1 Introduction

1.1 Motivation

The *maximum independent set* problem is a classic NP-complete graph problem [9] and therefore has been well studied over the last decades. Recent events like the PACE 2019 implementation challenge [**?**] show that maximum independent set and related problems are still of interest to researchers today. Given an undirected graph, the problem is to find a set of pairwise non-adjacent vertices of largest cardinality.

Applications of maximum independent set and its complementary problems *minimum vertex cover* and *maximum clique* cover a variety of fields including computer graphics [19], network analysis [18], rout planning [11] and computational biology [3, 7], among others. In computer graphics for instance, small vertex covers can be used to optimize the traversal of mesh edges in a triangle mesh. The dual graph of such a mesh is the graph that contains a vertex for each triangle and an edge between triangles that share a face. A minimum vertex cover in the dual graph corresponds to a minimal number of triangles that contain every mesh edge of the mesh. Unfortunately, due to the complexity of those problems, finding exact solutions to most real-world instances is computationally infeasible. However, a lot of work is invested into finding new techniques to handle the complexity.

One of the best known techniques for finding exact solutions to those problems, both in theory and practice, are branch and reduce algorithms. Such algorithms are generally based on kernelization. This means applying a set of reduction rules to decrease the complexity measure (in most cases the size) of an instance while still preserving the solvabilty. A solution to the original instance can then be constructed from a solution of the reduced instance in subexponential time. If an instance can not be reduced further, the algorithm branches into at least two subproblems of lower complexity which are then solved recursively. Branch and reduce algorithms also use problem specific upper and lower bounds to a solution and prune the search space by eliminating solutions which do not satisfy those bounds.

So far most studies on branch and reduce algorithms for the maximum independent set problem solely focused on finding new and improved reduction rules and lower or upper bounds, respectively. There are very few publications regarding different branching strategies. However, a comparison of three simple branching strategies by Akiba and Iwata [1] shows that the branching strategy can have a huge impact on the runtime of an algorithm.

1.2 Contributions

This thesis experimentally examines various different branching strategies for maximum independent set. We essentially follow two main approaches. The first approach is to branch on vertices that decompose the graph and then solve the resulting connected components independently. The second approach is to destroy complex structures by branching on vertices such that the structure can be reduced by kernelization afterwards. We implement a variety of different branching strategies following both approaches and compare them to the branching strategy used by the state of the art branch and reduce algorithm for minimum vertex cover proposed by Akiba and Iwata [1]. For testing we use instances from multiple graph classes.

Branching strategies following the first approach can also be used for other graph problems, but we did not evaluate this. We also did not analyze any of the branching strategies on a theoretical level.

1.3 Structure of Thesis

Following the brief introduction (Section 1), in Section 2 we introduce the notation and problem definitions used throughout this Thesis. Here, we also explain the two variants of a branch and reduce algorithm for maximum independent set that we used as a basic framework for testing our branching strategies. In particular we define the reduction rules used by the algorithm.
Section 3 gives an overview of related work focusing on branching strategies used by other branch and reduce algorithms for maximum independent set or the equivalent problems minimum vertex cover and maximum clique. In Section 4 we outline our approaches and explain the implemented branching strategies in detail. Section 5 contains the experimental results. We start by explaining our testing methodology and then state our results. Subsequently, we compare all branching strategies to each other and discuss the effectiveness of our approaches. Finally, in Section 6 proposals for future work will be discussed based on the results of the Thesis.

2 Preliminaries

This section introduces basic notation and problem definitions used throughout this Thesis.

2.1 Basic Definitions

An undirected graph $G = (V, E)$ is a tuple of a set V of vertices (also called nodes) and a set $E \subseteq \binom{V}{2}$ of edges. Two vertices $v, u \in V$ are called *adjacent* or neighbors if they are connected by an edge, i.e. $\{v, u\} \in E$. The set $N(v) := \{u \in V \mid \{v, u\} \in E\}$ of all neighbors of a vertex $v \in V$ is called the *neighborhood* of v and $N[v] := N(v) \cup \{v\}$ denotes to the *closed* neighborhood of v. The number $d(v) := |N(v)|$ is called the degree of a vertex. The neighborhood of a set of vertices S is defined as $N(S) := \bigcup_{v \in V} N(v) \setminus S$ and $N[S] := \bigcup_{v \in V} N[v]$ denotes the closed neighborhood of a set. For a vertex $v \in V$ we define $N^2(v) := N(N[v])$. For a subset $S \subseteq V$ the graph $G_S = (S, E \cap \binom{S}{2})$ is called the subgraph *induced by* S.

A subset $I \subseteq V$ is called an *independent set* (IS) of G, if no two vertices from I are adjacent, so formally if $\forall v, u \in I : \{v, u\} \notin E$. A *maximum independent set* (MIS) of G is an independent set of largest cardinality. The size of a maximum independent set is called the *independence number* of G and is denoted by $\alpha(G)$.
A subset $S \subseteq V$ is called a *vertex cover* of G if for all neighbors v and u in G either v or u (or both) is in S, so formally if $\forall \{v, u\} \in E : v \in S \vee u \in S$. A *minimum vertex cover* of G is a vertex cover of minimal cardinality. If I is a (maximum) independent set in G, then $V \setminus I$ is a (minimum) vertex cover in G.
A subset $C \subseteq V$ is called a *clique* of G, if any two vertices from C are adjacent, so formally if $\forall v, u \in C : \{v, u\} \in E$. A *maximum clique* of G is a clique of largest cardinality. If I is an (maximum) independent set in G, then I is also a (maximum) clique in the complement graph $\overline{G} = (V, \binom{V}{2} \setminus E)$.

A *path* $P = (v_1, \ldots, v_k)$ is a sequence of distinct vertices in G such that $\{v_i, v_{i+1}\} \in E$ for all $i \in \{1, \ldots, k-1\}$. A subgraph of G induced by a cardinality maximal subset of the vertices such that any two vertices are connected by a path is called a *connected component*. A graph that contains only one connected component is called *connected*.

A partition of V into (S, T) is called a cut and the set $C = \{\{v, u\} \in E \mid v \in S,\ u \in T\}$ is called its *cut set*. For two vertices s and t, a cut (S, T) such that $s \in S$ and $t \in T$ is called a s-t-cut. A subset $S \subset V$ is *called* a separator, if $G - S := (V \setminus S, E \cap \binom{V \setminus S}{2})$ has more connected components than G i.e. the removal of S from the graph splits at least one connected component of G. Analogous to cuts, a vertex separator that separates two vertices s and t is called a s-t-separator.

2.2 Algorithm Framework

In this Thesis we solely focus on testing new branching strategies. Thus, we do not implement our own branch and reduce algorithm but rather use a state of the art algorithm for minimum vertex cover by Akiba and Iwata [1] as a basis, and modify the branching step within it. Since minimum vertex cover and maximum independent set are complementary problems, the algorithm can be used to find the latter one by just inverting its output. This subsection briefly covers an overview of the algorithm (Algorithm 1) from the perspective of maximum independent set.

Given a graph as input, the algorithm starts with the kernelization step, i.e., reducing the instance's complexity by applying a set of reduction rules (described in 2.3). Next, the algorithm tries to prune the current branch by using different upper bounds to an optimal solution. If pruning was not successful but the reduced graph is empty, the algorithm just returns the current best solution size. Otherwise, the algorithm searches for an optimal solution in every connected component of the graph independently. This is denoted as the decomposition step. If a connected component can not be reduced further, the algorithm performs a branching step. In the branching step the algorithm uses the branching strategy to choose a vertex v and then branches into two subinstances. The first case is to include v into the current solution and the second case is to exclude v from the current solution, including its neighbors instead. Both subinstances are then solved recursively and the current optimal solution gets updated accordingly. The algorithm also makes use of branching rules covered in 2.3 (not to confuse with the actual branching strategy) that sometimes allow further reductions on branching. The *packing* branching rule manages a set of constraints which is updated on every branch and reduction step, and therefore is also handed to the recursive subcalls. For clarity, we omit the details of this in the pseudo code and refer the reader to the original article by Akiba and Iwata [1].

All of our branching strategies branch on a single vertex in each branching step. Thus, in the algorithm we only need to change the method that selects the vertex to branch in most cases. Some of our strategies also maintain additional information which gets updated in each branching step and gets distributed to subinstances accordingly. Other branching strategies require structural information which is obtained during each kernelization step. In those cases we also modify the decompose step or the reduction rules, respectively.

Since our first approach is to branch on vertices that decompose the graph, we also test a slightly modified version of the algorithm where we add an additional connected components check before the kernelization step. This way, if the graph gets disconnected in a branching step, the instance is decomposed and kernelization afterwards becomes more efficient.

2.3 Reduction and Branching Rules

In this subsection we explain the various reduction and branching rules used in the kernelization and branching steps of the algorithm by Akiba and Iwata [1]. We formulate all reduction and branching rules for the maximum independent set problem, although the algorithm was originally designed for minimum vertex cover and therefore uses the equivalent counterparts of those rules. The algorithm also keeps track of the order in which the reduction rules are applied, such that a correct solution to the original instance (a specific maximum independent set and not only the size of one) can be constructed from a solution in the reduced graph later on. For clarity, we omit the details of this.

The first reduction rule, the degree one reduction, is actually completely contained in the dominance and unconfined reductions (covered later in this section). Nevertheless, due to its low computational costs, it is used in addition to those more general rules.

Algorithm 1: branch & reduce algorithm for MAX INDEPENDENT SET – Akiba and Iwata [1]

Input: A graph G, current solution size c, current best solution size k

```
1  Solve(G, c, k) begin
2      G, c ← Reduce(G, c)                                   // Kernelization
3      l ← UpperBound(G)                                     // Calculate upper bounds
4      if c + l ≤ k then return k                            // Prune current branch
5      if G is empty then return k
6      if G is not connected then
7          foreach connected component G_i of G do
8              c ← c + Solve(G_i, 0, k − c)                  // Solve connected components independently
9          return max{c, k}
10     (G_1, c_1, k), (G_2, c_2, k) ← Branch(G, c)           // Branch on a vertex v into two subcases
11     k ← max{k, Solve(G_1, c_1, k)}
12     k ← max{k, Solve(G_2, c_2, k)}
13     return k
```

Output: the size k of a maximum independent set or the size $n - k$ of a minimum vertex cover

Theorem 1. *(Degree One Reduction) Let $G = (V, E)$ be a graph with a vertex v of degree one and let u be the only neighbor of v. Then, there is a maximum independent set that includes v and therefore excludes u.*

Proof. Consider a maximum independent set I in G. I has to contain either u or v because otherwise $I \cup \{v\}$ would be an *independent set* of larger size. If I contains u, it can not contain v and thus, $(I \setminus \{u\}) \cup \{v\}$ is another maximum independent set that include v and excludes u. □

At the beginning of each kernelization step, the algorithm searches for vertices of degree one, includes them into the current solution and deletes their neighbors from the graph. The algorithm also checks whether removing a neighbor from the graph produces new degree one vertices, and in this case further applies the degree one reduction.

The next reduction rule deals with vertices of degree two that are not part of a triangle, i.e., whose neighbors are not adjacent and was introduced by Chen et al. [5].

Theorem 2. *(Degree Two Folding) Let $G = (V, E)$ be a graph with a vertex v of degree two and let u, w be the neighbors of v. Let $G' = (V', E')$ be the graph with $V' = (V \setminus N[v]) \cup \{x\}$ where $x \notin V$ and $E' = (E \cap \binom{V'}{2}) \cup \{\{x, y\} \mid y \in (N(u) \cup N(w)) \setminus \{v\}\}$ and let I' be a maximum independent set of G'. Then,*

$$I = \begin{cases} I' \cup \{v\} & \text{, if } x \notin I' \\ (I' \setminus \{x\}) \cup \{u, w\}) & \text{, else} \end{cases}$$

is a maximum independent set in G.

Proof. Consider any maximum independent set I of G. If I contains v, then it can not contain u and w. Thus, $I \setminus \{v\}$ is an *independent set* in G' of size $|I| - 1$. Otherwise, if I does not contain v, I has to include at least one neighbor of v (since I is maximal). If I contains only one neighbor of v, removing this neighbor from I yields an *independent set* in G' of size $|I| - 1$. If I contains both u and w, then $I' = (I \setminus \{u, w\}) \cup \{x\}$ is an independent set in G' of size $|I| - 1$. So, in total $\alpha(G') \geq \alpha(G) - 1$. On the other hand, I constructed from a maximum independent set I' of G' is an *independent set* of G of size $|I| = |I'| + 1$ and thus, I is a maximum independent set in G. □

So, if the algorithm finds a vertex v of degree two whose neighbors are not adjacent, the algorithm reduces the size of the graph by removing $N[v]$ adding a new vertex connected to $N^2(v)$ instead. This procedure is called *folding* the closed neighborhood of v, hence the name degree two folding.

The next two reduction rules can be used to delete single vertices that are not required in a maximum independent set. The first of those rules (the dominance Reduction) is fully contained in the second rule (the unconfined reduction) and therefore the algorithm only uses the latter one. However, we used the concept of dominance to design one of our branching strategies. For this reason the dominance reduction rule is also featured in detail.

