Ole Kröger

Engineering Minimum Degree Node Ordering Algorithms

Ruprecht-Karls-Universität Heidelberg

June 30, 2021

Supervisor: Prof. Dr. Christian Schulz
Co-Supervisors: Matthew Fahrbach, PhD
Wolfgang Ost, MSc
Prof. Darren Strash

Abstract

The problem of solving large sparse systems of linear equations is of great importance for a variety of different fields in scientific computing ranging from computational fluid dynamics to circuit simulation. In the process of solving those systems, the matrix normally gets decomposed into two matrices which have a triangular structure. For sparse matrices, this decomposition often results in triangular matrices that are denser than the initial matrix. This increases memory consumption and running time such that in some cases the initial system becomes intractable. Those additional non-zero entries in the matrix are called fill-in. The number of non-zeros can be reduced by permuting the original system before solving the linear system.

The reduction is often solved by representing the matrix as a graph. In the graph representation one decomposition step, called an elimination step, in the decomposition of the sparse matrix can be represented efficiently. The graph representation additionally helps to create algorithms benefiting from previously known graph algorithms to reduce the number of additional non-zero entries. In the graph, an elimination can be represented as removing a node and adding additional so-called fill-in edges in a way that previously adjacent nodes form a clique.

Finding a permutation that minimizes the number of fill-in edges is NP-hard. Therefore, different heuristics have been studied in the last couple of decades to reduce the number of non-zeros in a short amount of time. The most widely used heuristic is called the minimum degree algorithm, as it uses a node with minimum degree in the next elimination step.

We implemented a minimum degree ordering algorithm with a running time of $\mathcal{O}(nm)$ which was previously developed in a theoretical paper [R. Cummings et al., SODA 2021]. Additionally we combine the techniques of the minimum node degree algorithm with data reduction rules that were formerly used for nested dissection algorithms. The combination of the base algorithm and continuous reductions using indistinguishable nodes, simplicial nodes and element absorption speed up the currently implemented minimum degree algorithm in KaHIP by a factor of two in most instances.

Zusammenfassung

In vielen unterschiedlichen Bereichen der Naturwissenschaften müssen lineare Gleichungssysteme mit dünnbesetzten Matrizen gelöst werden. Die Einsatzfelder erstrecken sich über Bereiche wie Flüssigkeitsdynamik bis zur Schaltungssimulation. Lösungsverfahren dieser linearen Gleichungssysteme enthalten das Zerlegen der Matrix in Dreiecksmatrizen. Für dünnbesetzte Matrizen sorgt diese Zerlegung häufig dafür, dass die Dreiecksmatrizen nicht mehr so dünnbesetzt sind wie die Ausgangsmatrix. Die Nichtnullen, die dafür extra abgespeichert werden müssen, sind der Grund dafür, dass mehr Speicherplatz benötigt wird. Dieser Anstieg an benötigtem Arbeitsspeicher kann dazu führen, dass die Lösung mancher Gleichungssysteme unmöglich wird.

Eine Permutation der Anfangsmatrix kann dafür sorgen, dass die Anzahl an Nichtnullen verringert werden kann. Um eine Permutation zu finden, die die Anzahl an Nichtnullen reduziert, wird die Matrix in einen Graphen umgewandelt. In einem Graphen ist es möglichen einen Schritt einer Zerlegung der Matrix effizient darzustellen. Die Überführung in eine Graphenrepräsentation hat außerdem den Vorteil, dass bereits bekannte graphentheoretische Konzepte angewendet werden können um das Problem zu lösen. Ein Zerlegungsschritt kann im Graphen so repräsentiert werden, dass ein Knoten und benachbarte Kanten gelöscht werden können, sowie vorher benachbarte Knoten eine Clique bilden.

Im Allgemeinen ist es unlösbar in verfügbarer Zeit eine Permutation zu finden, die die Anzahl an neuer Nichtnullen minimiert. Deshalb wurden in den letzten Jahrzehnten Heuristiken entwickelt, um eine geeignete Permutation in kurzer Zeit zu finden. Eine der häufigsten Heuristiken heißt minimum degree ordering heuristic, dabei wird immer ein Knoten aus dem Graphen als nächstes ausgwählt und gelöscht, der die wenigsten Nachbarn hat.

Wir haben eine Heuristik dieser Art entwickelt, der eine Laufzeit von $\mathcal{O}(nm)$ hat. Der entsprechende Algorithmus wurde zuvor in einem theoretischen Paper beschrieben [R. Cummings et al., SODA 2021]. Des Weiteren haben wir Techniken von einer anderen Heuristik mit dem Namen Nested Dissection verwendet um den Algorithmus im praktischen Einsatz zu beschleunigen. Diese Techniken sind Datenreduktionen, die dabei helfen, dass entweder der Graph an sich kleiner wird oder Datenstrukturen in dem Algorithmus effizienter durchführbar sind. Diese Kombination sorgt dafür, dass der Ausgangsalgorithmus im Median 2.5 mal schneller läuft.

Acknowledgement

I am very thankful for the weekly meetings with Prof. Christian Schulz, Prof. Darren Strash, Dr. Matthew Fahrbach and Wolfgang Ost. In those they provided me with valuable insight in the field and advice throughout my work on the thesis. Additionally, I would like to thank the Hamilton College as they provided me access to their cluster to run my benchmarks. My friends provided me with moral support, motivation and humor during the hard times of the pandemic. Furthermore, I very much appreciate the support of my family and girlfriend throughout all my Master's degree.

Contents

1 Introduction

1.1 Background

In a variety of fields in scientific computing, including structural engineering, computational fluid dynamics and computer graphics, linear systems of the form

$$Ax = b \tag{1}$$

need to be solved [9].

Let A be an $n \times n$ matrix. In those types of applications A is often a sparse matrix that has on the order of $\Theta(n)$ or $\Theta(n \log n)$ non-zeros in contrast to dense matrices which have $\Theta(n^2)$ non-zero entries. The algorithms discussed in this thesis are intended for sparse matrices, therefore A is in the following considered to be sparse. Further restrictions on the matrix A will be mentioned in the relevant sections.

Solving equations like (1) is often done by decomposing the sparse matrix A into parts that can be solved with fewer computations. Decomposition creates two or more matrices which have a defined structure i.e., are lower or upper triangular matrices. As an example, the Cholesky decomposition factorizes a symmetric positive definite matrix A into $A = LL^\top$ where L is a lower triangular matrix. The linear system $Ax = b$ can then be solved using $Ly = b$ for y and $L^\top x = y$ for x. Those two linear systems can be solved very efficiently by forward and back substitution.

Decomposing the matrix itself often has the disadvantage of making the decomposed matrices denser than the original matrix A. For sparse matrices only the non-zero values and their positions are stored to reduce the memory consumption. The additional non-zero elements in the decomposed matrices need to be stored which has the downside of having a higher space complexity and therefore memory usage. This often increases the running time of solving the linear system or makes it impossible due to memory constraints.

Non-zeros that are added during the process of decomposition are referred to as *fill-in*. Reducing the fill-in is required to solve some sparse linear system as the fill-in can outgrow the available memory capacity. The aim of so-called *node ordering algorithms* is to limit the required fill-in as much as possible without exceeding time constraints.

Node ordering algorithms reorder the equation and therefore matrix A to accomplish

this goal. This reordering can be done for symmetric matrices by using a permutation matrix P with the equation $PAP^\top$. Multiplying A from the left with the permutation matrix and from the right with its transpose has the effect of reordering the rows and columns of A. The system $Ax = b$ can then be written as $(PAP^\top)(Px) = (Pb)$.

The problem of finding a permutation matrix to minimize the fill-in during the decomposing step is named the minimum fill-in problem. As it is proven to be NP-hard [31], practical algorithms use heuristics to compute an ordering with small fill-in in a short amount of time. Solving this problem is done by converting the linear algebra problem into a problem in the field of graph theory.

For this transformation, the symmetric matrix A is represented as an undirected graph G with n vertices where n is the number of rows/columns in A. Two nodes i and j are adjacent in G iff $A_{ij} \neq 0$. One step in calculating a decomposition is a reduction of the matrix by one row and column. It involves additions of rows with the result of having a the first column being all zeros besides the first element. This results in a reduction of the matrix size by 1 and is iterated recursively until the whole matrix is decomposed. Transforming the matrix into a graph problem has the effect that such an elimination step can be represented as forming a clique of the neighborhood of the corresponding node in the graph. This relation between the matrix and the creation of a clique can be seen when we apply the matrix operation of eliminating a node to the graph representation. This is due to the fact that eliminating a row in the matrix is done by the addition of multiples of rows to obtain the triangular matrix structure. In the graph representation removing a row first of all removes the node corresponding to that row as well as the edges. However due to the addition of rows, the entries for which the elimination row was non-zero, can introduce additional non-zero entries in rows for which the addition occurs. Those rows are concretely the rows for which the first entry is not already zero. In terms of the graph representation these are exactly the edges that form a clique of previous neighbors of the elimination node. In the graph this operation results in a size reduction by one node as the node itself and all connected edges are deleted. However the number of edges can increase during this process depending on how many neighbors the elimination node had but more precisely dependent on the number of missing edges that need to be created to form the clique. The number of newly created edges that form a clique of the previous neighborhood gives then an upper bound for the fill-in that we want to reduce.

1.2 Contribution

We introduce an implementation of a minimum node degree algorithm with a running time $\mathcal{O}(nm)$ using a hypergraph data structure. Additionally we formulate data reduction rules that are applied in each node ordering step and reduce the running time

of the algorithm often by more than a factor of 2.5. The data reduction rules do not change the fill-in compared to the minimum node degree algorithm. Furthermore we benchmark our algorithm against a minimum node degree algorithm that is currently part of the KaHIP software package, and compare it against different node ordering approaches like nested dissection and approximate minimum node degree ordering. Those algorithms have been developed over the years and are used in practice to solve the minimum degree ordering problem.

1.3 Structure of the Thesis

In the next chapter the necessary fundamentals of graph theory are laid out. Additionally, the general idea for the minimum degree ordering algorithm will be explained in more detail. In Chapter 3 we give an overview over a variety of different algorithms to solve the minimum degree ordering problem. Chapter 4 then shows the methods we use to implement a fast minimum node degree algorithm as well as the data reduction rules we apply to improve the running time even further. Afterwards in Chapter 5 the implementation details are described in more detail. In Chapter 6 we first compare our implementation to a naïve implementation of the minimum node degree algorithm. Afterwards we discuss the effects of the applied data reduction rules with respect to the running time and the number of fill-in edges. Additionally, we compare our implementation to different heuristics to solve the node ordering problem. Chapter 7 sums up the techniques we used and discusses possible future improvements.

2 Fundamentals

This chapter introduces the fundamentals of graph theory needed for our minimum degree algorithm. In the subsections the general concept of reordering nodes are discussed as well as the idea of the minimum degree ordering algorithm.

An undirected graph $G = (V, E)$ is defined by a set of nodes V and a set of undirected edges E. Each edge $e \in E$ is a set of two nodes $\{v, w\}$ with $v, w \in V$. If $\{v, w\} \in E$, v and w are said to be *adjacent* or *neighbors*. We refer to the number of nodes $|V|$ with n and the number of edges $|E|$ is denoted as m. The *union* of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the union of the vertices and edges: $G_1 \cup G_2 := (V_1 \cup V_2, E_1 \cup E_2)$. We define the (open) *neighborhood* $N_G(v)$ of a node v as the set of all nodes that are connected to v via an edge. It is defined as $N_G(v) := \{w \in V \mid \{v, w\} \in E\}$. Additionally, we define the *closed neighborhood* as $N_G[v] := N_G(v) \cup \{v\}$ which contains the node v itself. The *degree* of a node v is given by $\deg_G(v) := |N_G(v)|$ as the number of adjacent nodes.

In the following the subscript G is omitted when corresponding graph is clear from the context. We extend the definition for the neighborhood and degree for sets of nodes $A \subseteq V$ such that, i.e., $N_G(A) := (\cup_{v \in A} N_G(A)) \setminus A$.

We define $E(A)$ for $A \subseteq V$ as the subset of edges in E where both endpoints are in A: $E(A) := \{\{v, w\} \in E \mid v, w \in A\}$. The subgraph $G[A] := (A, E(A))$ is called the *induced subgraph* of A. A *complete* graph K is a graph G in which every node is adjacent to every other node. Furthermore, a set of nodes $A \subseteq V$ is called a *clique* if the induced subgraph $G[A]$ is complete and we define $K_A := V(G[A])$. A clique is called *maximal* iff no vertex can be added to the clique while still forming a clique.

In our algorithm we extensively use the idea of storing only the nodes of a clique instead of all present edges. For this we use a *hypergraph* which is defined as $H = (V, C)$ with a set of nodes V as for the standard undirected graph and a set C of undirected hyperedges. A *hyperedge* is an extension of the edge definition and can contain more than two vertices in the set. The hyperedges C form a subset of the *power set* without the empty set of V, whereas the power set is defined as the set of all subsets of V including V as well as the empty set. For our use case each hyperedge represents a clique. Additionally we can use the fact that an edge defines a clique of size 2 and can therefore be represented as a hyperedge of size 2.

The above mentioned definitions are visualized in Figure 2.1 which shows an undi-

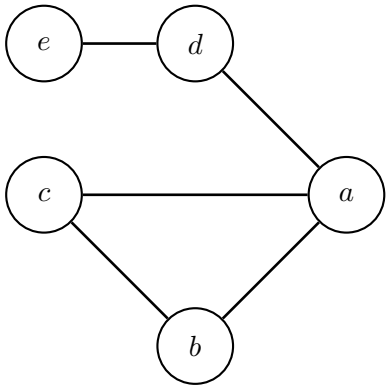

Fig. 2.1: An undirected graph with three maximal cliques $\{b, d\}$, $\{d, e\}$ and $\{a, b, c\}$.

rected graph $G = (V, E)$. As nodes we have $V = \{a, b, c, d, e\}$ with $n = 5$ as well as the edges $E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{d, e\}\}$ and $m = 5$. The open neighborhood of a is $N(a) = \{b, c, d\}$ and the closed one is $N[a] = \{a, b, c, d\}$. Furthermore is the neighborhood of the two nodes a and e: $N(\{a, e\}) = \{b, c, d\}$. The degree of node d is $\deg(d) = 2$. Let $A = \{a, b, c\}$ and the induced subgraph $G[A] = (\{a, b, c\}, \{\{a, b\}, \{a, c\}, \{b, c\}\})$. In this subgraph every node is adjacent to every other node such that $G[A]$ is a complete graph and A is a clique. The graph G can be stored as a hypergraph $H = (V, C)$ with the hyperedges $C = \{\{b, d\}, \{d, e\}, \{a, b, c\}\}$ which here uses the maximal cliques of the graph as the hyperedges.

