	**7.74260e+04**	**3.98344e+06**	0.07	1.39043e+05	1.02087e+07	0.14
wordassociation-2011	**3.82053e+06**	**5.60690e+09**	66.15	4.37669e+06	7.59091e+09	0.19	4.44859e+06	7.14352e+09	0.32

Table 14: Node orderings of social network graphs computed by KaHIP (*strongsocial*), METIS and SCOTCH.

A.2. Parameters of our Algorithms

Detailed list of the parameters of our node separator and nested dissection algorithms with default values used in our experiments if not stated otherwise.

Node Separator Algorithm

file	The path to a graph file in CHACO/METIS format.
--preconfiguration	Determines the quality of the computed partitions by KaHIP [42]. We refer to the manual of KaHIP for detailed information. Default is *eco*.
--k	Number of partitions after removing the k-way node separator. Default is 2.
--imbalance	Desired balance of the partitions induced by the node separator. Default is 3%.
--nSeps	Number of computed separators with different seeds. Default is 5.

Nested Dissection Algorithm

file	The path to a graph file in CHACO/METIS format.
--preconfiguration	Determines the quality of the computed partitions by KaHIP [42]. We refer to the manual of KaHIP for detailed information. Default is *eco*.
--k	Number of partitions after removing the k-way node separator. Default is 3 on normal and 2 on *social* configurations.
--nSeps	Number of computed separators with different seeds. Default is 5.
--imbalance	Desired balance of the partitions induced by the node separator. Default is 20% on normal and 1% on social configurations.
--balance_multiplier	Corresponds to ξ in Section 5.2. Default is 3 on normal and 1.5 on *social* configurations.
--final_partition_size	Threshold for minimum degree ordering. Default is 80 on normal and 120 on *social* configurations.
--initial_order	If set to 1, a preordering is calculated and used for tie-breaking (see Section 10). Default is 0.
--cc_order	If set to 1, the connected components version is used. Default is 0.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[2] S. N. Bhatt and F. Thomson Leighton. A Framework for Solving VLSI Graph Layout Problems. *Journal of Computer and System Sciences*, 28(2):300–343, 1984.

[3] P. Bonsma. Most Balanced Minimum Cuts. *Discrete Applied Mathematics*, 158(4):261–276, 2010.

[4] T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[5] E. Cuthill and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference*, pages 157–172. ACM, 1969.

[6] P. Elias, A. Feinstein, and C. E. Shannon. A Note on the Maximum Flow Through a Network. *IRE Transactions on Information Theory*, 2(4):117–119, 1956.

[7] U. Feige, M. T. Hajiaghayi, and J. R. Lee. Improved Approximation Algorithms for Minimum Weight Vertex Separators. *SIAM Journal on Computing*, 38(2):629–657, 2008.

[8] L. R. Ford and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[9] J. Fukuyama. NP-Completeness of the Planar Separator Problems. *Journal of Graph Algorithms and Applications*, 10(2):317–328, 2006.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability*, volume 29. WH Freeman & Co., San Francisco, 2002.

[11] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-complete Problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM, 1974.

[12] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[13] A. George, M. Heath, J. W. H. Liu, and E. Ng. Solution of Sparse Positive Definite Systems on a Hypercube. *Journal of Computational and Applied Mathematics*, 27(1):129–156, 1989.

[14] A. George and J. W. H. Liu. The Evolution of the Minimum Degree Ordering Algorithm. *Siam Review*, 31(1):1–19, 1989.

[15] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum-Flow Problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

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

[17] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[18] L. Hyafil and R. Rivest. Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. Technical Report 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, 1973.

[19] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, page 29. ACM, 1995.

[20] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *International Conference on Parallel Processing*, pages 113–122, 1995.

[21] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[22] D. König. Graphok és Matrixok. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.

[23] P. S. Kumar and C. E. Madhavan. Minimal Vertex Separators of Chordal Graphs. *Discrete Applied Mathematics*, 89(1):155–168, 1998.

[24] C. E. Leiserson. Area-Efficient Graph Layouts. In *21st Annual Symposium on Foundations of Computer Science*, pages 270–281. IEEE, 1980.

[25] C. E. Leiserson and J. G. Lewis. Orderings for Parallel Sparse Symmetric Factorization. *Parallel Processing for Scientific Computing*, pages 27–31, 1989.

[26] R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[27] R. J. Lipton and R. E. Tarjan. Applications of a Planar Separator Theorem. *SIAM Journal On Computing*, 9(3):615–627, 1980.

[28] J. W. H. Liu. Modification of the Minimum-Degree Algorithm by Multiple Elimination. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):141–153, 1985.

[29] E. G. Ng and B. W. Peyton. Fast Implementation of the Minimum Local Fill Ordering Heuristic. In *CSC14: The Sixth SIAM Workshop on Combinatorial Scientific Computing*, 2014.

[30] V. Osipov, P. Sanders, and C. Schulz. Engineering Graph Partitioning Algorithms. In *Experimental Algorithms*, pages 18–26. Springer, 2012.

[31] F. Pellegrini. Scotch and libScotch 6.0 User's Guide. 2012.

[32] F. Pellegrini and J. Roman. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm For Static Mapping. In *TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I*. Citeseer, 1996.

[33] F. Pellegrini and J. Roman. Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[34] F. Pellegrini and J. Roman. Sparse Matrix Ordering with Scotch. In *High-Performance Computing and Networking*, pages 370–378. Springer, 1997.

[35] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering. *Concurrency: Practice and Experience*, 12(2-3):69–84, 2000.

[36] J. C. Picard and M. Queyranne. On the Structure of All Minimum Cuts in a Network and Applications. *Combinatorial Optimization II*, pages 8–16, 1980.

[37] A. Pothen and C. J. Fan. Computing the Block Triangular Form of a Sparse Matrix. *ACM Transactions on Mathematical Software (TOMS)*, 16(4):303–324, 1990.

[38] P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Algorithms–ESA 2011*, pages 469–480. Springer, 2011.

[39] P. Sanders and C. Schulz. High Quality Graph Partitioning. *Graph Partitioning and Graph Clustering*, 588:1, 2012.

[40] P. Sanders and C. Schulz. KaHIP v0.53–Karlsruhe High Quality Partitioning–User Guide. *arXiv preprint arXiv:1311.1714*, 2013.

[41] P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Experimental Algorithms*, pages 164–175. Springer, 2013.

[42] C. Schulz. *Hiqh Quality Graph Partititioning*. PhD thesis, Karlsruhe Institute of Technology, 2013.

[43] W. F. Tinney and J. W. Walker. Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[44] C. Walshaw. Website of the Walshaw Benchmark. http://staffweb.cms.gre.ac.uk/~wc06/partition/. Accessed: 2014-08-20.

[45] M. Yannakakis. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.