Definition 1. *(Dominance) In a Graph $G = (V, E)$ a vertex u is called dominated by a neighbor v, if $N[u] \subseteq N[v]$.*

Theorem 3. *(Dominance Reduction) In a Graph $G = (V, E)$, if a vertex u is dominated by a neighbor v, then, there always exists a maximum independent set that does not include v, i.e.*

$$\alpha(G) = \alpha(G - v)$$

Proof. Consider a maximum independent set I that does contain v. Since $N[u] \subseteq N[v]$, I can neither contain u nor any of its neighbors. But then, clearly, $I' = (I \setminus \{v\}) \cup \{u\}$ is an *independent set* of the same size as I that does not include v. □

Thus, if a vertex v dominates another vertex u, one could safely remove v from the graph without compromising the solvabilty of the instance.

The core idea of the *unconfined reduction* proposed by Xiao and Nagamochi [21] is to detect a vertex that is not required for a maximum independent set and therefore can be removed from the graph by algorithmically contradicting the assumption that every maximum independent set contains the vertex.

Definition 2. *(Removable Vertex) In a graph $G = (V, E)$ a vertex v is called removable, if*

$$\alpha(G) = \alpha(G - v)$$

Definition 3. *(Child, Parent) In a Graph $G = (V, E)$ with an independent set I, a vertex v is called a child of I, if $|N(v) \cap I| = 1$ and the unique neighbor of v in I is called the parent of v.*

Theorem 4. *In a graph $G = (V, E)$ let S be an independent set that is not maximal but is contained in every maximum independent set of G and let v be any child of S. Then, every maximum independent set includes at least one vertex from $N(v) \setminus N[S]$.*

Proof. Assume that there is a maximum independent set I that includes S but no vertex from $N(v) \setminus N[S]$ and let u be the parent of v in S. Then, $I' = (I \setminus \{u\}) \cup \{v\}$ is an independent set of the same size as I, since I contains no neighbor of v other than u. This contradicts the fact that every maximum independent set includes S. □

Based on Theorem 4 Algorithm 2 detects so called *unconfined* vertices.

Theorem 5. *(Unconfined Reduction) In a Graph $G = (V, E)$, if Algorithm 2 returns true for an unconfined vertex v, then, there is always a maximum independent set that does not contain v.*

Proof. Assume, that v is included in every maximum independent set. Every vertex added to S by the algorithm is the unique neighbor of a child of S. Therefore, by Theorem 4 this vertex also has to be contained in every maximum independent set, and thus can be added to S. If the algorithm returns true, then there is a child of S that has no neighbor that can be included in S. Thus, by Theorem 4 the assumption that v is included in every maximum independent set was false and therefore v is removable. □

During kernelization, the branch and reduce algorithm uses Algorithm 2 to detect and remove unconfined vertices.

Algorithm 2: Unconfined – Xiao and Nagamochi [20]

Input: A graph G, a vertex v

```
1 Unconfined(G, v) begin
2     S ← {v}
3     while S has child u with |N(u) \ N[S]| ≤ 1 do
4         if |N(u) \ N[S]| == 0 then
5             return true                          // Contradiction to Theorem 4
6         else
7             {w} ← N(u) \ N[v]                    // By assumption w also has to
8             S ← S ∪ {w}                          // be contained in every MIS
9     return false
```

Output: true if v is unconfined, false otherwise

The twin reduction by Xiao and Nagamochi [20] deals with pairs of degree three vertices that share the same neighborhood.

Definition 4. *(Twins) In a Graph $G = (V, E)$ two vertices u and v are called twins, if $N(u) = N(v)$ and $d(u) = d(v) = 3$.*

Theorem 6. *(Twin Reduction) In a Graph $G = (V, E)$ let vertices u and v be twins. If there is an edge among $N(u)$, then there is always a maximum independent set that includes $\{u, v\}$ and therefore excludes $N(u)$. Otherwise, let $G' = (V', E')$ be the graph with $V' = (V \setminus N[\{u, v\}]) \cup \{w\}$ where $w \notin V$ and $E' = (E \cap \binom{V'}{2}) \cup \{\{w, x\} \mid x \in N^2(u)\})\}$ and let I' be a maximum vertex cover in G'. Then,*

$$I = \begin{cases} I' \cup \{u, v\} & \text{, if } w \notin I' \\ (I' \setminus \{w\}) \cup N(u) & \text{, else} \end{cases}$$

is a maximum independent set in G.

Proof. For the first case (there is an edge among $N(u)$) consider a maximum independent set I that does not contain u or v. Then, I has to include at least one neighbor of u and v, because otherwise $I \cup \{u, v\}$ would be an *independent set* larger than I. On the other hand, since there are neighbors of u and v that are adjacent, I can only contain at most two neighbors of u and v. But then, $I' = (I \setminus N(u)) \cup \{u, v\}$ is an *independent set* of the same size as I that includes u and v.
For the second case (no edges among $N(u)$) note, that the reduction produces a set that in both cases contains exactly two vertices more than a maximum independent set in G'. Now consider a maximum independent set I in G. If $N(u)$ is completely contained in I ($N(U) \subseteq I$), then I can not contain any vertex of $N^2(u)$, i.e., any neighbor of w in G'. Thus, $I' = (I \setminus N(u)) \cup \{w\}$ is an *independent set* of G' of size $|I| - 2$. Otherwise, I contains at most two vertices from $N(u) \cup \{u, v\}$ (either u and v or two vertices from $N(u)$). But then, $I' = I \setminus (N(u) \cup \{u, v\})$ is also an *independent set* of G' of size $|I| - 2$. In total $\alpha(G) \leq \alpha(G') + 2$ and thus, I is a maximum independent set of G □

During the kernelization step, the algorithm searches for twins u and v. If there is an edge among $N(u)$, the algorithm includes u and v to the current solution and deletes $\{u, v\} \cup N(u)$. Otherwise, the algorithm still deletes $\{u, v\} \cup N(u)$ introducing a new vertex connected to $N(u) \setminus \{u, v\}$ instead.

The next reduction rule as well as its special cases were also proposed by Xiao and Nagamochi [20].

Definition 5. *(Alternative Sets) In a Graph $G = (V, E)$ two non empty, disjoint subsets $A, B \subseteq V$ are called alternatives, if $|A| = |B|$ and there is a maximum independent set I in G such that $I \cap (A \cup B)$ is either A or B.*

Theorem 7. *(Alternative Reduction) In a Graph $G = (V, E)$ let A and B be alternative sets. Let $G' = (V', E')$ the graph with $V' = V \setminus (A \cup B \cup (N(A) \cap N(B)))$ and $E' = (E \setminus \binom{A \cup B \cup (N(A) \cap N(B))}{2}) \cup \{\{x, y\} \mid x \in N(A) \setminus N[B], y \in N[B] \setminus N(A)\}$ and let I' be a maximum independent set in G'. Then,*

$$I = \begin{cases} I' \cup A & \text{, if } (N(A) \setminus N[B]) \cap I' = \emptyset \\ I' \cup B & \text{, else if } (N(B) \setminus N[A]) \cap I' = \emptyset \end{cases}$$

is a maximum independent set in G.

Proof. Consider a maximum independent set I in G and without loss of generality let $A \subseteq I$ (by definition A or $B \subseteq I$). Thus, $I \cap ((A \cup B \cup (N(A) \cap N(B))) = A$ and $I \cap (N(A) \setminus N[B]) = \emptyset$. Now let $I' = I \setminus A$. I' is an *independent set* in G', since each added edge (from $E' \setminus E$) is incident to a vertex from $N(A) \setminus N[B]$ and $|I'| = |I| - |A|$. This implies $\alpha(G') + |A| \geq \alpha(G)$.
Conversely, let I' be a maximum independent set of G'. Obviously, I' is also an *independent set* of G. Since vertices from $N(A) \setminus N[B]$ are pairwise adjacent to vertices form $N(B) \setminus N[A]$, I' can only contain vertices from either $N(A) \setminus N[B]$ or $N(B) \setminus N[A]$. But then, $I = I' \cup A$ or $I = I' \cup B$ respectively is an *independent set* in G. Thus, $\alpha(G') + |A| \leq \alpha(G)$
In total $\alpha(G') + |A| = \alpha(G)$ and I is a maximum independent set in G. □

Note that the *alternative reduction* adds new edges between existing vertices of the graph. For this reason, applying the *alternative reduction* is not beneficial in every case. To counteract this, the algorithm only uses the following special cases of the *alternative reduction*.

Definition 6. *(Funnel) In a Graph $G = (V, E)$ two adjacent vertices u and v are called funnels, if $G_{N(v) \setminus \{u\}}$ is a complete graph, i.e, if $N(v) \setminus \{u\}$ is a clique.*

Theorem 8. *(Funnel Reduction) In a Graph $G = (V, E)$ let u and v be funnels. Then, $\{u\}$ and $\{v\}$ are alternative sets.*

Proof. We have to show that there is a maximum independent set that contains either v or u. So, consider a maximum independent set I that excludes both u and v. Then, I has to include at least one vertex from $N(v) \setminus \{u\}$, because otherwise $I \cup \{v\}$ would be an *independent set* of larger size. On the other hand, I can only contain at most one vertex x from $N(v) \setminus \{u\}$, since $N(v) \setminus \{u\}$ is a clique. But then, $(I \setminus \{x\}) \cup \{v\}$ is an *independent set* of the same size as I that does contain v. Thus $\{u\}$ and $\{v\}$ are alternative sets. □

Definition 7. *(Desk) In a Graph $G = (V, E)$ a cycle $u_1u_2u_3u_4$ of length four with no chords (i.e., an induced 4-cycle) is called a desk, if each of the vertices has at least degree three, $N(\{u_1, u_3\}) \cap N(\{u_2, u_4\}) = \emptyset$ and $|N(\{u_1, u_3\}) \setminus \{u_2, u_4\}| \leq 2$ as well as $|N(\{u_2, u_4\}) \setminus \{u_1, u_3\}| \leq 2$.*

Theorem 9. *(Desk Reduction) In a Graph $G = (V, E)$ let $u_1u_2u_3u_4$ be a desk. Then, $\{u_1, u_3\}$ and $\{u_2, u_4\}$ are alternative sets.*

Proof. Consider a maximum independent set I of G. If $|I \cap \{u_1, u_2, u_3, u_4\}| > 1$, then clearly $I \cap \{u_1, u_2, u_3, u_4\}$ is either $\{u_1, u_3\}$ or $\{u_2, u_4\}$. Otherwise, without loss of generality $u_2, u_3, u_4 \notin I$ and $|I \cap N[\{u_1, u_3\}]| = 2$. The last equation holds because $|N(\{u_1, u_3\}) \setminus \{u_2, u_4\}| \leq 2$ by definition, and u_1 has at least one neighbor in $N(\{u_1, u_3\}) \setminus \{u_2, u_4\}$ ($d(u_1) \geq 3$). But then, $(I \setminus \{N(\{u_1, u_3\})\}) \cup \{u_1, u_3\}$ is an independent set of the same size as I that does contain $\{u_1, u_3\}$. Thus, $\{u_1, u_3\}$ and $\{u_2, u_4\}$ are alternative sets. □

During kernelization, the algorithm searches for funnels or desks and reduces those structures according to the alternative reduction.

The algorithm also uses a reduction based on a solution to the LP-Relaxation of maximum independent set.

$$\begin{aligned} &\text{maximize} \sum_{v \in V} x_v \\ 0 \leq x_v \leq 1 \quad &\forall v \in V \\ x_v + x_u \leq 1 \quad &\forall \{u, v\} \in E \end{aligned}$$

Nemhauser and Trotter show that there always exists an optimal half integral solution to the LP-Relaxation, i.e., an optimal solution where $x_v \in \{0, \frac{1}{2}, 1\}$ for all $v \in V$ [17]. They also show that given an optimal half integral solution to the LP-Relaxation, there is always a maximum independent set that includes all vertices v with $x_v = 1$ and excludes all vertices u with $x_u = 0$. Furthermore, they show that finding an optimal half integral solution can be reduced to computing a *maximum matching* in a bipartite graph.
Iwata et al. develop an algorithm that given any optimal half integral solution constructs another half integral solution that minimizes the number of variable with half integral value [10].
The algorithm uses this solution to the LP-Relaxation to reduce the graph and also as an upper bound to an optimal solution.

Apart from reduction rules, the algorithm also uses branching rules that allow further reductions on branching when certain conditions hold. The first branching rule, mirror branching, was introduced by Fomin et al. [8]. According to Kneis et al. it is potentially useful, if the branching vertex has a rather low degree and thus, most likely has some mirror [12].