2.1 Node Ordering

This section explains the importance and general procedure of node ordering using a small example. A symmetric matrix A can be *factorized* or *decomposed* using a variety of decomposition schemes like LU and Cholesky decomposition. In the LU factorization the matrix A can be written as $A = LU$ where L and U are two triangular matrices more precisely L is a lower triangular matrix and U an upper triangular matrix.

The LU decomposition is known as the matrix form of Gaussian elimination which will later be used in an explanatory example.
A decomposition reformulates the matrix A into two or more matrices of a certain structure that make it easier to solve linear equations of the form $Ax = b$. In the LU decomposition the two matrices are triangular matrices. The linear system can then be written as $L(Ux) = b$ and can be solved more efficiently due to fewer additions and multiplications by solving $Ux = L^{-1}b$.

It is important to note that the symmetric matrix A can be reordered by swapping rows and the associated columns and doing the same for x and b such that the equation is still fulfilled. This is done by a permutation matrix P such that $(PAP^\top)(Px) = (Pb)$

can be solved. The *associated* column of a row is the column with the same index such that swapping two rows and their associated columns maintains the symmetry of the matrix.

For sparse matrices the LU decomposition can create dense triangular matrices L and U such that the matrices take more space than the initial matrix A. In some cases storing the triangular matrices consumes more space than the available memory, which makes the linear equation practically unsolvable. This procedure can then increase the space complexity to n^2 where n is the number of rows/columns. The number of extra non-zero entries is referred to as the *fill-in*.

There exist a variety of different algorithms to reduce the fill-in which will be discussed in Chapter 3. In this section we lay out how transforming the problem into a graph problem can help to reduce the fill-in.

As an example we will have a look at a matrix and its decomposition to show the amount of fill-in. The decomposition of a small matrix is shown in (2). It shows one particular example where the lower and upper triangular matrices are completely filled. In this case only a single elimination step is needed to decompose the matrix.

$$\begin{bmatrix} 1 & 2 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 1 & 0.\overline{6} & 1 & 0 & 0 \\ 1 & 0.\overline{6} & 0.25 & 1 & 0 \\ 1 & 0.\overline{6} & 0.25 & 0.2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 1 & 1 \\ 0 & -3 & -2 & -2 & -2 \\ 0 & 0 & 1.\overline{3} & 0.\overline{3} & 0.\overline{3} \\ 0 & 0 & 0 & 1.25 & 0.25 \\ 0 & 0 & 0 & 0 & 1.2 \end{bmatrix} \tag{2}$$

We have $A = LU$ with A as a 5×5 symmetric matrix and after applying the LU decomposition both triangular matrices L and U are dense and together have a space complexity of $\Theta(n^2)$ whereas the space complexity of A was $\Theta(n)$. The general example of this would be a diagonal matrix with only the first row and column filled.

By reordering the rows and columns a minimum amount of fill-in can be achieved, which can be seen in (3).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 2 \\ 1 & 1 & 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & -6 \end{bmatrix} \tag{3}$$

The matrix shown in (2) can be created as a graph by having five nodes. Each node corresponds to a row/column. We can create these undirected graphs as we only consider symmetric matrices. The xth node represents the xth row/column. Each non-zero and

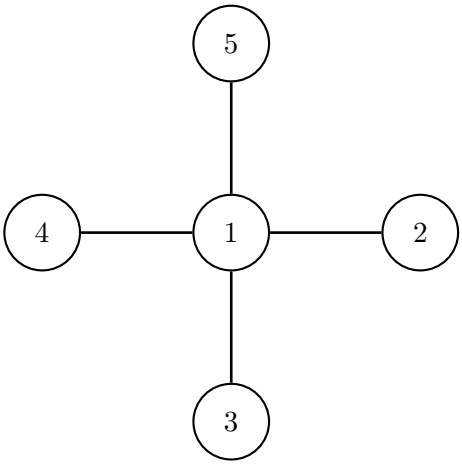

Fig. 2.2: Graph representation of the matrix in (2).

non-diagonal entry represents an edge between the two corresponding nodes.

Figure 2.2 shows the graph representation of the initial matrix as shown in (2). The elimination step shown in 2 can be represented as the removal of node 1 in Figure 2.2. Nodes that were neighbors of the removed node, in this case all nodes 2-5, need to be connected to form a clique. This process is referred to as *eliminating* node 1 from the graph.

We need to add a clique for the following reason: Each Gaussian elimination step performs an operation like $R_v = R_v + yR_u$ where R_u represents the row of vertex u and $y \in \mathbb{R}$ being a factor. In this example $u = 1$ and $v = 2, \ldots, 5$. For every v this will potentially create non-zero elements in the matrix which are represented as edges in our graph.

In the graph representation a new edge is added for each of those potential non-zero values as well as all edges adjacent to u and u itself are removed. For the example shown this means that the nodes $2, \ldots, 5$ form a clique after we eliminated node u. This adds the edges $\{2,3\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,5\}$ in our graph and removes edges $\{1,2\}, \{1,3\}, \{1,4\}, \{1,5\}$.

During this process as shown in Figure 2.3 we have added those six new edges to the graph which represents the additional fill-in of this step.

Afterwards we remove all other nodes to finalize the decomposition. The completion of this however does not add more fill-in edges in our example as after eliminating node 1, the remaining graph, also called elimination graph, is already a complete graph.

In the reordering example given in (3) however the order of the initial nodes is permuted such that no new fill-in edges are generated. In that case the last row and column are filled instead of the first row and column. This can be seen in Figure 2.4 by removing the nodes in ascending order. After each elimination step the next node in ascending order has at most a single neighbor. This means the node and edge are

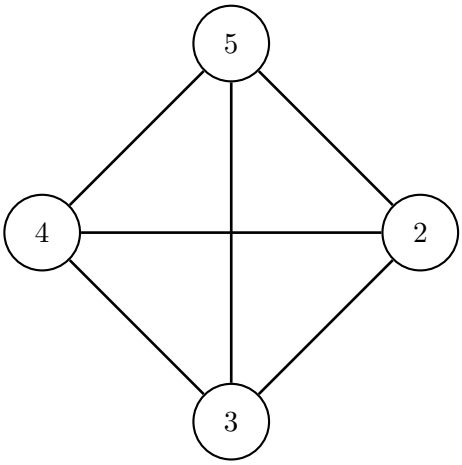

Fig. 2.3: Graph representation of the matrix in (2) after eliminating node 1

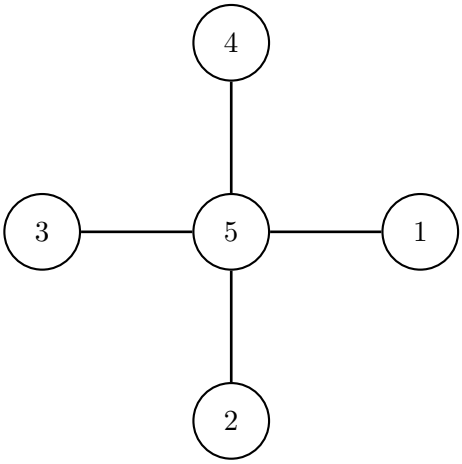

Fig. 2.4: Graph representation of matrix in (3).

simply removed and no new clique has to be formed, such that no additional edges need to be created. Node 5 in the end has no adjacent nodes such that only the node itself will be removed from the graph to finish the node ordering routine.

After reordering the nodes the fill-in in this case would be optimal as no fill-in edges are created which means that no non-zero entries are created in the process of the factorization of the reordered matrix.

Formally we introduce some more terms to define an ordering of the graph. This includes the definition of the graphs that are created after a node has been eliminated. We use the following notation for elimination graphs $G_v := (V_v, E_v)$ with $V_v = V \setminus \{v\}$ and $E_v = \{\{a, b\} \in E \mid a, b \in V \setminus \{v\}\} \cup E(K_{N_G(v)})$ as introduced by Rose [23]. This graph can be constructed by creating the fill-in edges, as the edges forming a clique of the adjacent nodes of v, and then removing v as well as the edges incident to v. The ordering in which we construct those elimination graphs is a bijection $\sigma : \{1, 2, \ldots, n\} \to V$ which

represents the sequence of elimination graphs $G^{(1)}, G^{(2)}, \ldots, G^{(n)}$. The ordering σ of nodes is often written as the sequence $\sigma = x_1 \cdots x_n$ such that x_1 is eliminated first, then x_2 up to x_n. The i-th elimination graph is defined as $G^{(i)} := (G^{(i-1)})_{\sigma(i)}$ for $i = 1, \ldots, n$ and $G^{(0)} := G$. In general the superscript specifies the number of the elimination graph and the subscript is used for specifying the node that gets eliminated. The elimination graph $G^{(n)}$ is the empty graph as all nodes have been eliminated.

Additionally, we define some functions to compute the number of fill-in edges of an ordering and define the fill-in of a minimum fill-in ordering in which the least amount of fill-in edges are created. We start with the *deficiency* $D_G(x)$ of a node x in G. It describes the fill-in edges that would be introduced when eliminating v from G: $D_G(x) := \{\{u, v\} \mid u, v \in N_G(x), u \neq v, u \notin N_G(v)\}$ which are the edges that extend the graph $G[N_G(x)]$ to form the clique $K_{N_G(x)}$. The introduced fill-in by eliminating x is then $|D_G(x)|$. Using the definition of an ordering we can define $\phi(G, \sigma) := \sum_{i=1}^{n} |D_{G^{(i-1)}}(\sigma(i))|$. Then $\Sigma(G) := \arg\min_\sigma \{\phi(G, \sigma)\}$ is a minimum fill-in ordering. We try to approximate this using a heuristic due to the fact that the problem of finding a minimum fill-in ordering is NP-complete [31].

We use a special notation for the minimum fill-in of the optimal ordering $\Sigma(G)$ as $\Phi(G) = |\Sigma(G)|$. We note that the fill-in $\Phi(G) \geq \Phi(G^{(1)}) \geq \cdots \geq \Phi(G^{(n-1)}) \geq \Phi(G^{(n)}) = 0$ as the fill-in can never decrease when eliminating a node.

2.2 Minimum Node Degree Ordering

There exist many different strategies to find a minimum degree ordering as well as heuristics to approximate a minimum degree ordering [4, 16]. In general, finding a node ordering that minimizes the number of fill-in edges is a NP-hard problem, therefore most algorithms focus on heuristic approaches to reduce the number of fill-in edges while computing an ordering much faster than an exact algorithm.

We focus on the most famous of those heuristics, called minimum node degree ordering. This algorithm is later used as the base of our algorithm. In this section we outline the idea of the algorithm. In general performing a single elimination step removes one node v and creates $\mathcal{O}(\deg(v)^2)$ fill-in edges by creating a clique of adjacent vertices. This is due to the fact that the previous adjacent nodes of v form a clique in the process and the highest number of fill-in edges must be created when v has a high degree and those neighboring nodes are not adjacent before the elimination step. Formally this will be the case if $\deg(v)^2 = |D_G(x)|$.

Therefore, the fill-in created in a single step can be limited by taking a vertex v of G with minimum degree. One has to note that the minimum degree itself is not the only factor as it only provides an upper limit to the deficiency of the node. The deficiency

Algorithm 1: Basic minimum node degree ordering algorithm

Result: The minimum node degree ordering
$ordering :=<>$
$G := (V, E)$
while $G \neq (\emptyset, \emptyset)$ **do**
  $v := \underset{u \in V}{\operatorname{argmin}} \deg(\mathrm{u})$
  $G := G_v$
  $ordering = ordering + v$
end

of another node can be smaller if the neighboring nodes already form a clique or only a few edges are needed to form a clique. This means that minimum node degree ordering is only greedy based on the degree of the nodes and not on the deficiency. The degree of a neighbor of v can change in the process of the elimination step therefore the degree needs to be updated in each step.

Algorithm 1 gives the basic pseudocode of this algorithm using the notion of the elimination graph.

The algorithm is initiated with an empty ordering and the elimination graph G is initialized as the symmetric graph given as an input. As long as not all nodes are ordered, there are still nodes in the elimination graph. This means we take one node v with minimum node degree in the elimination graph and update G as the elimination graph after v has removed. The last step is to add v to the ordering. With this ordering the permutation matrix P can be constructed.

Two main decisions in Algorithm 1 are abstracted in the pseudocode. These decisions are first obtaining a node with minimum degree and secondly how to create the elimination graph most efficiently. This will be discussed in Chapter 4 as well as techniques to reduce the graph size. These decisions are the main part of this thesis as the data structures used are the most important part of developing a fast version of this algorithm.

The degree of the vertices must be updated inside the loop as the removal of v can change the degree of $u \in N(v)$. For all other vertices, however, the degree stays the same. One can further observe that the following equation holds for the neighboring vertices u: $\deg_G(u) + \deg_G(v) - 1 \geq \deg_{G_v}(u) \geq \deg_G(u) - 1$. This holds as we only delete one incident edge of u namely (v, u) and can create up to $\deg_G(v) - 1$ additional incident edges when creating the fill-in edges.

Besides how the node with minimum degree is chosen as well as how the elimination graph is computed, Algorithm 1 is not fully specified as nodes with the same minimum degree can exist as shown in Figure 2.2. In that graph the nodes $\{2, 3, 4, 5\}$ all have degree one. An algorithm can break those ties in a deterministic way, such as always

choosing the lexicographically first, but most algorithms break the tie arbitrarily. This saves computing costs for ordering the nodes and can help give a lower fill-in when the algorithm is applied several times to the initial graph. This can be suitable for instances that can be ordered faster than a given time limit. Breaking the ties arbitrarily results in an nondeterministic algorithm as the resulting ordering as well as the fill-in can vary.

2.3 Data Reductions

The aforementioned algorithm can greatly benefit from *data reduction rules*. Those rules reformulate the problem by reducing the graph size while being able to extract the ordering of initial graph from the reduced ordering. The idea is to improve the running time, which depends on the graph size by reducing the number of nodes and edges, then apply the minimum degree ordering algorithm and afterwards apply a transformation to the solution on the reduced graph to obtain the solution of the original problem.

If the transformations of the graph and the resulting solution can be performed faster than the original algorithm needs to solve the extra size, the reductions improve the running time. Besides applying the reductions only for the initial graph we apply reductions also in each ordering step such that we use the data reduction rules n times. The details on which rules we apply are covered in Section 4.2 and the benefit of these reductions is shown in Section 6.2.

3 Related Work

In this section we discuss a variety of different algorithms that have been used to reduce the number of fill-in edges. They all apply a permutation matrix to A to reorder the symmetric sparse matrix but use varying algorithms to improve running time and reducing the number of fill-in edges. We give a broad overview of those algorithms while giving more detail on algorithms that we later use for comparison.