Definition 8. *(Mirror) In a graph $G = (V, E)$ a vertex u is called a mirror of a vertex v, if $u \in N^2(v)$ and $G_{N(v) \setminus N(u)}$ is a (possibly empty) complete graph, i.e. $N(v) \setminus N(u)$ is a (possibly empty) clique. The set of all mirrors of v is denoted by $\mathcal{M}(v)$ and $\mathcal{M}[v] := \mathcal{M}(v) \cup \{v\}$.*

Theorem 10. *(Mirror Branching) In a graph $G = (V, E)$, if there is no maximum independent set that contains a vertex v, then, every maximum independent set also excludes $\mathcal{M}[v]$.*

Proof. Consider any maximum independent set I. Then, I has to contain at least two neighbors of v because otherwise, we could get a maximum independent set $I' = (I \setminus N(v)) \cup \{v\}$ that includes v. Now let $u \in \mathcal{M}(v)$ be a mirror of v. Since $N(v) \setminus N(u)$ is a clique, I can only contain at most one vertex from $N(v) \setminus N(u)$. Thus, I contains at least another vertex from $N(v) \cap N(u)$ and therefore has to exclude u. □

So, when branching on a vertex v, the algorithm finds its mirrors $\mathcal{M}(v)$ and considers two possible cases. The first cases is that there is a maximum independent set that includes v and therefore exclude $N(v)$. And the second case is that no maximum independent set includes v. In this case, the vertices from $\mathcal{M}(v)$ can also be discarded from the graph.

The packing branching rule by Akiba and Iwata [1] is a generalization of the idea behind the satellite branching rule by Kneis et al. [12]. The core idea behind those rules is that when branching in the case of excluding a vertex v from the solution, one can assume that no maximum independent set contains v. Otherwise, if there is a maximum independent set that contains v, the algorithm finds it in the branch that includes v.
Based on the assumption that no maximum independent set includes a vertex v, constraints for the remaining vertices can be derived. For example, a maximum independent set that does not contain v has to include at least two neighbors of v. The corresponding constraint is $\sum_{u \in N(v)} x_u \geq 2$, where x_u is a binary variable that indicates whether a vertex is included in the current solution. The algorithm creates such constraints when branching, and updates them accordingly during the kernelization and branching steps. The constraints can then be used to reduce the graph or to prune the current branch when a constraint can not be fulfilled by the current solution.

3 Related Work

This section discusses related work. It focuses on presenting branching strategies used by other branch and reduce algorithms for maximum independent set or its equivalent problems minimum vertex cover and maximum clique.

There are various approaches to tackle the maximum independent set problem and its equivalent problems. These approaches comprise exact as well as heuristic methods. A technique that is frequently used for both exact and inexact algorithms is kernelization. We already covered the most important reduction rules used for kernelization in Section 2.

Due to the NP-Hardness of the maximum independent set problem, inexact algorithms have been well studied in practice. One of the best techniques used for finding large independent sets is local search [**?**]. Local search algorithms start with an initial solution and then utilize simple operations to iteratively improve the current solution. In practice, local search algorithms often find near optimal solutions very fast. However, most of them can not give any guarantee for the actual quality of a solution. One of the best local search algorithms for maximum independent set, called ARW, was proposed by Andrade et al. [**?**]. Their algorithm uses the concept of (j, k)-swaps, i.e. removing j vertices from the current solution and replacing them with k vertices instead. In every iteration of their algorithm, they perform a $(1, 2)$-swap to improve the current solution. To escape local optima, ARW occasionally disturbs the current solution by randomly inserting vertices into it and removing their neighbors instead.

Chang et al. [**?**] developed a linear time kernelization algorithm which reduces vertices of degree one and two. Another variant of their algorithm (`NearLinear`) additionally applies dominance and LP reduction rule but has only near-linear runtime. They also showed that applying their kernelization algorithm iteratively followed by removing vertices of high degree, is going to result in a large initial solution which, in turn, speeds up the ARW local search algorithm.

The theoretically most powerful exact exponential time algorithms for maximum independent set are branch and reduce algorithms. These algorithms use kernelization to compute a kernel of the input instance. After that, they branch into subinstances with lower complexity, which are then solved recursively. We now discuss the branching strategies used in various different branch and reduce algorithms.

The most common branching strategy used for maximum independent set and minimum vertex cover is branching on a vertex of maximum degree. Fomin et al. gave a theoretical analysis of this using the measure and conquer technique with a weighted degree sum as measure [8]. They showed that choosing a vertex of maximum degree that also minimizes the number of edges in its neighborhood is optimal with respect to their complexity measure. This greedy strategy is also used by the algorithm of Akiba and Iwata[1] and serves as a baseline for comparison in our experiments. Akiba and Iwata already compared this strategy with branching on a vertex of minimum degree and the strategy of choosing a branching vertex at random. Their experiments showed that those strategies are significantly worse than branching on maximum degree vertices.

Xiao and Nagamochi proposed a branch and reduce algorithm for maximum independent set that, in most cases, branches on a vertex of maximum degree but also uses a special edge branching strategy to handle dense subgraphs [21]. Edge branching is based on the principle of alternative subsets (like in alternative reduction). Given an edge $\{u, v\} \in E$ a maximum independent set can only contain u or v but not both of them. So, if there is a maximum independet set that includes u or v, then $\{u\}$ and $\{v\}$ are alternative sets. Thus, branching on the edge $\{u, v\} \in E$ yields two cases. The first

case is to remove both u and v and to search for a maximum independent set that does not include u and v. The second case is to compute the alternative reduction of $\{u\}$ and $\{v\}$, i.e., to remove $\{u, v\} \cup (N(u) \cap N(v))$ and insert an edge $\{x, y\}$ between any nonadjacent vertices $x \in N(u) \setminus N(v)$ and $y \in N(v) \setminus N[u]$ and to search for a maximum independent set that includes either u or v.
The algorithm by Xiao and Nagamochi uses edge branching in degree bounded graphs on edges $\{u, v\} \in E$, where $|N(u) \cap N(v)|$ is sufficiently large (the concrete values depend on the maximum degree of the graph).

Bourgeois et al. presented branch and reduce algorithm for maximum independent set that relies on fast algorithms for graphs with low average degree [2]. If the average degree of the graph is greater than 4, the algorithm branches on a vertex of maximum degree. Otherwise, if the average degree of the graph is at most 4, they use a specialized algorithm to solve the instance. If there is no vertex with degree of at least 5, this algorithm branches on vertices contained in 3- or 4-cycles.

Chen, Kanj and Xia developed a branch and reduce algorithm for the problem minimum vertex cover parameterized by the size k of the vertex cover, i.e., the problem of finding a vertex cover of size not larger than k [6]. In their algorithm, they use the concept of so called tuples and good pairs. A good pair is a pair of adjacent vertices that are advantageous for branching (the details are omitted here). A tuple is a set S of vertices together with the number of vertices in S that can be excluded from a minimum vertex cover. This information can be exploited during the branching to eliminate additional vertices. For example, consider the pair $(\{u, v\}, 1)$. We know that either u or v can be excluded from a minimum vertex cover and thus, if we include u to the vertex cover, we can exclude v. Otherwise, if we exclude u from the vertex cover, we can include v. Akiba and Iwata used the same idea in their packing reduction [1]. The algorithm by Chen, Kanj and Xia maintains a set of those structures as well as vertices of high degree and updates them accordingly during kernelization and branching. At each branching step the algorithm chooses the best structure and branches on it.

Most branch and reduce algorithms for maximum clique use some sort of greedy coloring to find an upper bound to the size of a maximum clique and also to reduce the number of possible vertices for branching. Given a coloring $c : V \to \mathbb{N}$ and the size $c_{\max}$ of a current best solution, it is easy to see that for $A = \{v \in V \mid c(v) \leq c_{\max}\}$, G_A can not contain a clique larger than the current best solution. Thus, only vertices from $V \setminus A$ are considered for branching.

More sophisticated algorithms use a MaxSAT encoding of maximum clique to achieve better upper bounds and to further reduce the set of branching vertices [13, 14]. Maximum clique can be reduced to MaxSAT by introducing a binary variable x_v for every vertex $v \in V$ and a hard clause $\overline{x}_u \vee \overline{x}_v$ for each pair of non adjacent vertices. The set $\{x_v \mid v \in V\}$ of unit literals forms the soft clauses. A solution to this MaxSAT instance yields a maximum clique where the value of a variable indicates whether the corresponding vertex is included in the clique or not. A more efficient MaxSAT enconding of maximum clique was introduced by Li and Quan [16]. Given a partition of V into independent sets (i.e. a coloring), instead of introducing a soft clause for each vertex, they merely formulate a single clause $\bigvee_{x_i \in I_j} x_i$ for each independent set I_j in the partition. Using this encoding they apply techniques from SAT solving like unit propagation and failed literal detection to identify conflicting independent sets. A set of t independent sets is called conflicting if there is no clique of size t in the subgraph induced by those independent sets. If conflicting independent sets are detected, the soft clauses get weakened by conjugating the clauses in a respective manner. Li and Quan show that if a Graph can be partitioned into k independent sets with t disjoint conflicting subsets, then the size of a maximum clique is bounded by $k - t$. This way, they achieve an upper bound which is often better than the bound obtained by the coloring.

Li et al. use a similar MaxSAT reasoning to reduce the number of vertices that are considered for branching [15]. They initially use a coloring to obtain a partition of V into parts A and $V \setminus A$ where A is defined as $A = \{v \in V \mid c(v) \leq c_{\max}\}$ and $c_{\max}$ is the size of the current best solution. After that they construct a MaxSAT instance where they add a soft clause for each color class (i.e. the set of vertices with the same color) in A. Then, they iteratively add a soft clause x_{v_i} for each vertex $v_i \in V \setminus A$ and apply unit propagation to it. If a conflict is detected, the affected soft clauses get weakened. This process is repeated until no more conflicts are detected or $V \setminus A$ becomes empty. Li et al. show that if there are conflicts for literals $x_{v_1}, \dots x_{v_k}$ with $v_i \in V \setminus A$, then the graph induced by $A \cup \{v_1 \dots v_k\}$ does not contain a clique larger than the current best solution $c_{\max}$. Thus, those vertices do not have to be considered for branching.

Another approach to decrease the number of branches in branch and reduce algorithms for maximum clique is to choose branching vertices in a specific beneficial order. A common strategy for choosing the branching vertex is to calculate a so called *degeneracy ordering* $v_1 < v_2 < \dots < v_n$ where v_i is a vertex of smallest degree in $G - \{v_1, \dots, v_{i-1}\}$, and to choose the vertices for branching in descending order [4]. Li et al. introduced another vertex ordering for branching using maximum independent sets [13]. While G is not empty, they repeatedly search for maximum independent sets and remove them from the graph. Then, the vertex ordering is defined in the following way using the degeneracy ordering for tie breaking: For two vertices u and v, $u < v$ if u has been removed later than v or if u and v have been removed at the same time but $u < v$ in the degeneracy ordering.

Most of the exact algorithms for maximum independent set and its equivalent problems have been analyzed on a theoretical level but are not very well tested on real instances. In practice, the best algorithms use a combination of multiple techniques. For example, the winning solver of the PACE challenge by Hespe, Lamm, Schulz and Strash [**?**] uses kernelization, iterated local search, a branch and reduce algorithm for vertex cover and a branch and bound algorithm for maximum clique. Their algorithm starts by applying a huge set of reduction rules to obtain a kernel as small as possible. After that, they use iterated local search to find a large initial solution and prime the branch and reduce algorithm with it. Subsequently, the branch and reduce algorithm is run for a short period of time. If no solution is found during that time, they run a branch and bound maximum clique solver on the complement of kernel and, thereafter, on the complement of the original instance. Again, if no solution has been found so far, they re-run the branch and reduce algorithm for a longer time followed by the clique solver if needed.

4 Branching Strategies

In this section we outline our approaches for designing the different branching strategies. We also give the details on the implementation itself and on the observations we made during the implementation. As mentioned in the introduction, we follow two main approaches in designing our branching strategies.

The first approach is to decompose the graph by branching. This way, the resulting connected components can be solved independently speeding up kernelization and, potentially, reducing the total size of the search space. For this thesis we implemented 3 branching strategies using this approach, which are described in Section 4.1.

The second approach is to destroy complex structures that can not be reduced by kernelization. Our main idea behind this approach is to identify vertices that prevent a certain reduction rule from being applicable, and subsequently branch on them. This way, the respective reduction rule can be applied afterwards, and the graph gets further reduced. Also, such vertices can be found during kernelization, which is continuously performed before every branching step. Therefore, the overhead of those branching strategies is rather small compared to the strategies following our first approach. We implemented branching strategies that target four different reduction rules and also tested a combination of those; branching strategies following this approach are covered in section 4.2.

For almost all of our branching strategies it is not guaranteed that they find a suitable vertex for branching in every branching step. Consequently, all of them use a default branching strategy as a fallback. In our implementation we used branching on a vertex with maximum degree that also minimizes the number of edges among its neighborhood as default branching strategy. This corresponds to the strategy proposed by Fomin et al. [8], and is already implemented in the algorithm by Akiba and Iwata [1].

4.1 Branching Strategies Based on Decomposition

Since the branch and reduce algorithm which we used as a our basis was designed for branching on a single vertex, our first idea is to find articulation points (i.e. cut vertices) of the graph and branch on them. Articulation points of a graph G are single vertices that form a separator of size one in G. Given any spanning tree of G, each articulation point v separates the vertices in the subtree rooted by a child of v from the rest of the graph. Thus, in a spanning tree obtained by a depth first search (DFS) there are no back edges from the graph induced by the subtree rooted at this child to the predecessors of v in the DFS tree.

Using this observation, articulation points in a connected graph can be found in linear time using the following algorithm (Algorithm 3) based on a depth first search scheme: The algorithm performs a depth first search starting at an arbitrary vertex of the graph. In every step of the DFS, the visited vertex is labeled with the current DFS number. If a back edge is found during the DFS run, the label of the currently visited vertex is updated to the minimum of the labels of both endpoints of the back edge. After the child of a vertex has been scanned by the DFS, the algorithm checks whether there were any back edges from the subtree rooted at the child to a predecessor of the vertex in the DFS tree. This is the case if the label of the child is not smaller than the current label of the vertex. If there are no such back edges, then the current vertex is an articulation point.

Eventually, the algorithm has to consider a special case for the root of the DFS tree (i.e. the start vertex). Obviously, there is no predecessor to the root vertex. However, the root can be an articulation point, too. Since there are no cross edges in a DFS tree, this is the case if and only if the root has more than one child. Thus, the root separates the subtrees rooted at the children from each other.

Algorithm 3: GetArticulationPoints

Input: A graph $G = (V, E)$

```
1  AP ← ∅
2  v ← u ∈ V                                              // pick arbitrary start vertex
3  currentDFSnum ← 1 ; rootDeg ← 0
4  ArticulationPoints(G, v, w) begin
5      if v = w then
6          rootDeg ← rootDeg + 1                                          // v is root
7      label(v) ← currentDFSnum
8      currentDFSnum ← currentDFSnum + 1
9      foreach u ∈ N(v) \ {w} do
10         if label(u) = ⊥ then
11             ArticulationPoints(G, u)                                   // {v, u} tree edge
12             label(v) = min{label(v), label(u)}
13             if label(u) > label(v) then
14                 AP ← AP ∪ {v}                  // no back edge: v is articulation point
15         else
16             label(v) = min{label(v), label(u)}                         // {v, u} back edge
17 if rootDeg < 2 then
18     AP ← AP \ {v}                                  // root is no articulation point
19 return AP
```

Output: the set AP of articulation points

In our first branching strategy (Algorithm 4) we manage a set of articulation points of the graph. During a branching step we first remove vertices from the set that are no longer contained in the graph (e.g. vertices removed by kernelization). Subsequently, we check if the set still contains any articulation points. If so, we remove a vertex from the set and return it for branching. Otherwise, we use Algorithm 3 to find the articulation points of the graph and insert them into the set. If there are no articulation points we use the default branching strategy as fallback.

Although this branching strategy has only little overhead, a major drawback is that articulation points are rarely found even in sparse graphs. Thus, in practice, the fallback strategy is used most of the time. So, we further develop our initial idea by also considering more general vertex separators, i.e. minimal separators that contain more than one vertex.

Since minimum edge cuts are generally easier to find than minimum vertex separators and can also be used to obtain small vertex separators, we opted to use edge cuts instead. (In fact, finding a minimum vertex separator can be reduced to finding a minimum edge cut in a transformed graph). A disadvantage of this approach is that vertex separators induced by minimum edge cuts are not necessarily minimal. Nevertheless, for our purpose the trade off between separator size and computation time may be worth it. Given an edge cut, a vertex separator can be obtained easily by just taking one of the incident vertices to each edge in the cut set. However, in our implementations we use a slightly more sophisticated method which yields potentially smaller vertex separators. Given an edge cut (S, T) with cut set C, we construct an auxiliary graph $G' = (V', C)$ with $V' = \{x, y \in V \mid \{x, y\} \in C\}$. By construction, G' is bipartite with parts $A = V' \cap S$ and $B = V' \cap T$. Thus, we can run the Hopcroft-Karp algorithm on G' to obtain a minimum vertex cover S' of G'. Since each edge of the cut set is incident to at least one vertex in the vertex cover, S' is indeed a vertex separator. Clearly S' has at most as many vertices as there are edges in C.

Algorithm 4: ArticulationPointsBranching

Input: A graph $G = (V, E)$

```
1  ArticulationPointsBranching(G) begin
2      foreach u ∈ AP do
3          if u ∉ V then
4              AP ← AP \ {u}                    // remove vertices no longer contained in G
5      if AP = ∅ then
6          AP ← GetArticulationPoints(G)        // search new articulation points in G
7      v ← none
8      if AP = ∅ then
9          v ← MaxDegBranching(G)                                  // use default branching
10     else
11         v ← u ∈ AP ; AP ← AP \ {u}           // use any articulation point for branching
12     return v
```

Output: a vertex v for branching

Besides that, the most crucial part of our next branching strategy is the calculation of the actual cut. Altogether, we test four methods for finding small edge cuts. For our first attempt we use a heuristic algorithm for global minimum cuts by Henzinger et al. [**?**]. Unfortunately, during implementation we noticed that searching a global minimum cut in our benchmark instances almost always results in a trivial cut with a part that only contains a vertex of minimum degree.

Hence, our next approach is to use s-t-cuts instead. This naturally raises the question which vertices should be used for s and t. At first, we try choosing s and t at random. However this frequently results in rather large and unbalanced cuts. Our next attempt is to use a pair of vertices that are as far apart as possible. The idea behind this is that this procedure might produce more balanced cuts. In our implementation we realized this by running a breadth first search (BFS) twice. The first BFS run is started at an arbitrary vertex, and the last vertex visited by the BFS is used for s. After that we start another BFS at s letting the vertex visited last become t. But choosing s and t this way also results in very unbalanced cuts similar to the global minimum cuts. This is due to the fact that the selected vertices almost always have low degree. Using the two vertices of highest degree as s and t delivers the best results on our benchmark instances of all three variants.

To calculate the actual minimum s-t-cut we use a preflow push maximum flow algorithm. Finally, it has to be considered how to branch on a vertex separator that contains more than one vertex. In our implementation we decide to branch on each vertex one after another.

Bringing all together, our second branching strategy (Algorithm 6) manages a set of branching vertices contained in the vertex separator currently. If the set is not empty, the branching strategy just removes and returns a vertex from that set. Otherwise, it searches a new vertex separator (Algorithm 5). For this purpose, we first retrieve the two vertices s and t in G with highest degree. After that, we use the preflow push algorithm to obtain a minimum s-t-cut. Using this cut, the branching strategy calculates a vertex separator by constructing the bipartite auxiliary graph induced by the cut set and then applies the Hopcroft-Karp algorithm to it. The new vertex separator, i.e. the minimum vertex cover returned by the Hopcroft-Karp algorithm is then inserted in the set of branching vertices and one of those vertices is returned.

An important optimization is that the branching strategy only considers vertex separators which are not larger than a certain size and at the same time meet certain balancing constraints. Furthermore, we noticed that if no suitable vertex separator is found, this will also happen in the next couple

of branches. For this reason, the branching strategy only searches for a vertex separator every few branching steps to reduce the overhead using the default fallback during this phase. The exact numbers used in our implementation are tuning parameters. The details of choosing those tuning parameters are discussed in Section 5 and are omitted in the pseudo code.

Algorithm 5: GetSeparatorFromEdgeCut

Input: A graph $G = (V, E)$

```
1  GetSeparatorFromEdgeCut(G) begin
2      s ← MaxDegVertex(G)
3      t ← MaxDegVertex(G − {s})
4      (S, T) ← MinCut(s, t)                          // utilizes a max flow algorithm
5      if |S|/|V| < MinBalance ∨ |T|/|V| < MinBalance then
6          return ∅                                    // cut to unbalanced
7      G' ← ConstructAuxiliaryGraph(S, T)
8      U ← HopcroftKarp(G')
9      if |U| > MaxSeparatorSize then
10         return ∅                                    // separator to large
11     return U
```

Output: a set of vertices for branching

Algorithm 6: CutBranching

Input: A graph $G = (V, E)$

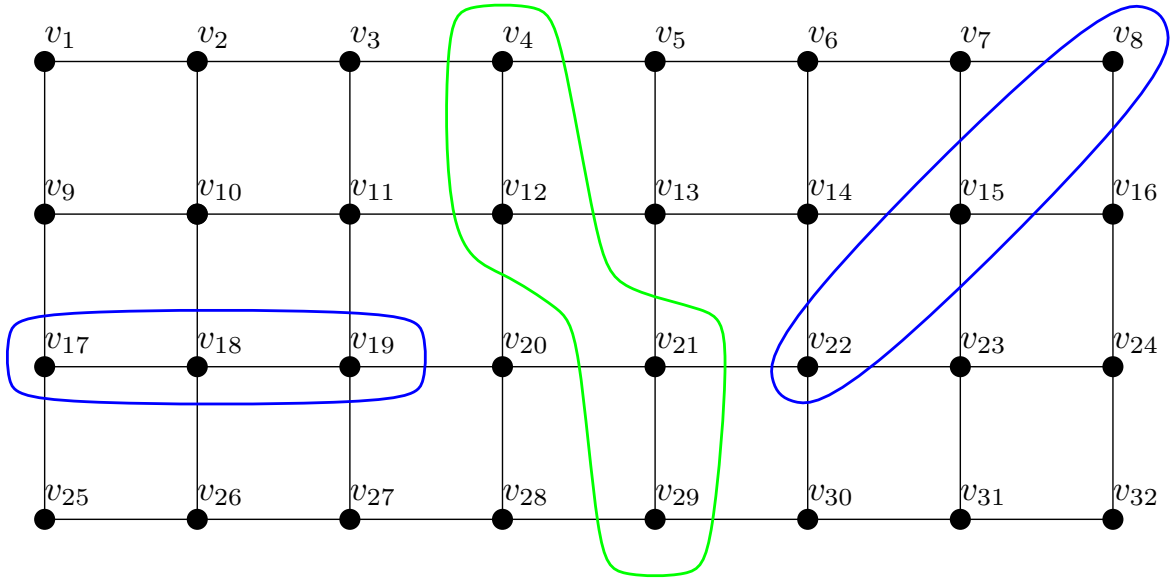
```
1  CutBranching(G, v) begin
2      foreach u ∈ BV do
3          if u ∉ V then
4              BV ← BV \ {u}          // remove vertices no longer contained in G
5      if BV = ∅ ∧ NumFallbacks = SearchFrequency then
6          BV ← GetSeparatorFromEdgeCut(G)
7          NumFallbacks ← 0                            // reset fallback counter
8      v ← none
9      if BV ≠ ∅ then
10         v ← u ∈ BV ; BV ← BV \ {u}          // use any vertex of the separator
11     else
12         v ← MaxDegBranching(G)
13         NumFallbacks ← NumFallbacks + 1
14     return v
```

Output: a vertex v for branching

Both of our branching strategies so far perform the calculation of the branching vertex dynamically in every branching step. This has the advantage that branching vertices are always chosen based on the current graph, but comes with the disadvantage of computation overhead in every branching step. Thus, our next approach is to use a static ordering, in which vertices are considered for branching and which is calculated only once before the first branching step. For this purpose we decided to use nested dissection ordering.

Nested dissection ordering is mainly applied as a heuristic to minimize the number of fill ins in factorization of sparse symmetric matrices, or for computing good contraction hierarchies in route planning. Nested dissection ordering of the vertices of a graph G is obtained by calculating a balanced bipartition of the graph into parts A and B using a vertex separator S. Subsequently, orderings of G_A and G_B are calculated recursively. Finally, the nested dissection ordering of G is composed by concatenating the orderings of A and B followed by the separator S. For vertices of the separator S, the relative order among each other is arbitrary. By choosing the branching vertices in reverse nested dissection ordering, the algorithm branches on the separators used for the bipartition at each level of the nested dissection. Consequently, the graph is getting decomposed piece by piece.

$G = (V, E)$:



Nested dissection ordering of G :

1 2 3 9 10 11 | 20 25 26 27 28 | 17 18 19 | 5 6 7 13 14 | 16 23 24 30 31 32 | 8 15 22 | 4 12 21 29

part 1 part 2 part 3 part 4

Sizes Array : parts separators

6	5	5	6	3	3	4	⊥

Branching order : 4 12 21 29 | 17 18 19 | 8 15 22

Figure 1: Possible nested dissection ordering of the 7 by 4 grid

In the implementation of our following branching strategy (Algorithm **??**) a nested dissection ordering is computed immediately prior to the first branching step, i.e., after the initial kernelization step. To calculate the nested dissection ordering, we use an algorithm provided in the METIS library [**?**]. We do not apply a custom configuration to the algorithm, but rather use the default settings of the METIS library. Afterwards, the vertices are inserted into a queue in reverse nested dissection ordering.

In each branching step, the branching strategy removes vertices from the queue until it finds a vertex that is still contained in the current graph. Finally, the respective vertex is returned for branching.

During initial tests of our implementation we noticed that branching on separators which come later in the reversed nested dissection ordering (i. e., separators obtained in a recursive call of higher depth), frequently does not result in decomposition of the graph. This is due to the fact that in addition to branching, the graph is also reduced by kernelization in each step of the algorithm. Thus, separators in the original graph are not necessarily separators in the current graph at the time of branching. Counteracting this, we optimized our branching strategy by performing a restricted number of levels of recursions instead of calculating a full nested dissection ordering. Then, the branching order is composed solely by the separators obtained in those levels of recursion. In a branching step, if there are no more vertices left in the ordering, the default branching strategy is used as a fallback. The exact number of recursions used is a tuning parameter. Details are discussed in the following section (Section 5).