The problem of finding a permutation matrix that minimizes the fill-in was proven to be NP-complete by Yannakakis et al. [31] in 1981. Nevertheless, algorithms have been developed throughout the years to compute the exact solutions as for example by Fomin et al. [12] in 2008. They have shown that the minimum fill-in can be computed in $\mathcal{O}(1.8899^n)$ time, which is not practical for real-world instances but gives a good idea of the complexity of the problem. Some special algorithms have been developed over the years to find the perfect elimination ordering, if one exists. A perfect elimination ordering means an ordering that has no fill-in edges. As an example finding an ordering of this form, if one exists can be done in $\mathcal{O}(n+m)$ time as described by Rose et al. [24]. Furthermore, graphs that do not have a perfect elimination ordering but have a limited maximum degree can be solved efficiently in $\mathcal{O}(n(\Delta^3 + \alpha(n)))$ time where Δ is the maximum degree and α is the inverse Ackermann function. This was shown by Dahlhaus in 2002 [8].

In contrast to finding the best node ordering many algorithms provide heuristics to decrease the number of fill-in edges in a short amount of time without proving optimality or giving optimality bounds. The most famous of those heuristics is the minimum node degree algorithm developed by Tinney and Walker [30] in 1967. It is the symmetric analog of a method developed by Markowitz [20] ten years earlier. A graph-theoretical model of this algorithm was later described by Rose [23] who gave it the name: the minimum degree ordering algorithm. The name comes from the idea of always eliminating a node with minimum degree next, to limit the number of fill-in edges in the current step. A thorough description and improvements over the years was given by George and Liu in 1989 [17].

The minimum degree algorithm was further improved by George and Liu [16] using a different data structure called quotient graphs as a representation. The quotient graph is a representation that does not exceed the amount of storage used for the original graph. Some other papers refer to it as the generalized element model [29].

Similar to the hyperedge model that we will describe further in Section 4.1, it does not store the list of all edges to represent a clique but instead just the nodes that form the clique as a hyperedge. The representation works with two kinds of nodes. The nodes which have not been eliminated yet are called variables of the graph and nodes which have already been eliminated are referred to as either elements or simply eliminated nodes. We follow the notation described by Amestoy et al. [4] to further explain the quotient graph representation. The state of the graph after k eliminations is described as $G^k = (V^k \cup \overline{V}^k, E^k \cup \overline{E}^k)$ with $G^0 = G$. V^k represents the variables after k eliminations and E^k the edges between them. $\overline{V}^k$ are the elements of the quotient graph at that stage and $\overline{E}^k$ the edges between variables and elements. No edges between two elements exist and for G^0 both sets $\overline{V}^k$ and $\overline{E}^k$ are initialized as empty sets. Furthermore three more sets are defined to compute the adjacent nodes after an elimination step. $\mathcal{A}_i = \{j \mid (i,j) \in E\}$ as the set of variables adjacent to variable i, $\mathcal{E}_i = \{e \mid (i,e) \in \overline{E}\}$ as the set of elements adjacent to variable i which is referred to as the element list i. The third set is the set of all variables adjacent to element e formally defined as $\mathcal{L}_e = \{i \mid (i,e) \in \overline{E}\}$. The neighbors of variable i can then be computed with: $N_G(i) = \mathcal{A}_i \cup \left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) \setminus \{i\}$. This equation can be used to compute $\mathcal{L}_p$, after one elimination step in which p is the variable that was selected as the kth pivot. Variable p gets then removed from the variables V and added to the elements $\overline{V}$. It follows $\mathcal{L}_p = N_G(p)$. The notation of $\mathcal{L}$ is inspired by the triangular matrix L in the Cholesky factorization as the following holds: If we have variable p as the kth pivot then for a variable i which becomes the pivot at $m > k$ the entry L_{mk} will be non-zero. After each elimination step we can potentially remove elements from our representation as all elements adjacent to variable p can be absorbed due to $\mathcal{L}_e \subseteq \mathcal{L}_p$. The paper by Amestoy et al. [4] further investigates how the structure can be used to detect indistinguishable nodes which is a data reduction rule that we will discuss in Section 4.2. For the next part we just need the following result of that discussion. Several nodes can be combined into one supervariable during the process of indistinguishable node reduction. They denote the set of several variables into a supervariable with principal i as $\mathcal{V}_i$ and initialize the supervariable as $\mathcal{V}_i = \{i\}$. If then a variable p is selected in the elimination ordering all variables in $\mathcal{V}_p$ are eliminated. Furthermore, they use an external degree formulation

$$d_i = |N_G(i) \setminus \mathcal{V}_i| = |\mathcal{A}_i \setminus \mathcal{V}_i| + |(\cup_{e \in \mathcal{E}_i} \mathcal{L}_e) \setminus \mathcal{V}_i| \,. \tag{4}$$

The external degree is the degree of the node based on adjacent nodes which are neither indistinguishable nor otherwise absorbed due to data reduction rules. It is mentioned that using the external degree provides generally better results than using the true

degree of the node when the ordering scheme is applied.

In contrast to the quotient graph representation, other representations have been introduced to increase the performance of standard minimum degree ordering. In this thesis we work with a data structure introduced by Cummings et al. [7] who described an algorithm that finds a minimum node degree ordering in $\mathcal{O}(nm)$ time and uses a hyperedge representation of the elimination graph.

Nevertheless, it is still the case that most of the running time for these algorithms is spent on computing and updating the node degree. Therefore Amestoy et al. [4] constructed an algorithm in 1996 that approximates the node degrees to decrease running time while trying to maintain the idea of the minimum degree ordering approach. They do this by using the quotient graph representation and a computation to provide an upper bound on the node degrees. This upper bound can be calculated faster than computing the exact degree such that larger graphs can be ordered in a given time frame. This technique of approximate node degrees was further improved by Fahrbach et al. [11] who described a nearly-linear time algorithm for an approximate greedy minimum node degree algorithm.

Another heuristic that is often used for larger graphs is called nested dissection. It was first described by George [14] to calculate orderings for regular finite element meshes. It recursively finds separators to partition the graph until a certain threshold of number of nodes for a partition is reached. A separator divides the graph into parts such that the parts are connected components that do not share an edge with another part. The smaller graphs are then ordered using methods like the above mentioned minimum node degree algorithm. Orderings of the smaller graphs are then combined to provide a reasonable ordering for the original graph. This is done by assigning the last ordering labels to the separator itself and the remaining partitions are ordered one after another as a block. As an example if a graph G is given with n vertices and it gets partitioned as S, A, B with S being the separator then the ordering is as follows: The labels from $n-|S|+1$ to n are given for the separator such that they are eliminated at the very end. The part A is given the range from 1 to $|A|$ and for B from $|A|+1$ to $n-|S|$. This can be easily extended for more than two partitions. The approach of nested dissection was later extended by George and Liu [15] in 1978 to work on general graphs and not just regular finite element meshes. Nested dissection was further improved over the years [5, 18]. The subproblem of finding small node separators in large graphs is a highly studied field on its own [6, 26, 27, 28].

Nested dissection often uses data reduction rules to scale down the number of nodes in each partition. Generally many graph algorithms benefit from data reduction rules [3]. An extensive list of data reduction rules that are used for nested dissection are formalized by Ost et al. [21]. Those rules are split into exact and inexact rules whereas exact rules do not increase the number of fill-in edges. It includes the indistinguishable and

simplicial reduction rules that are further explained in Section 4.2 as well twin reduction and path compression. Simplicial nodes are nodes whose neighborhood is already a clique and can therefore be eliminated directly. Indistinguishable nodes and twins are nodes with equal closed or open neighborhoods which can be contracted. If any of those nodes is eliminated the other become simplicial and can therefore be eliminated immediately afterwards. Paths with degree 2 can be replaced with a single degree 2 node and if the node with degree 2 is eliminated the degree-2 neighbors can be eliminated next.

The two inexact reduction rules are degree-2 elimination and triangle contraction. The former eliminates nodes of degree two, which becomes exact if none of the eliminated nodes are separators that can be exploited in nested dissection algorithms. They also mention the inexact triangle reduction which contracts adjacent nodes of degree 3 that share at least one neighbor.

4 Methods

In this chapter we discuss the methods used for the development of our minimum node degree algorithm. The groundwork of this is the *fast minimum degree algorithm* by Cummings et al. [7] where the theory for this algorithm is developed. Section 4.2 discusses several reductions that we use to speed up the algorithm. These reduction steps are applied in each ordering step and either reduce the elimination graph itself or simplify the data structures to decrease running time.

4.1 Minimum Degree Algorithm

The algorithm as described by Cummings et al. [7] is the base algorithm for our implementation and computes the minimum node degree ordering in $\mathcal{O}(nm)$ time.

The *fast minimum degree algorithm* uses a different representation of the elimination graphs as described in Section 2.2. It uses hyperedges as a representation for the cliques of the elimination graph and uses the fact that a single edge can be seen as a clique of two nodes.

An elimination graph $G_v = (V_v, E_v)$, which represents the graph G after eliminating vertex v, can be defined as a list of hyperedges $U_1, U_2, \ldots, U_k \subseteq V_v$. The complete graphs K_{U_i} of these hyperedges form the elimination graph $G_v = K_{U_1} \cup K_{U_2} \cup \cdots \cup K_{U_k}$ where K_{U_i} is the clique that is formed by the vertices in hyperedge U_i. Furthermore, we define $U_{v_1}, \ldots, U_{v_i}$ as the hyperedges that include vertex v.

We initialize the algorithm with the edges $e \in E$ of G as a list of hyperedges with only the two endpoints u and v of e. After one elimination step of vertex v the elimination graph can be constructed by removing $U_{v_1}, U_{v_2}, \ldots, U_{v_i}$ which are the hyperedges that include the vertex v and add a single new hyperedge W. The hyperedge W represents the clique that includes all the previous adjacent vertices of v and can be formulated as $W = (U_{v_1}, U_{v_2}, \ldots, U_{v_i}) \setminus \{v\}$.

The algorithm maintains the representation of the hyperedges as well as an hash table-based adjacency list for the edges. This is in contrast to the main data structure used by Cummings et al. [7] as they use an adjacency matrix to simplify the proof of the runtime complexity of $\mathcal{O}(nm)$. We use the adjacency list for practical reasons of reducing the space complexity of the algorithm. In Algorithm 2 we give the high level pseudocode for the algorithm as stated by Cummings et al. [7] as Algorithm 1 with

Algorithm 2: Minimum node degree ordering algorithm as described in [7].

```
Result: The minimum node degree ordering
Initialize the elimination graph as the adjacency list of G
ordering :=<>
hyperedges := ∅
Mark all vertices as active
for {u, v} ∈ E do
    Add {u, v} to the list of hyperedges
end
for i = 1 to n do
    Let v be an active vertex with minimum degree in the elimination graph
    Deactivate v
    push v to ordering
    W := ∅ (the new hyperedge)
    foreach hyperedge U that contains v do
        Remove U from the list of hyperedges
        U := U \ {v}
        X := W \ {U}
        Y := U \ {W}
        foreach pair {x, y} in X × Y do
            Add edge {x, y} to the elimination graph if it does not exist yet
        end
        foreach vertex w ∈ Y do
            Remove edge {v, w} from the elimination graph
        end
        W := W ∪ Y
    end
    push W to the list of hyperedges
end
```

minor changes in notation to resemble the one given in Section 2.

A vertex is *active* as long as it is a vertex in the elimination graph and gets deactivated after the vertex is removed from the elimination graph. In the following theorem it is shown that the algorithm creates a minimum node degree ordering. It was stated and proven in Lemma 3.1 in the paper by Cummings et al. [7] and changed here based on our different notation.

Theorem 1. *Algorithm 2 creates a minimum node degree ordering. Additionally let* $G = G^{(0)}, G^{(1)}, \ldots, G^{(n)}$ *be the sequence of elimination graphs. After each iteration of* $i \in \{1, \ldots, n\}$ *it holds that:*

1. *The elimination graph variable represents the state of* $G^{(i)}$.
2. *The hyperedges* $U_1, \ldots, U_k$ *satisfy* $K_{U_1} \cup \cdots \cup K_{U_k} = G^{(i)}$.

Proof. We prove that the algorithm creates a valid minimum node degree ordering by induction and show that the specified invariants hold in each iteration. At the start of the algorithm (before line 8) it holds that the elimination graph $G^{(0)} = G$ and the union of the hyperedges represent $G^{(0)}$. This is the base case of our induction therefore for the rest we assume we are currently at iteration $i \in \{1, \dots, n\}$ of the algorithm and the ordering represents a minimum node degree ordering up to this point.

The algorithm selects a node v with minimum degree in line 9, removes it from the list of active nodes and updates the ordering accordingly. Eliminating the vertex v should remove all edges $\{u, v\} \in E_{G^{(i-1)}}$ and add edges such that $N_{G^{(i-1)}}(v)$ forms a clique. Let $U_1, U_2, \dots, U_\ell \subseteq V$ be the hyperedges that include v. After one elimination step (after line 25) we have $W = (U_1 \cup U_2 \cup \cdots \cup U_\ell) \setminus \{v\}$ as we have $W := W \cup Y_k$ with $Y_k = (U_k \setminus \{v\}) \setminus W$, $\forall k \in \{1, \dots, \ell\}$.

Each hyperedge $U_1, U_2, \dots, U_\ell$ is removed from the representation in line 14 and W is added in line 26. Therefore, $W = N_{G^{(i-1)}}(v)$ such that the new list of hyperedges forms $G^{(i)}$. Next we show that the adjacency list representation of the elimination graph is updated correctly. Using our induction hypothesis we know that the representation holds for $G^{(i-1)}$ at the beginning of the loop starting in line 8 for i. Based on the hypothesis we also know that the hyperedge representation of the graph is correct such that for each hyperedge U that includes v the edges of the clique K_U are part of our adjacency list. Lines 18-20 add edges that are in K_W after line 24 to update the adjacency list accordingly. Additionally the edges to neighbors of v are removed in lines 21-23. Assume we currently process $U = U_k$ such that $X = (U_1 \cup \cdots \cup U_{k-1}) \setminus U_k$ and $Y = U_k \setminus (U_1 \cup \cdots \cup U_{k-1})$ after line 17. It holds that only edges $(x, y) \in X \times Y$ can be missing from the elimination graph which are added in lines 18-20. This proves that the adjacency list represents the elimination graph $G^{(i)}$ such that the invariants of the induction step hold and therefore completes the proof. □

The implementation details of the algorithm are important to achieve the running time of $\mathcal{O}(nm)$. These are discussed in Chapter 5.