To implement this optimization, we used an alternate method provided by the METIS library, which just performs a specified number of resursive calls of the nested dissection algorithm. Nevertheless, a problem remaining is that the algorithm still returns an ordering of all vertices of the graph. We are, however, only interested in the vertices contained in the separators of each recursive call. Luckily, the METIS library provides an array that stores the sizes of the parts as well as the separators for each level of recursion. The following recursive algorithm (Algorithm 7) capitalizes on this feature by extracting the individual separators from the vertex ordering and constructing the branching ordering in the following manner.

Algorithm 7: ExtractSeparators

Input: A vertex ordering $\sigma_1 \dots \sigma_n$ and sizes array s obtained by `METIS_NodeNDP`

```
1  ExtractSeparators(σ, s) begin
2      BranchingOrder ← ⊥
3      TopLevelSeparatorSize ← s[2p − 2]
4      foreach u ∈ BV do
5          if u ∉ V then
6              BV ← BV \ {u}                    // remove vertices no longer contained in G
7      if BV = ∅ ∧ NumFallbacks = SearchFrequency then
8          BV ← GetSeparatorFromEdgeCut(G)
9          NumFallbacks ← 0                                  // reset fallback counter
10     v ← none
11     if BV ≠ ∅ then
12         v ← u ∈ BV ; BV ← BV \ {u}              // use any vertex of the separator
13     else
14         v ← MaxDegBranching(G)
15         NumFallbacks ← NumFallbacks + 1
16     return v
```

Output: a branching order

The algorithm receives the vertex ordering and the sizes array as input. Initially, the branching order is empty. In each recursive call the algorithm starts by retrieving the size k of the top level separator from the sizes array. Thereafter, the top level separator, which just contains the last k vertices of the vertex ordering, is inserted into the branching order. Then, the algorithm uses the part and separator

sizes to split the rest of the vertex ordering and the sizes array in accordance to the recursion. This is going to result in orderings for each part of the bipartition and the two corresponding sizes arrays. Subsequently, the separators residing in those parts are extracted recursively and are added to the branching order.

Just as in the previous branching strategy using edge cuts, we optimize our implementation by considering a vertex ordering only if the sizes of the vertex separators computed at each level of recursion are not exceeding a certain threshold. The exact value of the threshold is a tuning parameter. In contrast to the previous branching strategy, we did not tweak the balancing constraints used by the nested dissection algorithm.

Another optimization we tested was to compute a new nested dissection ordering (with restricted levels of recursion), following removal of all vertices of the previous ordering from the graph. This way, we attempt to combine the advantage of using a static branching ordering with choosing vertices dynamically on the current graph. However, tests showed that this variant of the branching strategy performs worse than computing the vertex ordering only once. A possible explanation is that branches at a lower depth of recursion have a greater impact on the total number of branches needed than such at a higher depth. Hence, decomposing the graph at an early stage in the algorithm is more powerful than at later stages. An alternate reason might be that branches at a higher depth of recursion are more likely to be pruned before all vertices of a separator have been removed by branching.

4.2 Branching Strategies Based on Reduction Rules

The branching strategies described in this section are essentially based on the destruction of structures that can not be reduced by kernelization. The core idea behind this is to identify vertices such that branching on those enables the application of reduction rules afterwards. Our initial approaches using this concept are rather simple and attempt to find single vertices that prevent a certain reduction rule from being applicable. It is generally advantageous that finding of such vertices can be accomplished during kernelization without noticeable time overhead.

Following this idea, we implemented and tested four branching strategies each targeting a different reduction rule. Finding of potential candidates for branching, i.e. vertices that prevent the corresponding reduction rule from being applicable, works differently depending on the targeted reduction rule. However, the actual branching step is the same for all these variants. In a branching step, a vertex for branching is chosen from a set of vertices managed by the respective branching strategies. Vertices considered for branching are inserted into this set during kernelization. Notice, that at the time of branching, vertices in the set could have already been removed from the graph by another reduction rule. Thus, reduced vertices have to be filtered out first. Subsequently, if the set of branching vertices is not empty, all four branching strategies choose a vertex of highest degree from the set and return it for branching. Otherwise, if there are no vertices in the set, the default branching strategy is used as a fallback.

As in the case of the branching strategies described previously (Section 4.1), our implementations are optimized to the effect that even if the set of potential branching vertices is not empty, only vertices of a certain degree are considered for branching. Details on how the tuning parameter is chosen are explained in Section 5.

In the following, we describe for each of the four branching strategies how vertices considered for branching are found during kernelization.

The first branching strategy in this subsection is based on the Twin reduction rule (covered in Section 2.3). To characterize the structure in which a single vertex prevents the twin reduction from being applicable, consider the following definition.

Definition 9. *(Almost Twins) In a graph $G = (V, E)$ two non adjacent vertices u and v are called almost twins if $d(u) = 4$, $d(v) = 3$ and $N(v) \subseteq N(u)$ (i.e. $N(u) = N(v) \cup \{w\}$).*

Clearly, after removing w, it holds that $d(u) = d(v) = 3$ and $N(u) = N(v)$. Thus, by removing w, the vertices u and v become twins. Consequently, the twin reduction rule is applicable afterwards. Thus, when using this branching strategy, the branch and reduce algorithm searches for pairs of vertices that are almost twins during each kernelization step. Finding those vertices can be done while already searching for twins. The twin reduction rule checks for each vertex v of degree three if there is a vertex $u \in N^2(v)$ such that $d(u) = 3$ and $N(u) = N(v)$. To find almost twins, we modify this routine in order to simultaneously check if there is a vertex $u \in N^2(v)$ with $d(u) = 4$ and $N(v) \subseteq N(u)$. If such a pair of vertices u and v is found, the single vertex $w \in (N(u) \cup N(v)) \setminus (N(u) \cap N(v))$ is inserted into the set of vertices considered for branching.

For the second branching strategy we attempt to find vertices preventing application of the alternative reduction. As mentioned in Section 2.3, the branch and reduce algorithm only utilizes the special cases Funnel and Desk reduction, hence providing a further restriction in regards to eligible reduction rules. However, desk reduction is based on a very concrete graph structure and therefore is rarely applied in our benchmark instances (in contrast, funnel reduction is applied relatively often). Also, it is difficult to determine if a vertex prevents the desk reduction from being applicable, since a multitude of cases would have to be considered. For those reasons, we do not target the desk reduction but exploit the funnel reduction rule, instead.

Similar to the first branching strategy, we define the structure within which the funnel reduction can be applied upon removing a single vertex.

Definition 10. *(Almost Funnel) In a Graph $G = (V, E)$ two adjacent vertices u and v are called almost funnels if u and v are not funnels and there is a vertex w such that $N(v) \setminus \{u, w\}$ induces a clique.*

By removing the single vertex w, u and v become funnels and, thus, u can be reduced afterwards. Pairs of vertices that are almost funnels can be found easily during the funnel reduction. To check whether two vertices u and v are funnels, the funnel reduction rule iterates over the vertices in $N(v) \setminus \{u\}$ and checks if they are adjacent to the previous vertices in the iteration. Once it finds a vertex that is not adjacent to all prior vertices, the reduction rule concludes that u and v are not funnels and stops checking the remaining neighbors of v. To find almost funnels, we modify this procedure as follows. We still iterate over $N(v) \setminus \{u\}$ and check for adjacency. Once a vertex w is found that is not adjacent to all prior vertices, two cases are considered. Firstly, if w is not adjacent to at least two preceding vertices, then, u and v representing almost funnels can be verified by checking if $N(v) \setminus \{u, w\}$ induces a clique. In the second case there is only one vertex w' prior to w such that w' and w are not neighbors. Then, u and v are almost funnels if $N(v) \setminus \{u, w\}$ or $N(v) \setminus \{u, w'\}$ induces a clique. In both cases, if u and v are almost funnels, the neighbor of v, which is not contained in the induced clique (i.e. w or w' respectively), is inserted into the set of vertices considered for branching.

The third branching strategy is making use of the dominance reduction rule. Again, we define the structure which contains a vertex enabling dominance reduction upon removal.

Definition 11. *(Almost Dominance) In a Graph $G = (V, E)$ a vertex u is called almost dominated by a neighbor v if $|N(u) \setminus N(v)| = 1$ and $N[v] \not\subseteq N(u)$.*

The condition $N[v] \not\subseteq N(u)$ ensures that v is not already dominated by u. By removing the single vertex $w \in N(u) \setminus N(v)$, the vertex u becomes dominated by v, since $(N[u] \setminus \{w\}) \subseteq N(v)$. Similar to the previous branching strategy, we search for pairs of vertices u and v such that u is almost dominated by v. The dominance reduction verifies whether a vertex u is dominated by v through checking if all neighbors of u are also neighbors of v. Once it finds a vertex that is adjacent to u but not a neighbor of v, it is concluded that u is not dominated by v and the remaining vertices are not going to be checked anymore. We modify the dominance reduction to test if the neighbors of u are adjacent to v until either all neighbors have been checked or, alternatively, a second vertex adjacent to u is found that is not also a neighbor of v. When all vertices have been checked and there is no vertex $w \in N(u) \setminus N(v)$, then u is dominated by v. Otherwise, w is inserted into the set of vertices considered for branching.

A fundamental difference to the two preceding branching strategies is that the dominance reduction rule is not actually used by the branch and reduce algorithm. Nevertheless, dominance reduction is fully contained in the unconfined reduction rule used by the algorithm so that dominating vertices still get reduced. It is a major drawback, however, that vertices considered for branching making the dominance rule become applicable, can not be found during kernelization. Instead, one has to search for such vertices separately after kernelization resulting in a non-negligible time overhead.

Therefore, in the fourth branching strategy we generalize the previous strategy in order to exploit the unconfined reduction rule. We define an almost unconfined vertex analogously to the previous definitions.

Definition 12. *(Almost Unconfined) In a graph $G = (V, E)$ a vertex v is called almost unconfined if v is not unconfined but there is a vertex u such that v is unconfined in $G - u$.*

By definition, v becomes unconfined after removing u and, thus, by means of the unconfined reduction rule can also be removed. However, it is not clear how to determine whether a vertex u is almost unconfined and, moreover, how to find the specific vertex u preventing v from being unconfined. There are basically two cases to be considered. Firstly, at some point during the execution of the unconfined algorithm (Algorithm 2) there is an extending child, i. e., a child w with $\{u\} = N(w) \setminus N[S]$, and, coincidentally, inclusion of u into the set S leads to v not being unconfined. By removing u, vertex w becomes a child devoid of neighbors not being already contained in $N[S]$. Thus, v becomes unconfined. In the second case, at the end of the unconfined algorithm there is a child w with $\{u, x\} = N(w) \setminus N[S]$. Upon removing u, the vertex w becomes an extending child. Therefore, the unconfined algorithm has to include x into the set S and eventually concludes that v is unconfined.

It is easy to check whether the first case occurs. In each step of the unconfined algorithm, if there is only one extending child w with $\{u\} = N(w) \setminus N[S]$, we insert u into a buffer. When the algorithm terminates, the following condition applies. If it returns false and the buffer is not empty, then v is almost unconfined and removal of any vertex from the buffer makes v unconfined. Hence, all vertices from the buffer can be considered for branching.

Notably, there is an obstacle in detecting the second case. If the unconfined algorithm concludes a vertex v is not unconfined, we can in fact check if at the end of the unconfined algorithm there is a child w with $\{u, x\} = N(w) \setminus N[S]$; but we do not know if removal of either u or x is going to result in v becoming unconfined. Hence, in this case we can not easily check whether v is almost unconfined or not.

Thus, during kernelization almost unconfined vertices can only be identified if case one applies. Unfortunately, it is not clear whether this is true if a vertex u is almost dominated by another vertex v. It holds that v is either unconfined (then v is reduced anyway) or, alternatively, v is almost unconfined since by removing the single vertex $w \in N(u) \setminus N(v)$, u becomes dominated by v; hence, v becomes unconfined. Also, during the unconfined procedure, u is an extending child as long as the set S does not contain $w \in N(u) \setminus N(v)$ following from $N(u) \setminus \{w\} \subseteq N(v)$. Hence, if u is the

only extending child at any point during the unconfined algorithm, then case one applies. However, there can also be another extending child being considered prior to u by the unconfined algorithm, consequently forcing w into the set S. In this instance, u is no longer a child of S and, eventually, case two might prevail. Moreover, if a vertex v is almost unconfined, there can also be a vertex such that removing the latter results in v being neither unconfined nor almost unconfined anymore. As an example see Figure **??**. Notably, this vertex can also be removed during a branching step by the mirror branching rule (covered in Section 2.3).

During initial test runs using the first four branching strategies covered in this section, we observed that branching on a vertex may result in a whole series of reductions. There are two main causes for this. First, within one branching step multiple vertices frequently get removed from the graph. Specifically, this is the case when the branching vertex is included into the current solution and therefore its neighbors are excluded or, alternatively, when the branching vertex has mirrors (see Mirror branching rule in Section 2.3). Then, each of the removed vertices might enable a reduction during the following kernelization. The second cause is that removal of a vertex during branching might lead to a reduction which, in turn, enables further reductions.

We make use of this observation in our fifth branching strategy. Here, the main idea is to choose a vertex with the effect that branching on this vertex is going to trigger a largest possible chain of reductions. Consider the following example shown in Figure **??**.

To find large reduction chains, we define the following directed graph $R = (V, E')$, where V is just the vertex set of the original graph; further there is an edge from a vertex u to another vertex v if the removal of u causes the reduction of v based on a single reduction rule (we omit transitive edges). We call this a reduction graph. Given the reduction graph R, the number of vertices, which can be reduced after removing a vertex v, at best corresponds to the number of vertices reachable from v in R. We can compute this number with a simple BFS run starting at v.

Unfortunately, in practice the exact reduction graph is hard to construct as there are a few pitfalls. We can in fact use the results of our previous branching strategies to identify vertices that enable the reduction of other vertices upon their removal so that parts of the reduction graph can be constructed. However, in doing so, we do not consider all reduction rules used by the branch and reduce algorithm. Also, we disregard the order in which the reduction rules are applied. Since the reduction rules are not executed iteratively but rather in a fixed order, it is therefore not guaranteed that the whole chain of reductions is performed within one kernelization step. Folding reductions (e.g., degree two folding or twin reduction, respectively) add new vertices while alternative reductions introduce new edges between existing vertices. Consequently, those reductions might prevent other reductions. Moreover, Somebody et al. [Quelle fehlt noch] show that the order reduction rules are applied may affect the kernel size. Thus, the number of vertices reachable in the reduction graph starting at a specific vertex does not necessarily correspond to the exact number of reductions caused by removal of that vertex.

In our implementation, we only consider almost dominated and almost unconfined vertices for constructing an approximation to the reduction graph, as the unconfined reduction does not introduce new vertices or edges. We use the modified variants of the dominance and unconfined reduction to find vertices that enable reductions upon removal. In a branching step, initially, the reduction graph is constructed. Subsequently, we perform BFS runs starting at each vertex of the graph to compute the number of vertices reachable. These numbers are utilized to find the vertex starting from which the highest number of vertices are reachable. The same vertex is finally used for branching if it has a certain degree.

5 Experimental Results and Conclusions

In this section we evaluate our branching strategies by testing them using a set of benchmark instances from multiple graph classes. We start with an explanation of our testing methodology. Then, we outline our approaches for tuning the various parameters. Finally, we compare the branching strategies to each other and discuss the results with respect to their effectiveness.

5.1 Experimental Setup

In our experiments we use a C++ implementation of the branch and reduce algorithm by Akiba and Iwata [1] as a basis. For each branching strategy we extend the existing source code with the respective implementation as explained in Section 4. The code is compiled using gcc 4.8 with full optimizations (-O3 flag) enabled. We run all our tests on a machine equipped with four Intel Xeon E5-46408-core processors clocked at 2.4 GHz and 512 GiB of ECC DDR3 RAM. The installed operating system is Ubuntu version 20.04.1 LTS running the Linux Kernel 5.4.0-42-generic. Since all of our branching strategies as well as the actual branch and reduce algorithms run single threaded, we made full use of the cores of our machine by executing multiple tests simultaneously with GNU Parallel [**?**].

For testing the various branching strategies, we choose a similar benchmark set as Akiba and Iwata did for their original algorithm [1]. This set consists of real world sparse network, complements of DIMACS instances and instances from the *Odd Cycle Traversal* (OCT) problem. We transform the latter to vertex cover instances in the following way. Given an OCT instance $G = (V, E)$, we construct a vertex cover instance $G' = (V', E')$ with $V' = \{l_v, r_v \mid v \in V\}$ and $E' = \{\{l_u, l_v\}, \{r_u, r_v\} \mid \{u, v\} \in E\} \cup \{\{l_v, r_v\} \mid v \in V\}$. Moreover, we use some of the public instances from the PACE 2019 implementation challenge's vertex cover track. Since our focus is on comparing branching strategies, we only consider instances that require at least 50 branches using the default strategy. Furthermore, we omit instances that did not finish within 24 hours in the experiments of Akiba and Iwata. A list of all benchmark instances along with their sources can be found in the Appendix (Table **??**).

We run all tests with a time limit of 24 hours per instance. In each test run, we track the time and total number of branches needed to solve an instance as well as the number of decompositions and fallbacks to the default strategy. Most of our branching strategies are not optimized in terms of runtime. Therefore, we further consider the total number of branches in order to comparatively evaluate our strategies. Since the absolute values for both the runtime and the total number of branching steps vary greatly between the individual instances, we compare the respective measurements relative to a baseline measurement. This way, results from different benchmark instances can be put in relation to each other.

During initial testing we noticed only negligible deviations in the runtime of an instance when multiple test runs were performed. In the case of the branching strategy using nested dissection, we made similar observations for the number of branching steps. This follows from the fact that the METIS library uses non deterministic methods in its nested dissection algorithms. Therefore, we repeat all tests three times per instance in order to obtain reliable measurements. The final test results are obtained by taking the average of those measurements.

5.2 Parameter Tuning

Prior to evaluating the effectiveness of our branching strategies, it has to be ensured that tuning parameters have been chosen in a meaningful way. For this purpose, we conduct a series of experiments in which parameters are systematically varied in order to find the best possible combination of values. In each experiment we test a different assignment of tuning parameters with our benchmark instances. However, we ultimately use a subset of the benchmark instances comprising representatives from

all graph classes. This way, we avoid over-fitting of tuning parameters in regards to our specific benchmark set. A list of all benchmark instances used for parameter tuning can be found in the Appendix (Table **??**). Concrete values for the tuning parameters are selected from a domain that we choose based on observations we have made during implementation and initial testing, i. e., based on a start configuration that produces reasonably good results. To find the best combination of values we calculate the geometric mean over the measurements of all instances obtained with a certain configuration. We choose the geometric instead of the arithmetic mean as no absolute differences are determined. Instead we aim at comparing the ratio of the runtime and the total number of branching steps relative to the default branching strategy.

For optimization of the strategy based on edge cuts, three different parameters were selected for tuning.

- Maximum cardinality of a vertex separator (obtained from an edge cut)
- Balance of an edge cut, i.e., the minimum percentage of vertices that have to be contained in each part of the cut
- Frequency of searching edge cuts on branching, i.e., the number of fallbacks to the default branching strategy between two searches if no suitable edge cut has been found

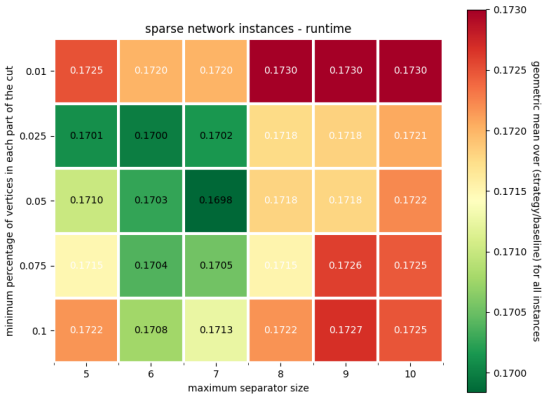
Since the total number of possible value combinations grows exponentially with the number of parameters, we do not optimize all three tuning parameters simultaneously. Instead, we start by optimizing only two of them with a fixed value for the third one. Once we find the optimal combination for those two parameters, we tune the third one independently. Note that in this case the first two parameters, i. e., maximum separator size and cut balance, both determine whether the separator is considered for branching or not. Also, we noticed that the third parameter, i.e, search frequency, may have a great impact on the runtime if rather few suitable cuts are found. In this case, however, the impact on the total number of branches is very small. Therefore, we tune the first two parameters separately from the third.

During implementation we tried different tradeoffs between separator size and cut balance. More specifically, we tested branching on small unbalanced cuts versus branching on larger, more balanced cuts. However, the latter was not successful, because we solely encountered rather unbalanced separators with separator sizes below ten, and parts that contained less than ten percent of all vertices. Based on these observations, we chose the maximum separator size from the set $\{5, 6, 7, 8, 9, 10\}$ and the minimum percentage of vertices that have to be contained in each part of the cut from the set $\{0.01, 0.025, 0.05, 0.075, 0.1\}$. In all experiments, in which those two parameters are tuned, we fix the third parameter to 5. After finding the optimal combination of the first two parameters we test the remaining values for the third parameter from the set $\{1, 5, 10, 15, 20\}$.

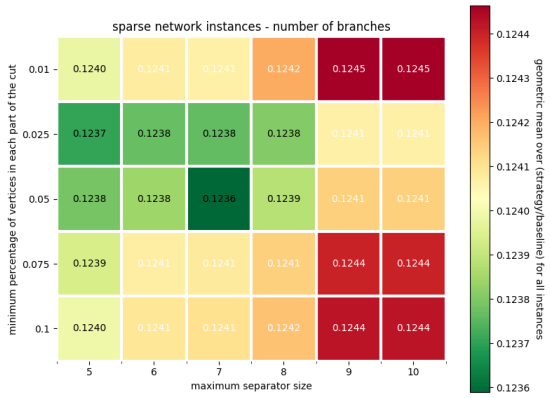
Following the concept of tuning the first two parameters, we could show that the branching strategy using edge cuts is performing generally well on real world sparse networks. The experiments also confirmed our observation that cuts with a balance of at least ten percent of the vertices in each part are almost never found. Surprisingly, considering separators with a size larger than eight has a non negligible impact on the runtime despite only very few cuts of such separator sizes being encountered.

On DIMACS and PACE instances the branching strategy performs worse than default branching. This is due to the fact that in most of the cases unbalanced cuts with parts containing less than 2.5 percent of all vertices are found. On theses instances the size of the separator has almost no influence on the runtime what so ever. An explanation for this can be found as follows. Based on the constraint that both parts of the cut must contain at least 2.5 percent of the nodes, the fallback strategy will be applied most of the time. Cuts with less than 2.5 percent of the vertices in one part, on the

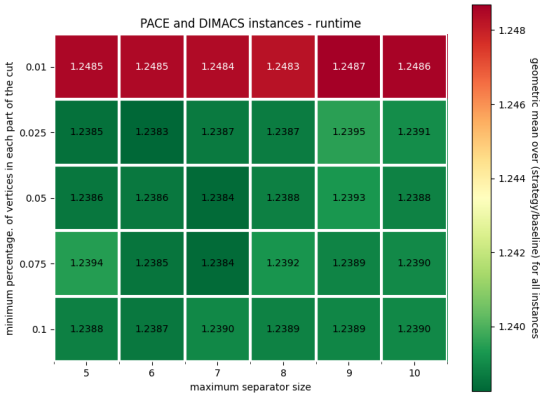
contrary, often yield small separators. Hence, branching on such cuts is not beneficial. Overall, the best performance is obtained by choosing the value 7 for the maximum separator size and 0.05 for the balance of a cut.



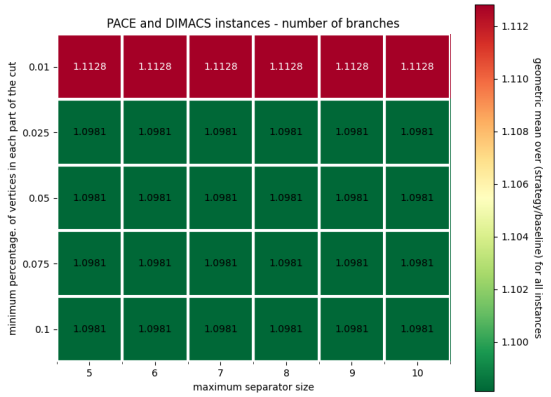
(a) sparse networks - runtime



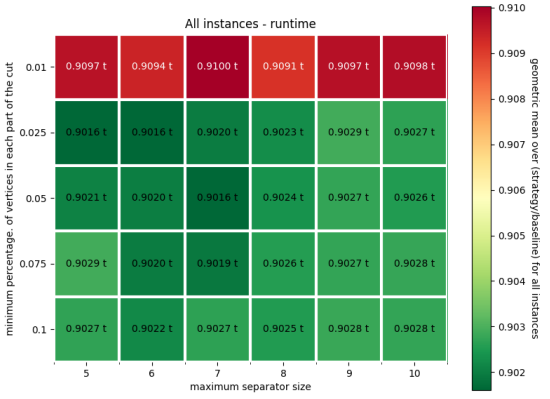
(b) sparse networks - number of branches



(c) PACE and DIMACS - runtime



(d) PACE and DIMACS - number of branches



(e) All instances - runtime

(f) All instances - number of branches

Figure 2: Experimental results tuning the maximum separator size and the balance of an edge cut considered by the branching strategy utilizing edge cuts.

Results of tuning the third parameter, i.e. frequency cuts are being searched, are shown in Table 1 and 2. However, tuning the third parameter does not give any new insight. Obviously, on sparse networks, where the branching strategy performs very well, it is advantageous to search for edge cuts in each branching step. On the other instances, however, best performance is achieved by setting the highest possible value for the third parameter, as suitable cuts are rarely found. Therefore searching for cuts with a high frequency produces unnecessary overhead. Considering all instances used for tuning, searching for edge cuts every 10 branching steps (if no suitable edge cuts are found) yields the best outcome.