4.2 Data Reductions

Minimum node degree algorithms can be sped up by reducing the size of the graph G and form G' in such a way that the number of vertices $n = |V|$ decreases. The reduced graph G' is often equivalent to G in the sense that the minimum degree ordering of G' can be transformed to a minimum node degree ordering of the original graph. For other reductions, nodes can be eliminated without increasing the number of fill-in edges even though they will not form a minimum node degree ordering. This is beneficial if many reductions can be applied and the calculation of the reductions is sufficiently fast.

In the following two reductions are described that have been used by Ost et al. [21] to reduce the running time which depends on the number of nodes and edges. They used it for their development of a nested dissection algorithm. Additionally one common reduction called element absorption, which is often applied in minimum degree ordering algorithms and was first introduced by Duff and Reid [10], is implemented.

The three reductions we use are exact reductions in the sense that applying them still produces a minimum node degree ordering. The *simplicial* node reduction eliminates nodes for which the neighborhood is already a clique as this does not add additional fill-in. This can result in an ordering which is not necessary a minimum node degree ordering. Two nodes that have the same closed neighborhood are called *indistinguishable* nodes and can be removed together as eliminating one makes the other simplicial.

Element absorption is a different kind of reduction as it does not change the graph itself but simplifies the problem by reducing the number of hyperedges. The hyperedges are often non-maximal cliques such that during the algorithm the vertices of some hyperedges are part of a larger hyperedge such that the smaller clique is absorbed fully by the larger one and can therefore be removed from the algorithm.

4.2.1 The Simplicial Node Reduction

If the neighborhood $N(v)$ of node v is a clique, v is called *simplicial* and can be removed without adding new edges. Therefore, simplicial nodes can be eliminated at the beginning of the node ordering without increasing the number of fill-in edges. An example of a simplicial node can be seen in Figure 4.1. The following theorem and proof are taken from the original paper by Ost et al. [21] and slightly changed to fit our notation.

Theorem 2. *Let $G = (V, E)$ be a graph with simplicial node v. The ordering $v\Sigma(G_v)$ is a minimum fill-in ordering of G.*

Proof. $N_G(v)$ is a clique such that $D_G(v) = \emptyset$. The fill-in introduced by the ordering $v\Sigma(G_v)$ is $\phi(G, v\Sigma(G_v)) = |D_G(v)| + \Phi(G_v) = \Phi(G)$. □

From this follows that in each step in our algorithm we can order simplicial nodes whenever they are detected without increasing the number of fill-in edges in our ordering.

4.2.2 The Indistinguishable Node Reduction

Let $G = (V, E)$. Nodes $u, v \in V$ are indistinguishable if $N[u] = N[v]$. We show that these two nodes can be eliminated together without increasing the number of fill-in edges. This means there exists a minimum fill-in ordering $x_1 \cdots x_i uv x_{i+1} \cdots x_{n-2}$ such that u and v can be eliminated together. In Figure 4.2 the nodes u, v are indistinguishable as they have the same closed neighborhood. Furthermore we can reduce G to G' by contracting the nodes u, v into a single node w and when w is eliminated in G' we

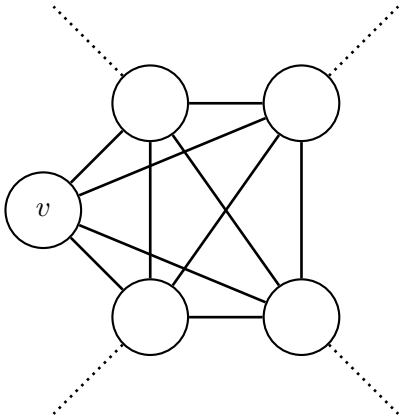

Fig. 4.1: v is simplicial as $N(v)$ is a clique.

eliminate u, v in the original graph. Without loss of generality we assume that u is the first node that is eliminated in G.

Proving that this can be done without affecting the number of fill-in edges is done in two steps. First we show that indistinguishable nodes stay indistinguishable during the elimination process such that we can contract them into a single node. Afterwards we show that the number of fill-in edges does not increase when we eliminate v directly after we eliminated u. This also holds if more than two nodes are indistinguishable. The following two theorems and proofs are taken from the original paper by Ost et al. [21]. Only the notation is changed to fit ours.

Theorem 3. *If $u, v \in V$ are indistinguishable nodes in $G = (V, E)$, then u, v are indistinguishable in any elimination graph G_x with $x \in V \setminus \{u, v\}$.*

Proof. Node x must be either a neighbor of u excluding v or a neighbor of u and v as $N[u] = N[v]$. Let $x \in N(u) \setminus \{v\} = N(v) \setminus \{u\}$ be eliminated from G. It holds in the elimination graph G_x that: $N_{G_x}(u) = (N_G(u) \setminus \{x\}) \cup N_G(x)$ as well as $N_{G_x}(v) = (N_G(v) \setminus \{x\}) \cup N_G(x)$. Therefore with $u \in N_{G_x}(v)$ and $v \in N_{G_x}(u)$ it holds that $N_{G_x}[u] = N_{G_x}[v]$.
For the second case let $x \notin N(u) \cup N(v)$. If x is eliminated in G the neighborhoods of u and v do not change in G_x, as $u, v \notin N(x)$.
Therefore in both cases it holds that $N_{G_x}[u] = N_{G_x}[v]$ such that u, v are indistinguishable in G_x if they are indistinguishable in G. □

The next theorem makes sure that applying the indistinguishable node reduction does not increase the fill-in of our ordering by showing that there exists a minimum degree ordering even if we use the reduction rule.

Theorem 4. *Let $G = (V, E)$ with a set of nodes $A \subseteq V$, such that $\forall a_i, a_j \in A, N[a_i] =$*

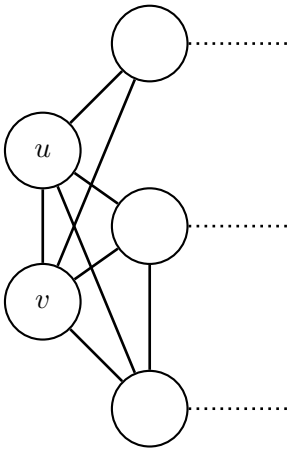

Fig. 4.2: u, v are indistinguishable nodes as $N[v] = N[u]$.

$N[a_j]$. *Let* $\{x_1, \ldots, x_k\} = V \setminus A$. *Then there exists an ordering* $\sigma = x_1 \cdots x_i A x_{i+1} \cdots x_k$ *such that* $\phi(G, \sigma) = \Phi(G)$.

Proof. Theorem 3 shows that each pair $a_i, a_j \in A$ is indistinguishable in every elimination graph. Let $a \in A$ be the first node in A that is eliminated from G. Then there is a graph $G^{(\ell)}$ in the sequence of elimination graphs such that the minimum degree ordering of $G^{(\ell)}$ is $a\Sigma(G_a^{(\ell)})$. $\forall b \in A \setminus \{a\}$ it holds $N_{G_a^{(\ell)}}(b)$ is a clique as eliminating one indistinguishable node of a pair of indistinguishable nodes makes the other node simplicial. Therefore $A\Sigma(G_A^{(\ell)})$ is a minimum degree ordering of $G^{(\ell)}$ and G has a minimum degree ordering $x_1 \cdots x_i A x_{i+1} \cdots x_k$. □

4.2.3 Element Absorption

Element absorption was first introduced by Duff and Reid [10] and does, in contrast to the previous two reductions, not order nodes directly or change the initial graph or the elimination graphs. Instead it simplifies the structure in which the graph is represented.

In Section 4.1 we introduced the way of using hyperedges to represent the cliques of the graph. During the elimination process it happens that a $K_k \subseteq K_\ell$ for some k, ℓ such that K_k can be removed from the list of hyperedges without changing the underlying graph itself. This is true as we are only interested in the union of all hyperedges that form the elimination graph.

The removal of such absorbed cliques can speed up the algorithm as it removes repetitive steps which would be done for K_k and K_ℓ. Figure 4.3 shows an example where a clique of size 4 absorbs a clique of size 3.

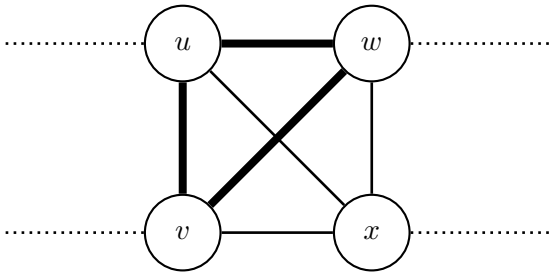

Fig. 4.3: The clique $\{u, v, w\}$ is absorbed by the clique $\{u, v, w, x\}$.

5 Implementation

Our code is integrated in the graph partitioning framework KaHIP - Karlsruhe High Quality Partitioning (https://github.com/KaHIP/KaHIP) by Sanders et al. [25]. In this chapter we show how we implement the base in Section 5.1. In the second part of this chapter, namely in Section 5.2, we specify how we implement the data reductions as explained in Section 4.2.

5.1 Minimum Degree Algorithm

In this section we discuss the data structures used to implement Algorithm 2 as shown in Section 4.1 and specify in what regards we differ from the approach by Cummings et al. [7].

In contrast to the original approach we use an adjacency list to represent the edges in our graph instead of an adjacency matrix. This reduces the space complexity of our algorithm. In [7], the authors used an adjacency matrix to simplify the running time analysis of the algorithm. We initialize the hyperedges as the given edges in the graph due to the fact that an initialization with maximal cliques did not turn out to be as performant. The degree of each node is computed once at the beginning and saved in a priority queue to access a node with minimum degree in constant time. We use random tie breaks for selecting the next node with minimum degree using the priority queue. Each hyperedge is saved as a flat hash set which is a data structure provided by the abseil library [1]. This data structure allows a fast way of accessing the nodes in a clique, adding or removing a node as well as checking whether a node is part of the clique. Furthermore, we save for each node the incident hyperedges to have a fast access to all hyperedges that contain the node with minimum degree.

Before we continue with the storage of our temporary cliques we explain again the functionality of the clique W which is used throughout this chapter. For this we include a summary of Algorithm 2 that can be seen in Algorithm 3. It additionally includes the pseudocode for the data reduction rules. Their implementation is explained in Section 5.2.

In each elimination step we create a new clique that is later added to our list of hyperedges and remove other cliques during this process. W is the new clique that is added to our list of hyperedges. We implemented this in such a way that we do not

Algorithm 3: Summary of the the minimum node degree ordering algorithm and data reduction rules.

```
   Result: The minimum node degree ordering
 1 Initialize the elimination graph G, ordering and hyperedges
 2 while G ≠ ∅ do
 3 |   Let v be the next elimination node
 4 |   Push v to the ordering
 5 |   W := ∅ (the new hyperedge)
 6 |   foreach hyperedge U that contains v do
 7 |   |   Remove U from the list of hyperedges
 8 |   |   U := U \ {v}
 9 |   |   Update the elimination graph
10 |   |   Add the nodes in U that are not already in W to the clique W
11 |   end
12 |   Apply element absorption
13 |   Push W to the list of hyperedges
14 |   Apply simplicial node reduction
15 |   Apply indistinguishable node reduction
16 end
```

actually add W to the list as this would increase the space requirements. Instead we overwrite the first clique that is removed in an elimination step by the new clique W.

The temporary cliques X, Y and W that we use in the algorithm are saved in various data structures. In general we use the same clique data structure using a flat hash set to have a fast access on the nodes in each clique. Additionally the cliques U and W are stored in global bitmasks to compute the set differences $X := W \setminus U$ and $Y = U \setminus W$. The bitmasks here simply represent which nodes are in the temporary cliques. The set differences can then be easily computed by iterating over the first clique and checking whether the bit is set in the second clique.

Adding an edge involves several steps. First we check whether the edge exists and if not we change the adjacency list for both nodes as well as updating their degree. A similar approach is used for removing an edge.

5.2 Data Reductions

Each of the following reductions are applied after each elimination step of the base algorithm. In this section we explain how each of the three data reduction rules are added to the base algorithm as shown in Section 4.1.

We use the Algorithm 3 as a reference for where we apply the data reduction rules and focuses on the necessary parts. The following line numbers always refer to Algorithm 3 if not stated otherwise.

The simplicial node reduction is applied after adding W to the list of hyperedges (line 13). It checks whether there are nodes which only appear in a single clique. If this is the case then the node is simplicial and can be ordered directly without adding any new fill-in edges. We use a priority queue to do this efficiently by saving the number of incident hyperedges for each node. It is initialized with the number of neighbors and gets updated when a clique is removed from the list of hyperedges in line 6 by subtracting one for all $u \in U$ as well as in line 12 when a clique is absorbed. Additionally after the new clique W is added in line 13 we add one to $w \in W$. The simplicial node reduction then simply takes a node with the smallest clique counter as long as this counter is ≤ 1. If a node is not part of any clique it can be eliminated directly and the next node can be checked. If the number of incident hyperedges is one, we found a simplicial node and add it to the ordering.

The indistinguishable node reduction involves a couple more steps. After each ordering step, i.e., after line 13, we need to check for each of the nodes in clique W whether it is an indistinguishable node. Each node $v \notin W$ can not become indistinguishable in this ordering step therefore they do not need to be checked. More precisely, they can only be indistinguishable to a node in W which can be detected by checking only nodes in W. For each node $v \in W$ it involves checking $N[v] = N[u]$ for all $u \in N[v]$. Checking the neighborhoods directly is not performant, therefore we use hashing to compare if two closed neighborhoods can be the same. For this we sum up the node ids of $N[v]\,, \forall v \in V$ in the initialization phase for each node and let it overflow in a 64 bit integer hash variable. Theses hashes are updated when the neighborhood of the corresponding node changes while we remove or add an edge (line 9). In the check for indistinguishable nodes we then compare whether the hashes match and whether the number of adjacent nodes is the same. If both of these prechecks are fulfilled we test if $N[v] = N[u]$. For two indistinguishable nodes we link the node v to the indistinguishable node u and remove u from the graph. When v is eliminated we order the linked node u as well directly after v. Additionally we increase the weight of v to compensate for the changed node degree of neighboring nodes. In the initialization phase we therefore set the weight of each node to 1 and when a node u is removed from our representation because it is indistinguishable from v we increase the weight of v by the weight of u. Whenever we eliminate edges incident to v we then decrease the degree by the current weight of v. Same is true for adding edges incident to v.

The element absorption reduction is in contrast to the other two performed before adding W to the list of hyperedges (line 12). This check removes all hyperedges that are completely absorbed by the new hyperedge W. Checking whether a clique is absorbed by any other clique is performed by iterating over the nodes $v \in W$. We iterate over each node v and increase a clique counter for all cliques it belongs to. This way we can check whether a clique consists only of nodes which are also in W. After this

counting step we check whether a clique has the same counter as its size as this can only happen iff the clique is absorbed by W. Then the clique is removed from the list of hyperedges as it is absorbed. It is important that this step is performed before we detect simplicial nodes as that algorithm depends on the fact that simplicial node is only in a single clique. The clique counter used in this reduction step is a global counter that is initialized with zeros and reset after each relevant clique is checked. This way the counter does not need to be reallocated in each elimination step.