	Frequency of searching edge cuts				
Instances	1	5	10	15	20
PACE	1.1437	1.0733	1.0565	1.0539	**1.0537**
DIMACS	1.1542	1.0564	1.0325	1.0253	**1.0215**
sparse networks	**0.0867**	0.1239	0.1698	0.1743	0.1742
All instances	1.0019	0.9976	**0.9966**	0.9969	0.9972

Table 1: Runtimes tuning the frequency of searching edge cuts

	Frequency of searching edge cuts				
Instances	1	5	10	15	20
PACE	**1.0981**	**1.0981**	**1.0981**	**1.0981**	**1.0981**
DIMACS	**1.0051**	**1.0051**	**1.0051**	**1.0051**	**1.0051**
sparse networks	**0.0867**	0.1007	0.1239	0.1279	0.1281
All instances	1.0063	0.9963	**0.9962**	1.0062	1.00625

Table 2: Number of branches tuning the frequency of searching edge cuts

Next, we optimize the branching strategy based on nested dissection. As already mentioned in Section 4, we do not tune the settings of the actual nested dissection algorithm. Instead, we selected the following two parameters for tuning.

- Maximum cardinality of the vertex separators at each level of recursion for a branching order
- Number of recursions performed by the nested dissection algorithm.

The number of recursions does not influence the sizes of the separators found at each level of recursion by the nested dissection algorithm. For this reason we decide to tune both parameters separately in order to reduce the number of experiments needed.

Since the nested dissection algorithm from the METIS library uses balanced bipartitions, the calculation of a complete nested dissection ordering using METIS requires $\mathcal{O}(\log n)$ levels of recursion where n denotes the remaining number of vertices in the current graph when calculating the ordering. However, as explained in Section 4, it is advantageous to perform a restricted number of recursions, only. Therefore, we choose the number of levels of recursion from the set $\{2, 3, 4, 5\log n - 6, \log n - 5, \log n - 4, \log n - 3\}$. For the parameter relating to the maximum separator sizes we also tried constant values as well as thresholds depending on the number of vertices in the initial kernel of the instance (i.e. the graph right after the first kernelization step). Overall, we tried the following values $\{20, 30, 40, 50, 0.05n, 0.1n, 0.2n, \log n\}$.

Results, depicted in Table 3 an 4, show that the branching strategy performs best with $\log n - 5$ levels of recursion. On our tuning instances this corresponds to a range of two to four levels of recursion. However, choosing a constant number from this range performs definitely worse. A possible explanation for this is that initial kernelization often produces few relatively small connected components which are solved separately. Consequently, a branching order is calculated for each of them. Depending on the size of the components, different numbers of recursion. Tuning the maximum separator size yields best results when choosing a non constant threshold of ten percent of the remaining vertices.

	number of levels of recursion							
Instances	2	3	4	5	$\log n - 6$	$2\log n - 5$	$\log n - 4$	$\log n - 3$
PACE	16.1889	16.4489	12.6748	21.0732	12.5689	**10.4004**	15.6870	13.0480
DIMACS	1.2103	1.2418	1.1844	1.1287	**1.0235**	1.2107	1.1968	1.1780
sparse networks	1	1	1	1	1	1	1	1
All	1	1	1	1	1	1	1	1

Table 3: Runtimes tuning the number of levels of recursion in the nested dissection branching strategy

	maximum separator size at each recursion							
Instances	20	30	40	50	$\log n$	$0.005n$	$0.1n$	$0.2n$
PACE	16.0360	11.3461	10.6191	13.3114	13.9448	23.3915	**10.4004**	15.7568
DIMACS	**1.0136**	1.2107	1.3165	1.3230	1.1175	1.1507	1.0507	1.1506
sparse networks	1	1	1	1	1	1	1	1
All	1	1	1	1	1	1	1	1

Table 4: Number of branches tuning the number of levels of recursion in the nested dissection branching strategy

Finally, we optimize the branching strategies utilizing reduction rules (i.e., first four branching strategies in Section 4.2). Those strategies only have one tuning parameter, which is the minimum degree required in order for a vertex to be considered for branching by the respective branching strategy. In general, the degree distribution in a graph depends on the graph class and the size of the graph. Furthermore, applying dominance, unconfined or twin reduction, respectively, only reduces a constant number of vertices. Thus, we do not use a constant threshold. Instead, our strategies consider vertices whose degrees differ from the maximum degree of the current graph by a certain constant values at the time of branching. We choose the concrete values from the set $\{1, 2, 3, 4\}$.

The results of our tuning experiments are shown in Tables 5 to 8. They suggest that the branching strategies based on dominance and unconfined reduction perform best when only vertices with degree greater or equal one less than the current maximum degree are considered. This was expected, as both unconfined and dominance reduction rule each eliminate one vertex of the graph.

The branching strategy based on the twin reduction achieves the best results when choosing 2 for the value of the tuning parameter. Applying twin reduction rule reduces either four or five vertices from the graph. Using the branching strategy based on funnel reduction, best results are obtained with 1 as value for the tuning parameter.

5.3 Evaluation

Having tuned the branching strategies, we can now evaluate their effectiveness. We start by comparing the strategies designed by following our first approach which is to decompose the graph by branching.

	almost dominance				almost unconfined			
Instances	0	1	2	3	0	1	2	3
PACE	1.0518	**1.0059**	1.0470	1.0283	1.0358	**1.0216**	1.0306	1.0449
DIMACS	1.3339	1.3386	1.3374	**1.3218**	1.2784	1.2786	1.2757	**1.2313**
sparse Networks	0.9560	**0.9399**	0.9590	1.0667	0.9739	**0.9589**	0.9754	1.1130
All	0.9981	**0.9942**	0.9944	0.9975	0.9957	**0.9870**	0.9925	1.0025

Table 5: Runtimes tuning the minimum degree required in order for a vertex to be considered for branching by the respective branching strategy

	almost twins				almost funnels			
Instances	0	1	2	3	0	1	2	3
PACE	0.9767	0.9841	**0.9617**	0.9856	1.0152	0.9532	**0.9148**	0.9634
DIMACS	1.2338	1.2713	**1.2235**	1.2745	1.2921	1.3400	**1.2877**	1.3250
sparse Networks	0.9394	**0.9382**	0.9387	0.9429	0.9986	**0.9724**	0.9810	1.1417
All	0.9499	0.9643	**0.9397**	0.9666	0.9931	0.9733	**0.9406**	0.9944

Table 6: Number of branches tuning the minimum degree required in order for a vertex to be considered for branching by the respective branching strategy

	almost dominance				almost unconfined			
Instances	0	1	2	3	0	1	2	3
PACE	1.0003	0.9931	**0.9875**	0.9897	1.0016	**0.9962**	1.0027	1.0225
DIMACS	1.3093	1.3172	1.3173	**1.3040**	1.3049	1.2926	1.2885	**1.2711**
sparse Networks	1.0682	**1.0494**	1.1361	1.3363	1.0686	**1.0472**	1.1518	1.4241
All	0.9996	**0.9961**	1.0036	1.0217	0.9990	**0.9901**	1.0043	1.0364

Table 7: Runtimes tuning the minimum degree required in order for a vertex to be considered for branching by the respective branching strategy

	almost twins				almost funnels			
Instances	0	1	2	3	0	1	2	3
PACE	1.0025	1.0025	1.0023	**1.0022**	0.9649	0.8973	**0.8924**	0.92467
DIMACS	1.3103	1.3103	**1.3103**	**1.3103**	1.3139	1.311	1.3054	**1.2917**
sparse Networks	1.0791	1.0745	**1.0659**	**1.0659**	1.0697	**1.0143**	1.0468	1.3213
All	1.0023	1.0017	**0.9985**	1.0002	0.9837	0.9423	**0.9420**	0.9835

Table 8: Number of branches tuning the minimum degree required in order for a vertex to be considered for branching by the respective branching strategy

Results of testing these strategies are depicted in Figures 3 to 7. To compare different strategies we utilize performance plots introduced in [**?**]. These plots depict the performance of a strategy in relation to the best strategy on a per instance basis. The y-axis shows $1 -$ (best strategy/strategy). For each branching strategy, these values are sorted in decreasing order. Smaller values indicate a performance closer to the best strategy, whereas values close to one show that the strategy performed considerably worse. A value of zero signifies the strategy performed best on an instance. Timeouts are placed above one. We use a square root scale in order to increase the visible details in the area close to zero.

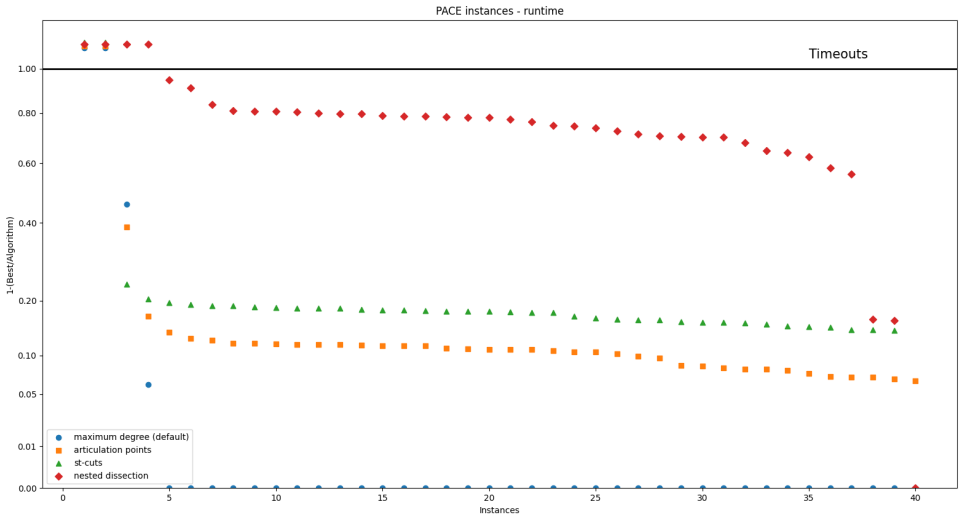
Results show that for most of the PACE and DIMACS instances the branching strategy utilizing articulation points turns out to be slower than the default branching strategy. The same is true for the graphs obtained from the OCT instances. The reason for this is that almost no articulation

points are found on these graphs. Consequently, default branching is applied anyway, resulting in unnecessary overhead by searching for articulation points in the first place. Nevertheless, there are a few outliers among the sparse networks, where branching on articulation points significantly out performs the default branching strategy. The most notable of these exceptions is the `web-Stanford` instance (sparse network), which is intractable on our machine within 24 hours using the default branching strategy. The same instance, however, is solved in less than 20 seconds utilizing branching on articulation points, although only 35 articulation points are discovered throughout the whole run. In fact, 34 of the 35 articulation points are found at recursion depths lower than 12. This supports the hypothesis that decompositions performed at early stages of the algorithm are highly effective. Furthermore, articulation points can be found very efficiently in linear time using a DFS scheme. Thus, the additional overhead needed for searching articulation points, even if none exists, is very small. In our experiments, this branching strategy is about five to ten percent slower on average than the default branching strategy, if almost no articulation points are found. Obviously, the total number of branching steps needed to solve the respective instances is almost equal to baseline.

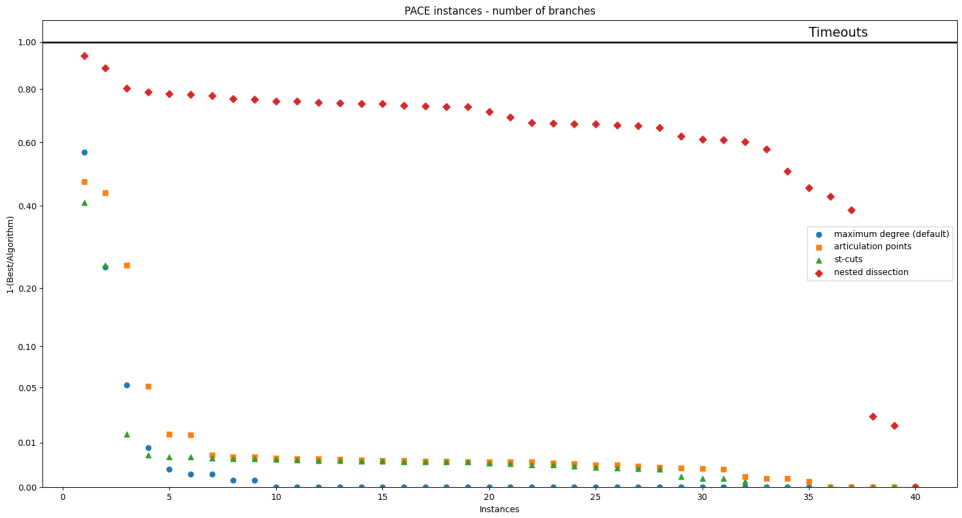
Similar to branching on articulation points, the branching strategy using edge cuts performs worse than the default branching strategy on PACE and DIMACS instances as well as on the graphs obtained from OCT instances. Our experiments show that cuts suitable for branching are rarely found. Thus, the fallback strategy is getting applied almost always, which results in a nearly equal number of branches as with the default strategy. However, the overhead of calculating the edge cut is considerably higher than detecting articulation points. Using the configuration obtained by parameter tuning, solving PACE and DIMACS instances takes about 20 percent longer on average than with the default branching strategy. Nevertheless, just as branching on articulation points, this strategy performs considerably well on some of the sparse networks. That is because a lot of small and reasonable well balanced cuts are found on those instances which are therefore easily decomposable. However, branching on articulation points overall performs better, since less overhead is produced.