5.2.1 Data Reduction Parameters

Based on our experimental evaluation we decided to introduce two different parameters. These can be used to reduce the potential additional running of the reductions. The running time of checking whether two nodes are indistinguishable as well as checking whether a hyperedge can be absorbed depends on the number of neighbors. For the indistinguishable node reduction more hashes have to be checked. Applying element absorption iterates over the number of cliques which depends on the number of neighbors. Additionally the probability that a node with a high degree has an indistinguishable counterpart is very low. This is also true for the element absorption data reduction. Due to these potential slow downs when reductions are applied we implemented two different options.

The first parameter is intended to speed up the algorithm by not applying reductions for high degree nodes. However, in certain instances this slows down the algorithm as the degree needs to be checked every time. Additionally before there might be some instances where applying reductions even for high degree nodes was actually valuable. This parameter evens out the potential slow down or speed up and does not change the fill-in as a the result is still a minimum node ordering.

Our second parameter can be used to filter out high degree nodes as a preprocessing step. The main algorithm including reductions is then only applied to the reduced graph. Those initially removed nodes are then ordered at the very end. This speeds up the algorithm a bit but sometimes introduces more fill-in as this customized algorithm is not computing a minimum degree ordering anymore. We experimented with a variety of different values for both parameters and discuss reasonable choices in Section 6.4. Due to the limited significance in the overall results we decided to not reevaluate all experiments but instead only add some plots where the performance changes can be observed.

6 Experimental Evaluation

In the following we show the benchmark results of different node ordering algorithms. This includes benchmarks with different parameters as well as a comparison to other types of node ordering algorithms.

We used all symmetric real graphs, with 1,000 to 1,000,000 nodes, from the Suite Sparse Dataset [9] for our benchmark tests. The number of instances with these criteria form a benchmark set of 394 instances. In those instances the number of edges is ranging from 1,000 to roughly 75 million. An extraction of those instances with the number of nodes and edges can be seen in Table 6.1 where we list every 15th instance ordered by number of nodes.

Besides the comparison of our algorithm with different parameters we also compared it to three algorithms from different software packages. The parameters of our algorithm are defining the data reduction rules we apply after each elimination step. We compare our algorithm regarding the number of fill-in edges and running time with Metis [19] (version 5.1.0), the current minimum node degree ordering algorithm in KaHIP (version 3.10) which we refer to as KaMND and to Scotch [2] (version 6.1.0). The Scotch framework provides different algorithm to find an elimination ordering similar to KaHIP. We mostly use their approximate minimum degree algorithm which we call ScAMD from now on. For Metis as well as KaMND the number of fill-in edges after reordering is computed using the ordering algorithm described by Rose et al. [24] which is implemented in KaHIP (version 3.10) [25]. The Scotch framework itself provides its own calculation to determine the number of fill-in edges which will be used for evaluating ScAMD. All benchmarks were run on a cluster running an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz with 100GB memory.

From now on we will refer to our implementation as HyperOrdering as it extensively uses an efficient hyperedge representation to gain its speed up when compared to other minimum degree ordering algorithms.

We first check whether the runtime of the base algorithm matches the theoretical running time in practice. Secondly, we compare the differences between applying various reductions to the instances in each ordering step compared to no reductions as described in Section 4.1. For each algorithm and its variation we always compare the number of fill-in edges and the running time. Afterwards, the benchmark results are compared to the other node ordering algorithms mentioned before. That comparison is split into

Table 6.1: Every 15th instance from our benchmark set with n number of nodes and m number of edges.

instance	n	m
saylr3	1 000	2 750
bcsstk11	1 473	32 768
c-18	2 169	12 976
c-21	3 509	28 636
bcsstk28	4 410	214 614
s1rmt3m1	5 489	212 162
Kuu	7 102	333 098
nemeth14	9 506	486 638
nemeth16	9 506	577 506
fv2	9 801	77 224
t2dah	11 445	164 672
Pres_Poisson	14 822	700 982
bodyy4	17 546	104 004
c-50	22 401	157 844
TSOPF_FS_b39_c7	28 216	722 962
helm3d01	32 226	396 218
case39	40 216	1 032 042
gridgena	48 962	463 122
c-67	57 975	472 254
H2O	67 024	2 149 712
rail_79841	79 841	474 080
s3dkq4m2	90 449	4 337 276
ship_003	121 728	3 655 308
Si41Ge41H72	185 639	14 825 626
c-big	345 241	1 995 618
af_shell9	504 855	17 083 990
ldoor	952 203	41 541 614

three parts. The first part is a comparison with KaMND in Subsection 6.3.1. Secondly, we compare it with approximate minimum node degree ordering which was explained in Chapter 3. For this we use the implementation provided by the Scotch framework and refer to it as ScAMD. The comparison is made in Subsection 6.3.2. The last evaluation is a comparison of HyperOrdering and nested dissection using Metis in Subsection 6.3.3. In Section 6.5 we conclude this chapter.

6.1 Runtime Analysis

The theoretical running time of the base algorithm, HyperOrdering without data reductions, is $\mathcal{O}(nm)$. In Figure 6.1 we can see that the theoretical running time holds in

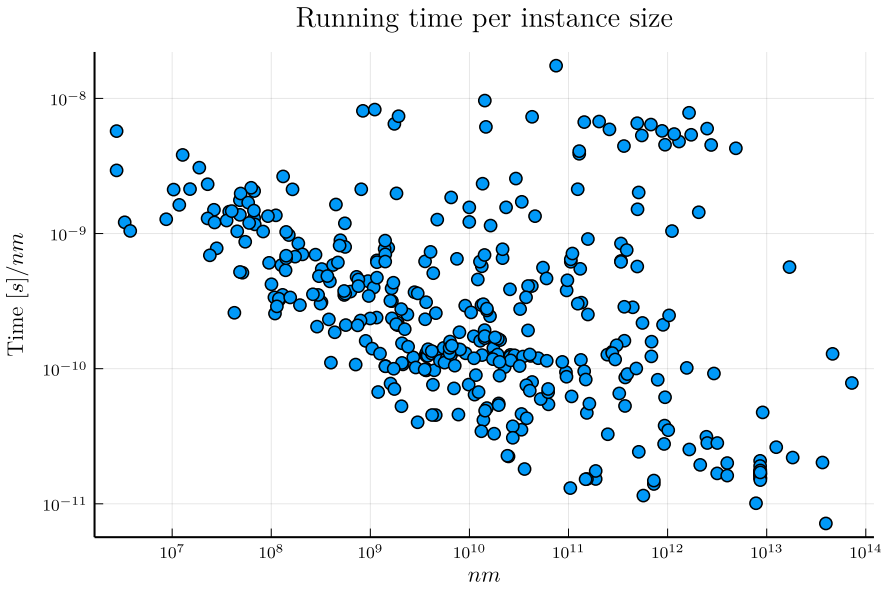

Fig. 6.1: The runtime in seconds per nm graph size is not growing for bigger graphs such that the theoretical runtime of $\mathcal{O}(nm)$ holds.

practice as the runtime per nm instance size is not increasing with bigger instances. In practice it is even faster than $\mathcal{O}(nm)$ which can be seen by the decreasing time per instance size for larger instances. Cummings et al. [7] provided a stricter bound which uses the maximum node degree in the initial graph. To describe the more precise running time we use the following notation: Δ is the maximum node degree in the initial graph and m^+ is the number of fill-in edges that get added in total. m^+ is not always easy to approximate beforehand but it leads to the tighter bound of $\mathcal{O}((m^+ + \Delta m^+) \log n)$. This bound can be used to estimate the running time for a class of matrices when the number of fill-in edges can be estimated. The practical running time using this tighter bound can be seen in Figure 6.2 which compares it against $(m^+ + 0.001\Delta m^+) \log n$. The constant 0.001 was chosen to see that the time spend compared to this new value of graph "size" is neither falling nor rising for bigger graphs or graphs with more fill-in.

6.2 Reductions

In this section we show the effect of applying data reductions to the minimum node degree algorithm as described in Section 4.1. For the following we have applied all reductions that are described by Ost et al. [21] as a preprocessing step. These data reductions are included in the KaHIP framework [25] and include the reduction of indistinguishable nodes and simplicial nodes as described in Section 4.2. Additionally they include twin reductions, path compression, degree-2 elimination and triangle contraction. All these reductions are applied only to the initial graph.

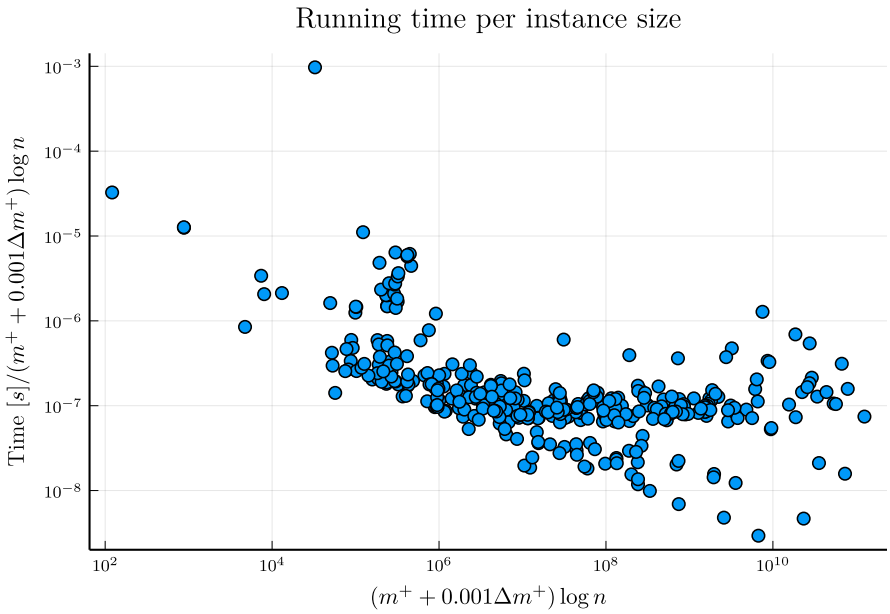

Fig. 6.2: The runtime in seconds per $(m^+ + 0.001\Delta m^+)\log n$ with m^+ being the number of fill-in edges and Δ as the maximum node degree in the initial graph.

Therefore, we only compare the effect of reductions applied during the minimum node degree ordering process as described in Section 4.2. Figure 6.3 shows the difference in number of fill-in edges when certain reductions are applied compared to our baseline algorithm in which no reductions beside the preprocessing root reductions are applied. Our described minimum degree ordering algorithm is not deterministic due to the fact that tie-breaks of minimum degree are broken randomly. For this reason we ran each algorithm four times for all of the 394 instances. Then we compared the number of fill-in edges when the specified reductions have been used to the mean fill-in of the four runs without reductions. The reductions applied are given a number for easier comparison:

1. Simplical node reduction
2. Indistinguishable node reduction
3. Element absorption

These can be combined such that "1 2 3" in Figure 6.3 represents all implemented reductions while "1 3" would not apply the indistinguishable node reduction.

The median number of fill-in edges is in all cases similar to the number of fill-in edges without reductions. Additionally, the lower and upper quartile are in a 5% range around the mean fill-in when no reductions are used during the minimum degree ordering. For some reductions the resulting ordering is still a minimum degree ordering such that those results are expected. This holds for both the indistinguishable node reduction

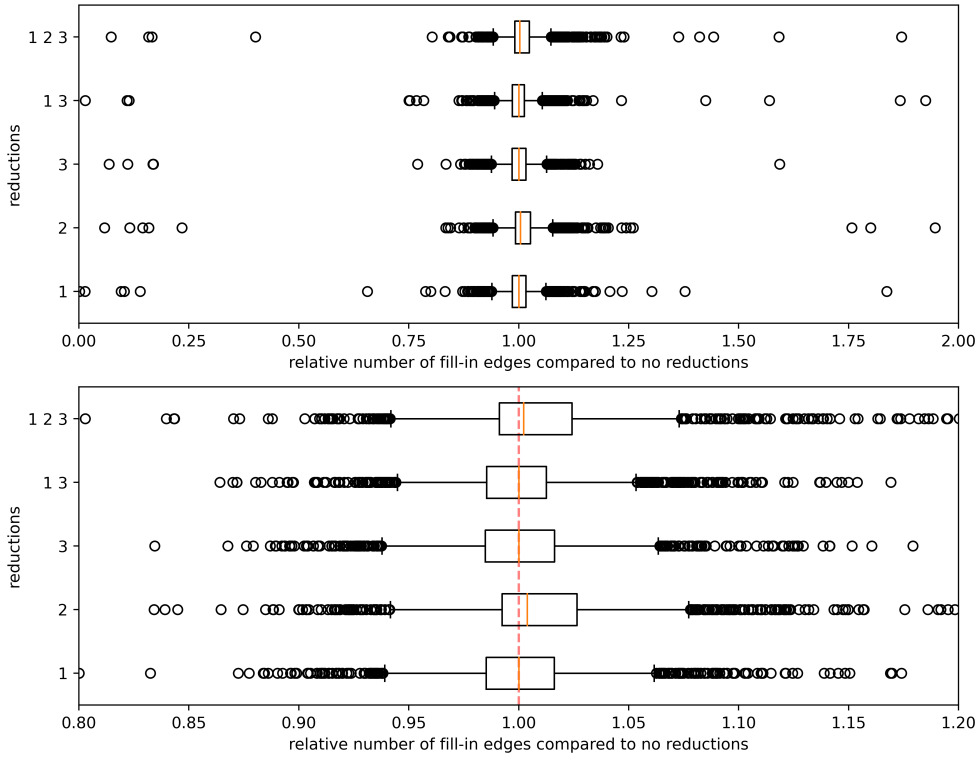

Fig. 6.3: Relative number of fill-in edges compared to the mean applying no reductions (taken from four runs). Each instance is run four times. Top: Shows the whole range (besides a single outlier for some reductions at 4 times the fill-in) Bottom: Shows the range from 80% fill-in to 120% fill-in

as well as element absorption. The simplicial node reduction on the other hand does not guarantee a minimum degree ordering, but it only eliminates higher degree nodes earlier if no additional fill-in is added. In general neither a relevant increase or decrease in number of fill-in edges was expected by those reductions which has been verified with this experimental evaluation. The main reason for applying reductions during the ordering process can be seen in Figure 6.4 and 6.5. In the former we show a scatter plot with a speedup for all 394 instances. In that case we took the mean of the four runs for every parameter setting. In the later where we do the same comparison as before but looking at the running times when the different reductions are applied and only show the zoomed in version based on some outliers that would otherwise distort the figure.

In Figure 6.4 we can see that the simplicial node reduction (blue) does not have an effect on the running time when applied without other reductions. Applying only element absorption (green) has a speedup of up to 3 for some of the instances but it is

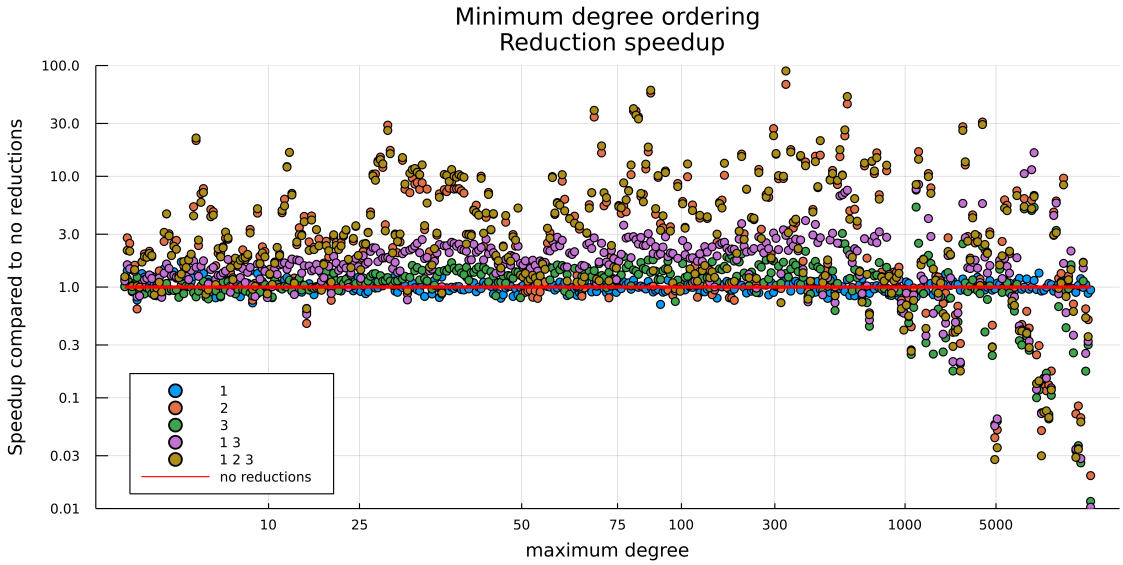

Fig. 6.4: Speedup of reductions compared to applying no reductions. Each dot is the mean of four runs. The instances are sorted by the maximum degree in the initial graph.

around the baseline for most of the instances. Applying both of them together (pink) however has a big impact on the running time with a speedup of about 2 for roughly 24% of the instances. This can be seen more clearly in Figure 6.5. Interestingly it holds for all parameter settings besides applying only the simplicial node reduction that there are certain types of instances which are slowed down by applying reduction rules. This can be mostly seen in the lower right of the figure where the dots are well below the baseline and are stacked on top of each other. This means that for those instances applying any of the three reductions slows down the algorithm. Most of the instances for which reductions slow down the process are instances with a maximum degree of 1000 or higher. In these instances most of the time no reductions were found such that search for possible reductions slowed down the search without providing the benefit of reducing the instance size. Due to this reason we added the parameters as described in Section 5.2.1. It is also true for the simplicial node reduction as that is only working correctly when the element absorption setting is set as well. The connection between the two reductions is due to the fact that we do not directly check whether a node is simplicial by checking whether the neighborhood is already a clique. We check instead whether the node is only part of a single hyperedge. As hyperedges can be overlapping when the element absorption reduction is not applied not all simplicial nodes are found. The instances in the Figure have been sorted by the maximum degree in the initial graph as this turned out to be a relatively good separator for distinguishing whether there is a potential slowdown when applying reductions.

In Figure 6.5 we can have a more detailed look at overall difference each reduction

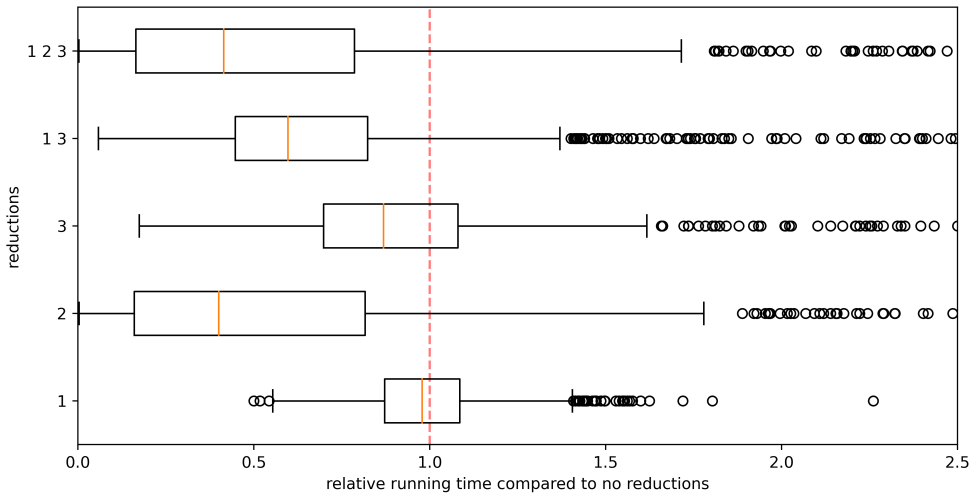

Fig. 6.5: Relative running time compared to applying no reductions.
Every of the 394 instances is run 4 times.
Zoomed in view from 0% running time to 250% running time.

makes. It shows a similar plot as the fill-in Figure 6.3 but for a comparison of running time. The single most important reduction is the indistinguishable node reduction (2). Applying this reduction rule during the ordering process increases the median performance by factor of about 2.5. For more than 75% of the instances applying the indistinguishable node reduction speeds up the algorithm by a factor of at least 1.2. Performing the indistinguishable node reduction alone is approximately equivalent to performing all three reductions when comparing the running time. Both have a median speed up of about 2.5 ($\pm 3\%$), the 75th percentile however is slightly better when applying all reductions with a speed up of at least 1.27 compared to a speed up of 1.22 when only the indistinguishable node reduction rule is used. We use this observation as a justification of using all data reduction rules for the further evaluation.

6.3 General Evaluation

In this section we compare our algorithm with other node ordering algorithms. Our algorithm is denoted as HyperOrdering and we use all implemented reductions as well as the preprocessing steps for reducing the graph size. The usage of all implemented reductions in each ordering step is justified in the previous section where we have seen that it performs slightly better than only applying the indistinguishable node reduction rule. We mostly compare our implementation to existing (approximate) minimum node degree algorithms. This includes the minimum degree ordering algorithm implemented in KaHIP [25] (version 3.10) which we refer to as KaMND and the approximate mini-

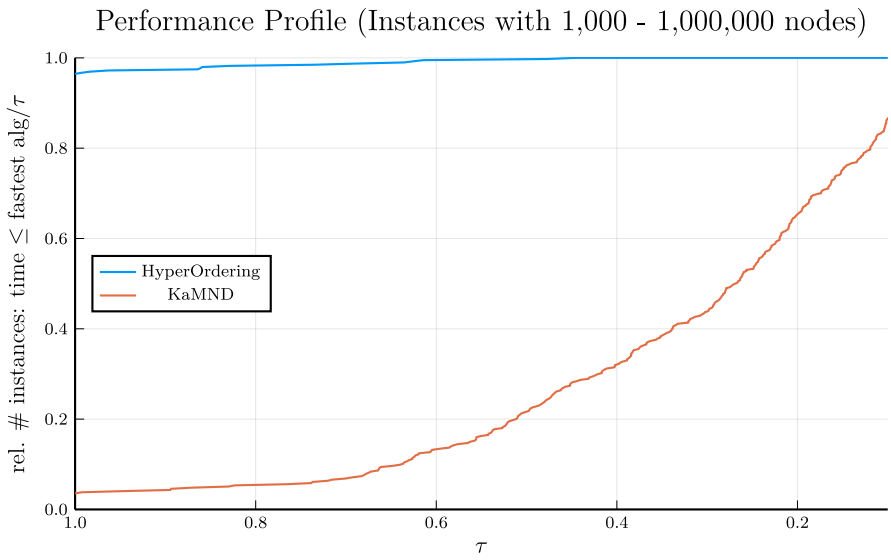

Fig. 6.6: Running time compared to fastest running time of the two algorithms per instance.

mum node degree algorithm in the Scotch [2] software package (version 6.1.0) which we named ScAMD. Additionally, we also compare HyperOrdering with the nested dissection approach that is implemented in Metis (version 5.1.0) [19].

6.3.1 Minimum degree ordering in KaHIP

In this subsection we compare our algorithm against KaMND, the minimum degree ordering approach implemented in the KaHIP framework (version 3.10). We start with comparing the running times of the two algorithms and afterwards looking at the fill-in difference produced by the two implementations.

In Figure 6.6 the difference of running time between our implementation and KaMND is shown. The currently implemented minimum node degree ordering algorithm uses a different data structure to compute the minimum degree ordering. Additionally that algorithm does only implement the indistinguishable node reduction rule which as we have seen in Section 6.2 is the reduction that improves the running time the most. On the y-axis we see the relative number of instances which have the same or faster running time compared to the artificial fastest of those two algorithms multiplied by τ which is given on the x-axis and is between 0 and 1. If one algorithm would run faster for all instances the respective curve would be the constant $y = 1$. In our case there is no algorithm which is always performing faster but HyperOrdering is the faster one for more than 95% of the instances. Additionally, one can see that our implementation is at least twice as fast for roughly 80% of the instances.

A similar plot but showing the number of fill-in edges is show in Figure 6.7. It shows that the number of fill-in edges is quite similar which is expected as they both mostly

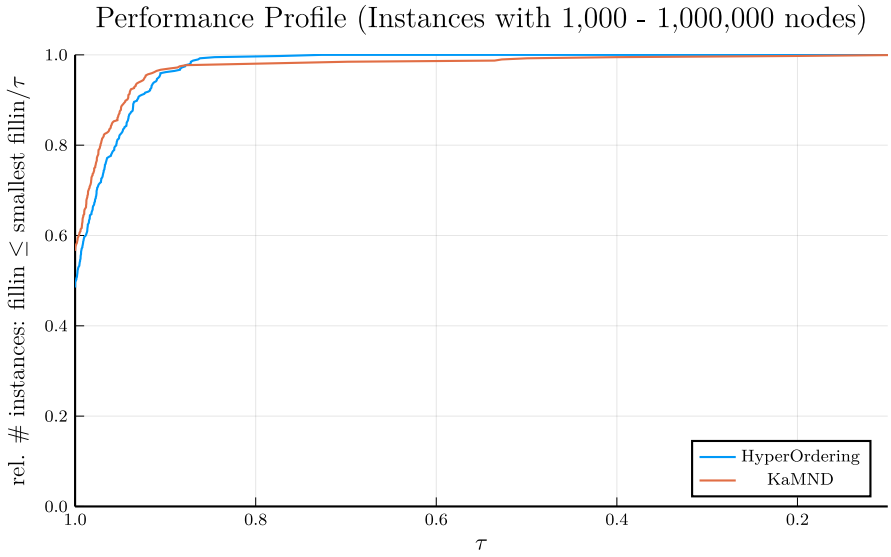

Fig. 6.7: Number of fill-in edges compared to smallest fill-in provided by one of the two algorithms.

perform minimum node degree ordering. KaMND uses only the indistinguishable node reduction and therefore computes a minimum degree ordering whereas our approach uses also the simplicial node reduction such that no minimum degree ordering is guaranteed. Nevertheless as described in Section 6.2 it results in a near minimum node degree ordering.

6.3.2 Approximate minimum degree ordering

In the following we compare HyperOrdering with ScAMD, the approximate minimum degree degree algorithm implemented in the Scotch framework. The Scotch framework implements three different ways for node ordering, namely approximate minimum degree ordering, approximate minimum fill-in ordering and nested dissection. We have discussed the approach of approximate minimum node degree ordering in Chapter 3. This part will mostly compare HyperOrdering against that approach and we mention an evaluation with the approximate minimum fill-in algorithm at the end.

The comparison of running time and number of fill-in edges is done with the same plots as in the previous subsection. Figure 6.8 shows the running time comparison and Figure 6.9 the evaluation with respect to the number of fill-in edges. The approximate minimum node degree algorithm clearly outperforms calculating the exact minimum node degree in terms of running time. It can be seen that ScAMD is at least 10 times faster in 80% of the tested instances. The running time was expected to be significantly smaller due to the fact that computing the exact node degree takes a major fraction of the time in the minimum node degree ordering.

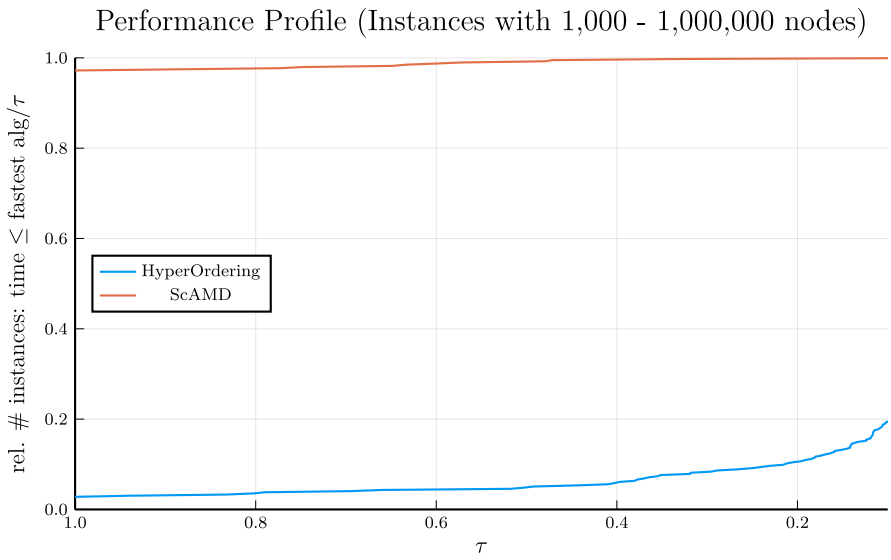

Fig. 6.8: Running time compared to fastest running time of the two algorithms per instance.

Figure 6.9 shows that calculating the exact node degree for the ordering is beneficial for reducing the number of fill-in edges. Nevertheless, the performance regarding the number of fill-in edges is in both cases relatively similar.

The Scotch framework additionally provides an approximate minimum fill-in algorithm which performs worse than their approximate minimum degree implementation for almost all instances in our benchmark set. More precisely the curve of the approximate minimum degree algorithm is always above the one for the approximate minimum fill-in algorithm. This holds for both the number of fill-in edges and the running time plot. Therefore we have not included a figure for this comparison.

6.3.3 Nested dissection

In this subsection we compare HyperOrdering approach with Metis, the nested dissection algorithm provided by Metis [19] (version 5.1.0). Nested dissection is generally used for large graphs for which minimum node degree orderings were too slow in the past. This is due to the fact that partitioning the graph is less dependent on the graph size. Nested dissection still applies minimum degree ordering for small enough subgraphs after several dissection steps have been applied.

In Figure 6.10 we can see that Metis is in almost all instances faster than our minimum node degree algorithm. It is at least twice as fast in roughly 80% of all tested instances and at least 10 times faster for 20% of the instances.

To our surprise Metis often provides an ordering with fewer fill-in edges than our algorithm which is visualized in Figure 6.11. There are nevertheless still about 40 out of the 394 instances for which the fill-in provided by the minimum node degree algorithm

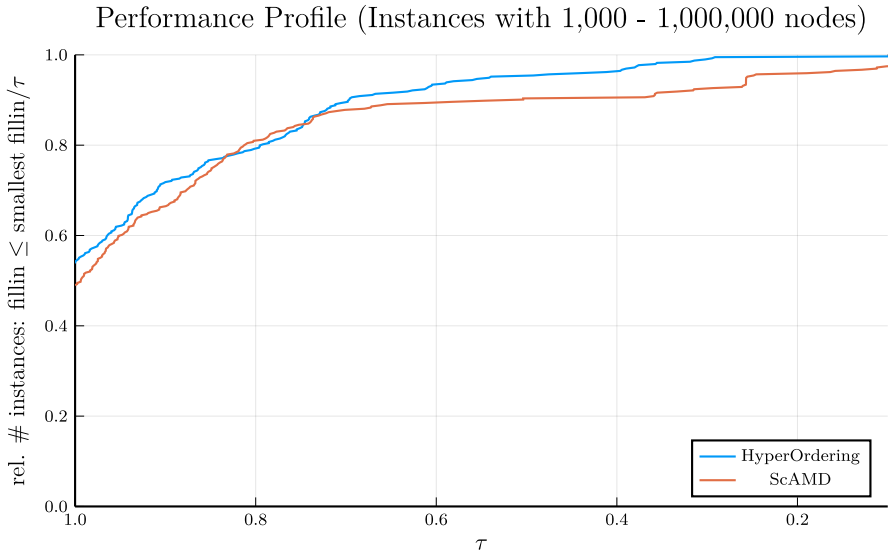

Fig. 6.9: Number of fill-in edges compared to smallest fill-in provided by one of the two algorithms.

is only 10% of the fill-in calculated by Metis. Generally this means that for certain instances the minimum degree ordering approach performs ordering with many fewer fill-in edges. This differences give reason to a further investigation for which instances one of these algorithm reduces the fill-in way better than the other.

Therefore, we tried to figure out simple preconditions for which instances Metis provides either way less or way more fill-in than HyperOrdering. For this we created a test set of random geometric graphs (RGG) using KaGen [13][22] which is a library to generate graph instances with certain properties. Our first idea of a possible precondition was to check whether the density of the graphs can explain the difference in the node ordering. We used KaGen to generate a new test set of 155 instances with a wide range of different densities while maintaining the main graph structure to only compare differences in densities. The comparison of Metis and HyperOrdering with this test set can be seen in Figure 6.12. The generated instances have a density between nearly 0% and roughly 10%. One can see that for most instances with density of 5% or more HyperOrdering provides results with less fill-in than Metis. In the figure this is shown as dots below the red line at $y = 1$ which represents the line of same fill-in as Metis. Every dot below that line is an instance where HyperOrdering provided an ordering with fewer fill-in edges than the nested dissection algorithm.

Additionally, we used KaHIP to compute an initial node separator for each instance to see whether the size of the node separator is an indicator for the difference between HyperOrdering and nested dissection. We assumed that there might be an effect of large separators such that for larger separators the resulting nested dissection ordering per-

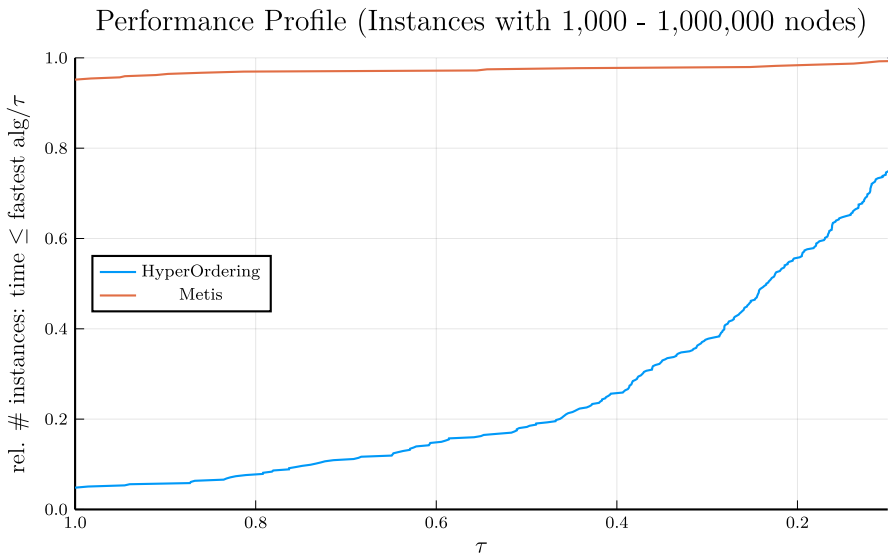

Fig. 6.10: Running time compared to fastest running time of the two algorithms per instance.

forms worse than HyperOrdering. Nevertheless, we could not find a correlation between these two quantities.

6.4 Parameters

Based on the comparison of applying the various data reduction rules in our algorithm as shown in Figure 6.4, we saw that using the reductions for nodes with more than 1,000 neighbors slowed down the search. Therefore, we decided to introduce two parameters as explained in Section 5.2.1. In short we have implemented one parameter to specify when reductions are not applied based on the degree of a node. The other parameter can be used to reduce the initial graph size by removing high degree nodes as a preprocessing step and order them at the very end. Our first parameter was intended to mitigate the slow down for instances with high degree nodes while still providing a minimum degree ordering. The effect of not applying reductions for nodes with a degree of at least 1,000 is shown in Figure 6.13. One can see that this parameter setting results in an overall more similar outcome to applying no reductions for instances with high degree nodes. In contrast to applying reductions for all nodes this parameter avoids slowdowns of a factor of 3 or more compared to applying no reductions. On the other hand using this new parameter does not have a high speed up either for some instances with high degree nodes.

The results of the second parameter can be seen in Figures 6.14 and 6.15. There we compared the standard HyperOrdering with HyperOrdering + Preprocessing. We choose to set the parameter such that nodes with a degree of more than 2^{14} are removed in the

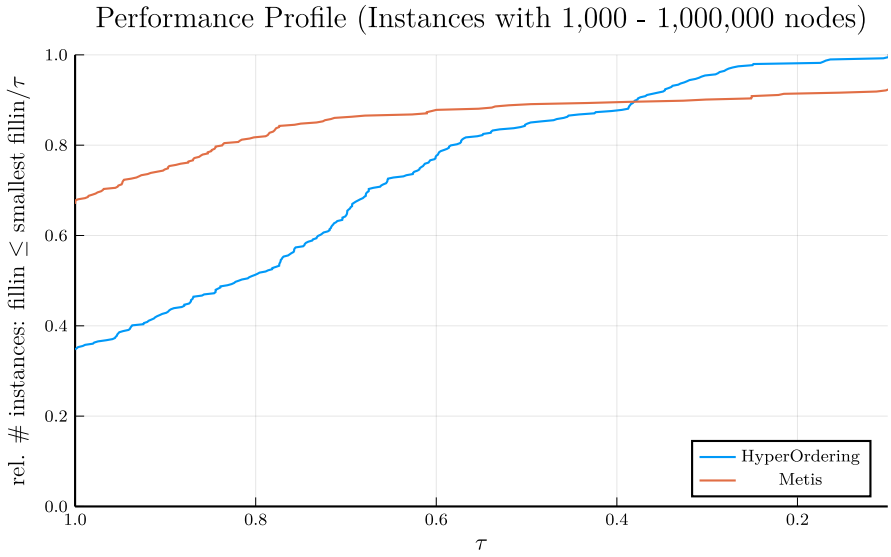

Fig. 6.11: Number of fill-in edges compared to smallest fill-in provided by one of the two algorithms.

preprocessing step and are ordered last. For this setting decreasing the parameter can substantially increase the fill-in while speeding up the algorithm due to removing many edges as a preprocessing step for instances with several of those high degree nodes. In some cases this optimization speed up HyperOrdering by a factor of 10 and more and is at least 10% faster for 5% of our benchmark instances. One has to note however that in certain instances this increased the number of fill-in edges by a factor of up to 5.

6.5 Summary

We have shown that our implementation HyperOrdering, which uses an efficient way to use hyperedges as a representation for the elimination graph, often outperforms the theoretical running time of $\mathcal{O}(nm)$ in practice. The tighter bound of $\mathcal{O}\left((m^+ + \Delta m^+)\ \log n\right)$ that was deduced by Cummings et al. [7] is also fulfilled in practice with Δ being the maximum node degree and m^+ the number of fill-in edges that get added.

Furthermore, we have verified that our three reduction rules namely, indistinguishable node reduction, simplicial node reduction and element absorption perform equally well with respect to the number of introduced fill-in compared to running the algorithm without reductions. Using the indistinguishable node reduction rule was mostly equivalent to applying all three reductions and speed up the base algorithm by a factor of 2.5 in the median.

In comparison to the current minimum degree algorithm in KaHIP our algorithm is at least twice as fast for more than 80% of the tested instances with comparable fill-

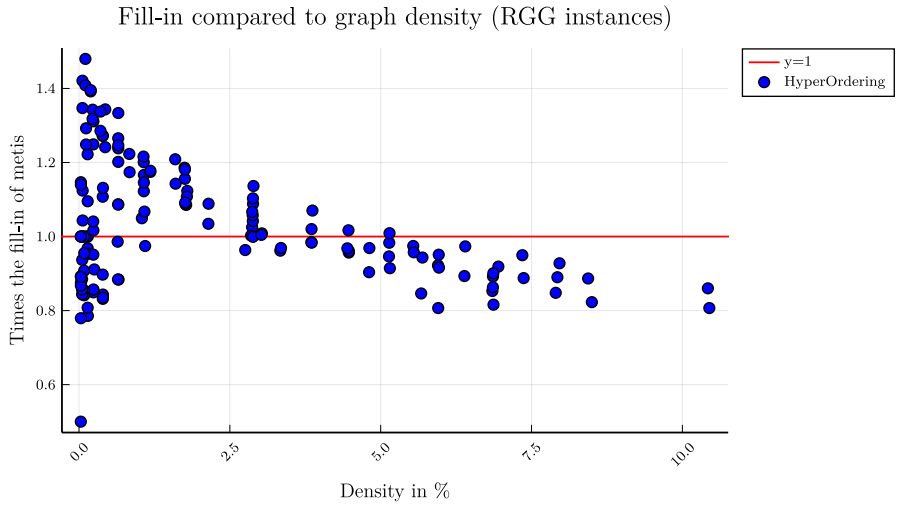

Fig. 6.12: Number of fill-in edges of HyperOrdering compared to nested dissection as calculated by Metis with respect to RGG graphs of varying density.

in. When compared to the approximate minimum node degree algorithm in the Scotch framework we have seen that our minimum node degree ordering performs slightly better with respect to the number of fill-in edges but approximating the degree provides a substantial speedup of about 10 for 80% of the instances. Lastly, we compared HyperOrdering to the nested dissection algorithm in Metis. Their nested dissection algorithm is about twice as fast for 80% of the instances and often provides an ordering with fewer fill-in edges but for about 10% of the instances the fill-in provided by HyperOrdering is roughly ten times smaller. In general combining those two algorithms might yield to the best combination with respect to fill-in and running time.

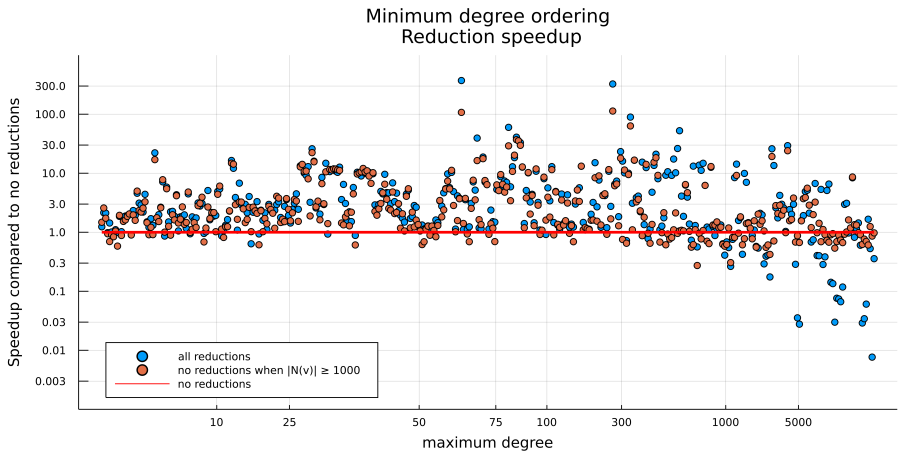

Fig. 6.13: Speedup of applying all reductions in each ordering step, no reductions for high degree nodes in comparison to no reductions.

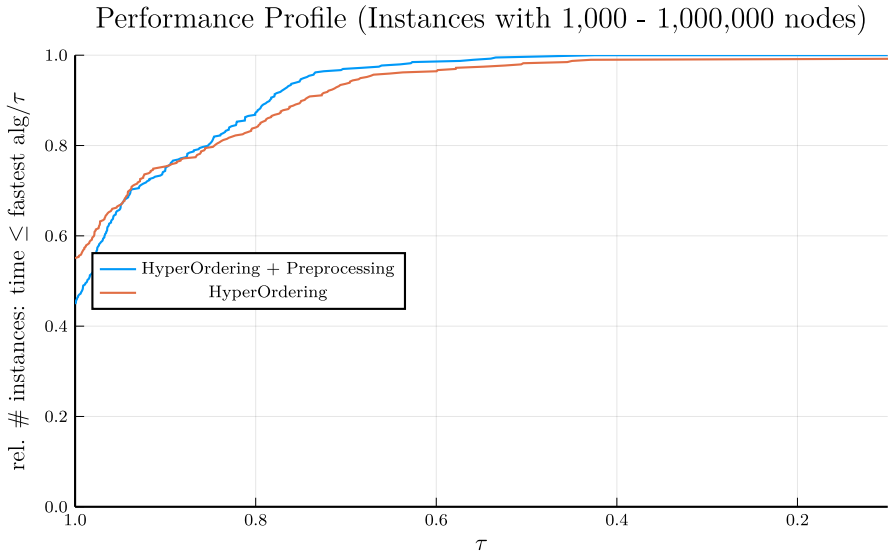

Fig. 6.14: Running time compared to fastest running time of the two algorithms per instance.

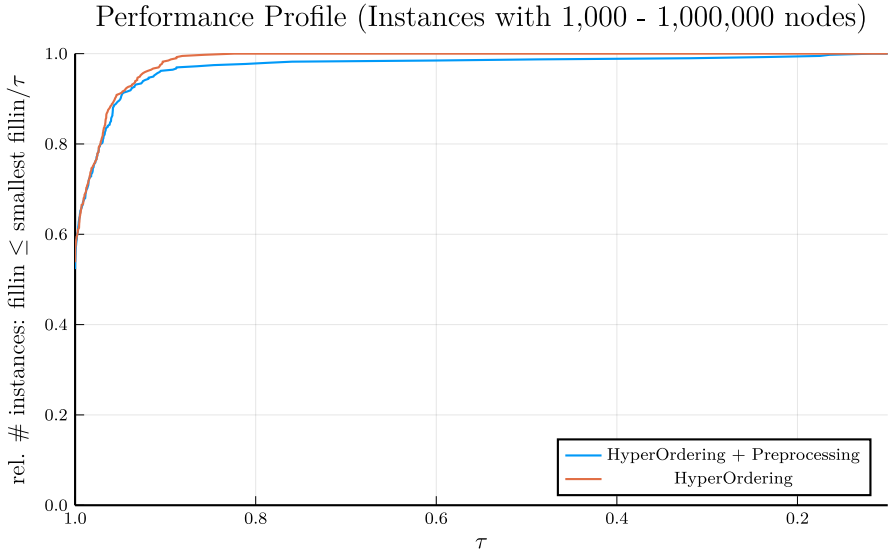

Fig. 6.15: Number of fill-in edges compared to smallest fill-in provided by one of the two algorithms.

7 Conclusion & Discussion

We implemented a minimum node degree algorithm which has a running time of $\mathcal{O}(nm)$ by using a different graph representation and reducing the number of needed node degree updates. The running time is often even faster and a more precise analysis was given in Section 6.1. Additionally we have shown that applying data reductions rules during each ordering step is effective with respect to the running time of the algorithm. In the median applying the implemented three reductions rules cut down the running time of our minimum degree algorithm by a factor of about 2.5 when compared with no reductions.

Furthermore, reductions made only a marginal difference in number of non-zeros in the Cholesky factorization compared to the base minimum node degree ordering algorithm. In comparison to approximate minimum node degree algorithms our exact approach of calculating the node degree is effective in reducing the number of fill-in edges. Nevertheless, an approximate minimum node degree algorithms outperforms our algorithm in running time often by more than an order of magnitude.

We are interested in applying the new data structure of hyperedges to the approximate minimum node degree framework. This can potentially speed up the algorithm compared to the currently used quotient graph. As we have seen the fill-in of the approximate degree ordering is close to the fill-in provided by an exact minimum degree ordering. Furthermore applying additional reduction rules before using the approximate degree ordering might shrink the gap between those two approaches.

In the future it might be of interest to implement some more reduction rules to reduce the running time further without increasing the number of fill-in edges. One of the possible data reductions rules that might have a similar positive effect as detecting indistinguishable nodes, is the twin reduction rule. The idea of this reduction is quite similar to the indistinguishable nodes but in this case the open instead of the closed neighborhood is compared. This can be seen in Figure 7.1 which shows the same graph excerpt as Figure 4.2 but without an edge between u and v. Adding this reduction rule is not straight forward and might need a new data structure to efficiently compute twin nodes in each ordering step. A naïve implementation of the twin reduction would need to check neighbors of neighbors of currently changed nodes in each ordering step in comparison to the indistinguishable node reduction which only compares the neighborhood of adjacent nodes.

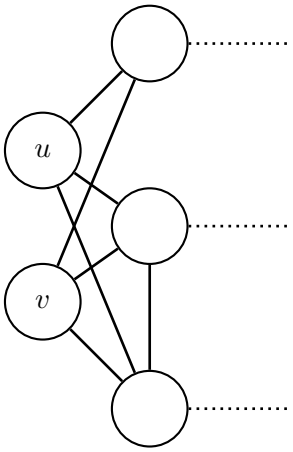

Fig. 7.1: u, v are twin nodes as $N(v) = N(u)$.

Additionally, having an insight into when reductions can be applied for high degree nodes would be useful to decide between the parameters explained in Section 6.4. One analysis for this would involve checking the number of reductions applied in each instance.

Of high interest would be an analysis for which graphs minimum degree ordering provides an ordering with fewer fill-in edges than approaches like nested dissection. This information could be used to decide in which particular cases of instances nested dissection algorithms like Metis will give a better node ordering compared to minimum node degree algorithms and vice versa. As minimum node degree algorithms are the base case for nested dissection this information could also be used to further improve the ordering provided by nested dissection algorithms. Additionally having an understanding on the underlying structure of the graph and the effect on the runtime of both algorithm types can help to decide when one wants to switch from the recursive approach of nested dissection to the base case of applying the minimum degree algorithm.

The base case in nested dissection algorithms is the part during the recursive reduction in which the remaining graph in the partition has fewer nodes than a predefined threshold. In those cases the partition is ordered using a minimum node degree ordering algorithm. These steps could result in overall less fill-in and HyperOrdering can be used to speed up this computation for existing nested dissection algorithms.

8 Bibliography

[1] *abseil/abseil-cpp*. https://github.com/abseil/abseil-cpp. – original-date: 2017-09-20T15:10:30Z

[2] *SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package*. https://www.labri.fr/perso/pelegrin/scotch/

[3] Abu-Khzam, F. ; Lamm, S. ; Mnich, M. ; Noe, A. ; Schulz, C. ; Strash, D. : Recent Advances in Practical Data Reduction. In: *CoRR* abs/2012.12594 (2020). https://arxiv.org/abs/2012.12594

[4] Amestoy, P. R. ; Davis, T. A. ; Duff, I. S.: An Approximate Minimum Degree Ordering Algorithm. 17, Nr. 4, 886–905. http://dx.doi.org/10.1137/S0895479894278952. – DOI 10.1137/S0895479894278952. – Publisher: Society for Industrial and Applied Mathematics

[5] Ashcraft, C. ; Liu, J. W.: A partition improvement algorithm for generalized nested dissection. In: *Boeing Computer Services, Seattle, WA, Tech. Rep. BCSTECH-94-020* (1994)

[6] Buluç, A. ; Meyerhenke, H. ; Safro, I. ; Sanders, P. ; Schulz, C. : Recent Advances in Graph Partitioning. Version: 2016. http://dx.doi.org/10.1007/978-3-319-49487-6_4. In: Kliemann, L. (Hrsg.) ; Sanders, P. (Hrsg.): *Algorithm Engineering - Selected Results and Surveys* Bd. 9220. – DOI 10.1007/978–3–319–49487–6_4, 117–158

[7] Cummings, R. ; Fahrbach, M. ; Fatehpuria, A. : A Fast Minimum Degree Algorithm and Matching Lower Bound. http://dx.doi.org/10.1137/1.9781611976465.45. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics (Proceedings). – DOI 10.1137/1.9781611976465.45, 724–734

[8] Dahlhaus, E. : Minimal elimination ordering for graphs of bounded degree. 116, Nr. 1, 127–143. http://dx.doi.org/10.1016/S0166-218X(00)00331-0. – DOI 10.1016/S0166–218X(00)00331–0

[9] Davis, T. A. ; Hu, Y. : The university of Florida sparse matrix collection. 38, Nr. 1, 1:1–1:25. http://dx.doi.org/10.1145/2049662.2049663. – DOI 10.1145/2049662.2049663

[10] Duff, I. S. ; Reid, J. K.: The Multifrontal Solution of Indefinite Sparse Symmetric Linear. 9, Nr. 3, 302–325. http://dx.doi.org/10.1145/356044.356047. – DOI 10.1145/356044.356047

[11] Fahrbach, M. ; Miller, G. L. ; Peng, R. ; Sawlani, S. ; Wang, J. ; Xu, S. C.: Graph Sketching against Adaptive Adversaries Applied to the Minimum Degree Algorithm. http://dx.doi.org/10.1109/FOCS.2018.00019. – DOI 10.1109/FOCS.2018.00019. – ISSN: 2575-8454

[12] Fomin, F. V. ; Kratsch, D. ; Todinca, I. ; Villanger, Y. : Exact Algorithms for Treewidth and Minimum Fill-In. 38, Nr. 3, 1058–1079. http://dx.doi.org/10.1137/050643350. – DOI 10.1137/050643350. – Publisher: Society for Industrial and Applied Mathematics

[13] Funke, D. ; Lamm, S. ; Meyer, U. ; Penschuck, M. ; Sanders, P. ; Schulz, C. ; Strash, D. ; Looz, M. von: Communication-free massively distributed graph generation. 131, 200–217. http://dx.doi.org/10.1016/j.jpdc.2019.03.011. – DOI 10.1016/j.jpdc.2019.03.011. – ISSN 0743–7315

[14] George, A. : Nested Dissection of a Regular Finite Element Mesh. 10, Nr. 2, 345–363. http://dx.doi.org/10.1137/0710032. – DOI 10.1137/0710032. – Publisher: Society for Industrial and Applied Mathematics

[15] George, A. ; Liu, J. W. H.: An Automatic Nested Dissection Algorithm for Irregular Finite Element Problems. 15, Nr. 5, 1053–1069. http://dx.doi.org/10.1137/0715069. – DOI 10.1137/0715069. – Publisher: Society for Industrial and Applied Mathematics

[16] George, A. ; Liu, J. W. H.: A Fast Implementation of the Minimum Degree Algorithm Using Quotient Graphs. 6, Nr. 3, 337–358. http://dx.doi.org/10.1145/355900.355906. – DOI 10.1145/355900.355906

[17] George, A. ; Liu, J. W.: The Evolution of the Minimum Degree Ordering Algorithm. 31, Nr. 1, 1–19. http://dx.doi.org/10.1137/1031001. – DOI 10.1137/1031001. – ISSN 0036–1445. – Publisher: Society for Industrial and Applied Mathematics

[18] Karypis, G. ; Kumar, V. : A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. 20, Nr. 1, 359–392. http://dx.doi.org/10.1137/

S1064827595287997. – DOI 10.1137/S1064827595287997. – ISSN 1064–8275. – Publisher: Society for Industrial and Applied Mathematics

[19] Karypis, G. ; Kumar, V. : *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. http://conservancy.umn.edu/handle/11299/215346 Accepted: 2020-09-02T15:04:02Z

[20] Markowitz, H. M.: The Elimination form of the Inverse and its Application to Linear Programming. 3, Nr. 3, 255–269. http://dx.doi.org/10.1287/mnsc.3.3.255. – DOI 10.1287/mnsc.3.3.255. – Publisher: INFORMS

[21] Ost, W. ; Schulz, C. ; Strash, D. : Engineering Data Reduction for Nested Dissection. http://dx.doi.org/10.1137/1.9781611976472.9. In: *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics (Proceedings). – DOI 10.1137/1.9781611976472.9, 113–127

[22] Penschuck, M. ; Brandes, U. ; Hamann, M. ; Lamm, S. ; Meyer, U. ; Safro, I. ; Sanders, P. ; Schulz, C. : Recent Advances in Scalable Network Generation. http://arxiv.org/abs/2003.00736

[23] Rose, D. J.: A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations. http://dx.doi.org/10.1016/B978-1-4832-3187-7.50018-0. In: Read, R. C. (Hrsg.): *Graph Theory and Computing*. Academic Press. – DOI 10.1016/B978–1–4832–3187–7.50018–0, 183–217

[24] Rose, D. J. ; Tarjan, R. E. ; Lueker, G. S.: Algorithmic Aspects of Vertex Elimination on Graphs. 5, Nr. 2, 266–283. http://dx.doi.org/10.1137/0205021. – DOI 10.1137/0205021. – Publisher: Society for Industrial and Applied Mathematics

[25] Sanders, P. ; Schulz, C. : Think Locally, Act Globally: Highly Balanced Graph Partitioning. https://doi.org/10.1007/978-3-642-38527-8_16. In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)* Bd. 7933. Springer, 164–175

[26] Sanders, P. ; Schulz, C. : Advanced Multilevel Node Separator Algorithms. Version: 2016. http://dx.doi.org/10.1007/978-3-319-38851-9_20. In: Goldberg, A. V. (Hrsg.) ; Kulikov, A. S. (Hrsg.): *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings* Bd. 9685. Springer. – DOI 10.1007/978–3–319–38851–9_20, 294–309

[27] Sanders, P. ; Schulz, C. ; Strash, D. ; Williger, R. : Distributed evolutionary k-way node separators. Version: 2017. http://dx.doi.org/10.1145/3071178.3071204. In: Bosman, P. A. N. (Hrsg.): *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*. ACM. – DOI 10.1145/3071178.3071204, 345–352

[28] Schulz, C. ; Strash, D. : Graph Partitioning: Formulations and Applications to Big Data. Version: 2019. http://dx.doi.org/10.1007/978-3-319-63962-8_312-2. In: Sakr, S. (Hrsg.) ; Zomaya, A. Y. (Hrsg.): *Encyclopedia of Big Data Technologies*. Springer. – DOI 10.1007/978–3–319–63962–8_312–2

[29] Speelpenning, B. : *Generalized element method. [For solving large, sparse symmetric systems of equations from networks; GAUSS, in FORTRAN for CDC 6600]*. https://www.osti.gov/biblio/6502916

[30] Tinney, W. F. ; Walker, J. W.: Direct solutions of sparse network equations by optimally ordered triangular factorization. 55, Nr. 11, S. 1801–1809. http://dx.doi.org/10.1109/PROC.1967.6011. – DOI 10.1109/PROC.1967.6011. – Conference Name: Proceedings of the IEEE

[31] Yannakakis, M. : Computing the Minimum Fill-In is NP-Complete. 2. http://dx.doi.org/10.1137/0602010. – DOI 10.1137/0602010

Eidesstattlich Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle wörtlich und sinngemäß übernommenen Textstellen als solche kenntlich gemacht habe. Dies gilt auch für die in der Arbeit enthaltenen Zeichnungen, Skizzen und graphischen Darstellungen.

Ort, Datum Unterschrift