Using the branching strategy based on nested dissection yields mixed results in our experiments. On the PACE instances, this strategy performs a lot worse than default branching as the other two strategies examined so far had done. However, there is one outlier, for which this branching strategy gives the best result compared to all branching strategies. Furthermore, nested dissection branching performs slightly better than default on DIMACS instances. The latter is surprising as both, branching on articulation points and branching on edge cuts, each perform more or less equal on PACE and DIMACS instances. Unfortunately, we do not have an explanation for this. On sparse networks, the results are mixed. For two of the instances nested dissection branching yields the best runtimes of all branching strategies. However, on the remaining three instances this branching strategy performs very badly.

Summarizing, the results so far show that our approach to decompose instances by branching is highly effective on sparse networks. The reason for this is that those instances are easily decomposable. Also, sparse networks have a low average degree. Thus, branching on vertices of maximum degree becomes ineffective as soon as all vertices of high degree have been removed. Unfortunately, this approach does not work on denser graphs. The experiments also indicate that decomposing the graph at an early stage of the algorithm is much more powerful than in later stages.

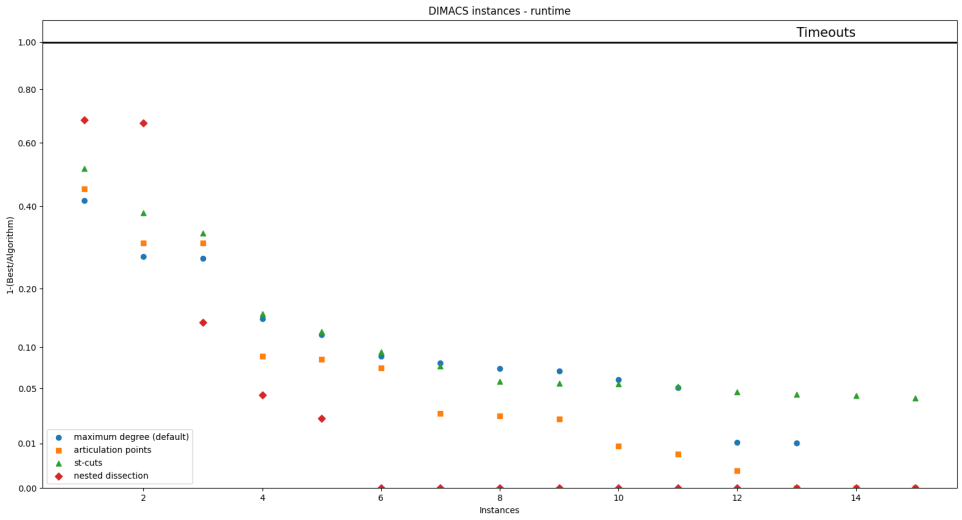


(a) runtime

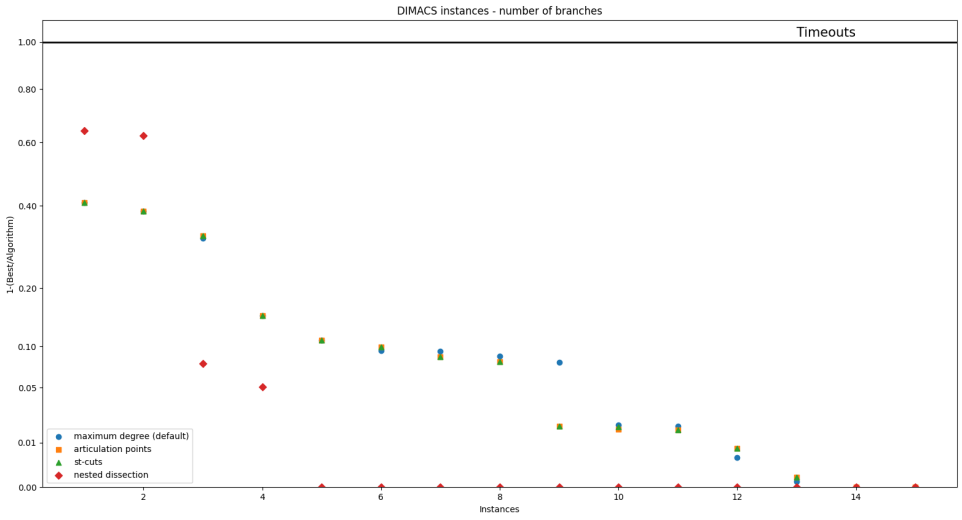


(b) number of branches

Figure 3: Performance plots of decomposing branching strategies on PACE instances

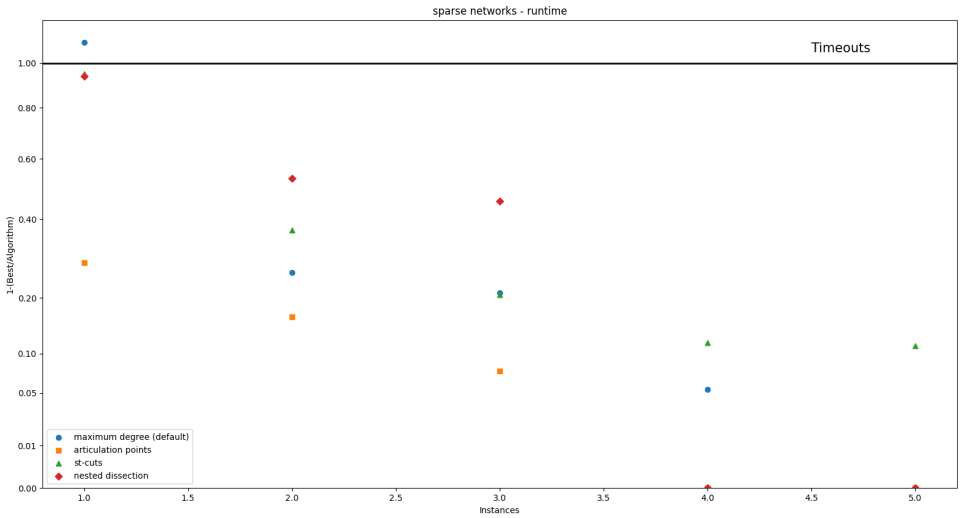


(a) runtime

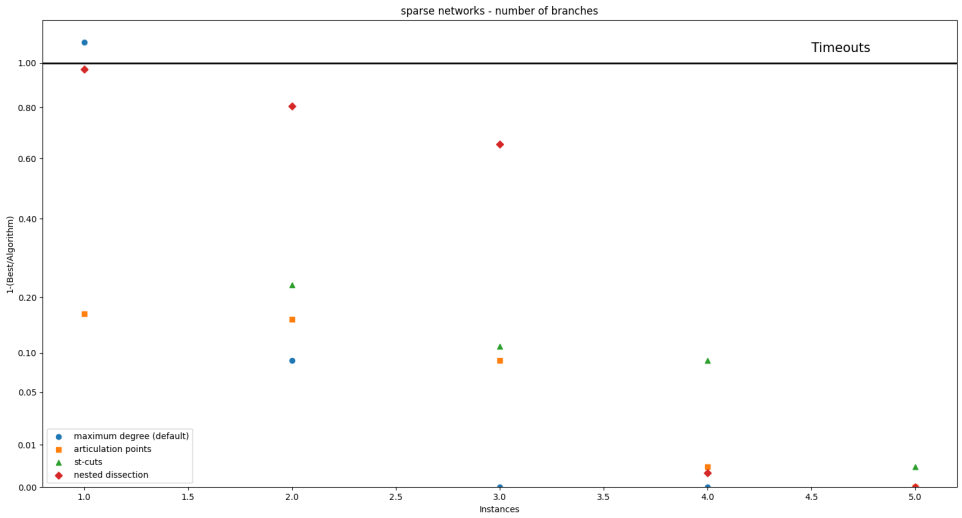


(b) number of branches

Figure 4: Performance plots of decomposing branching strategies on DIMACS instances

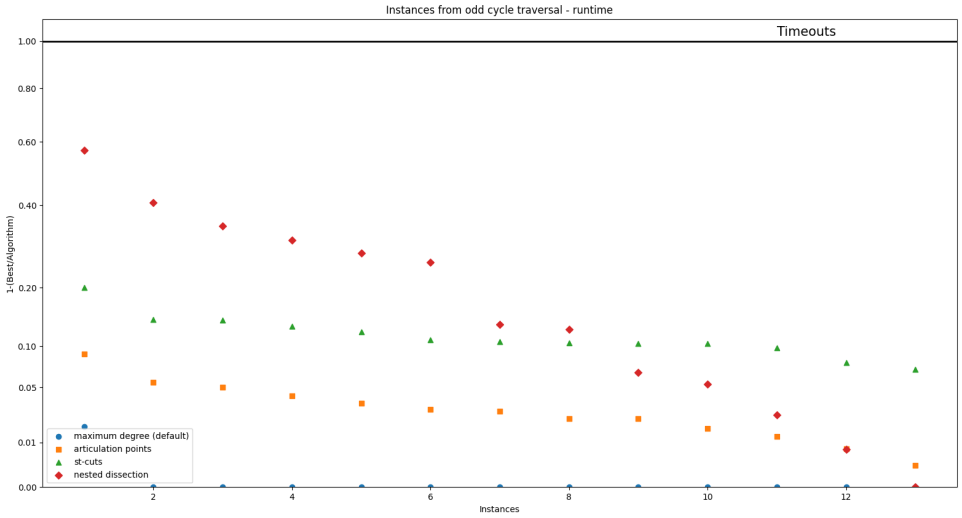


(a) runtime

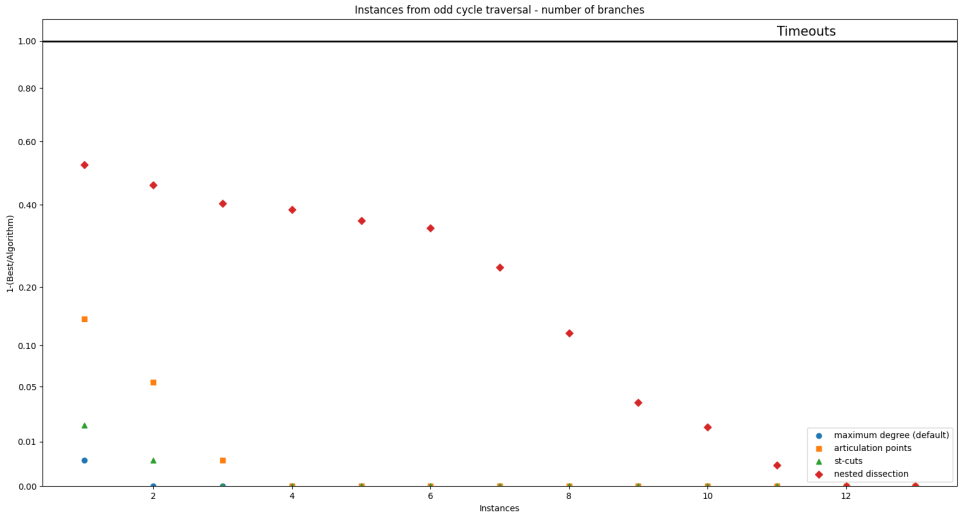


(b) number of branches

Figure 5: Performance plots of decomposing branching strategies on sparse networks

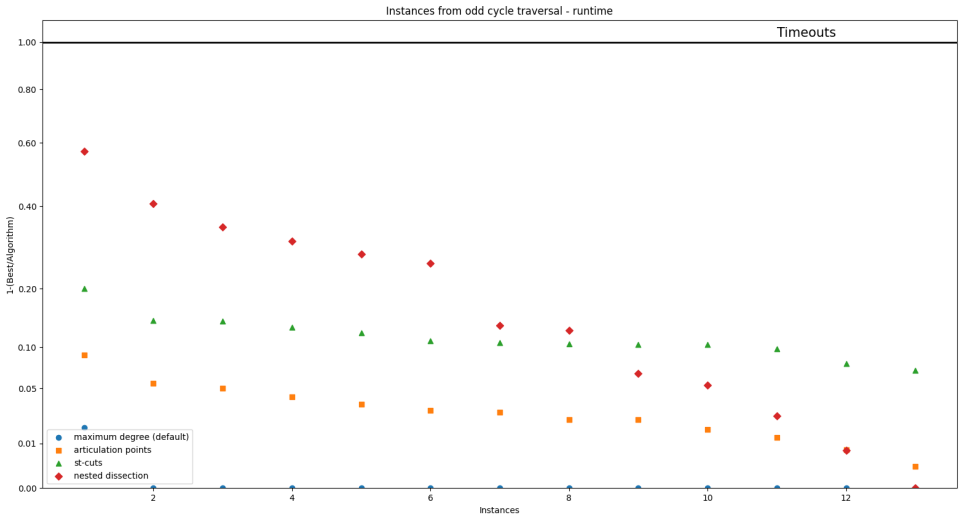


(a) runtime

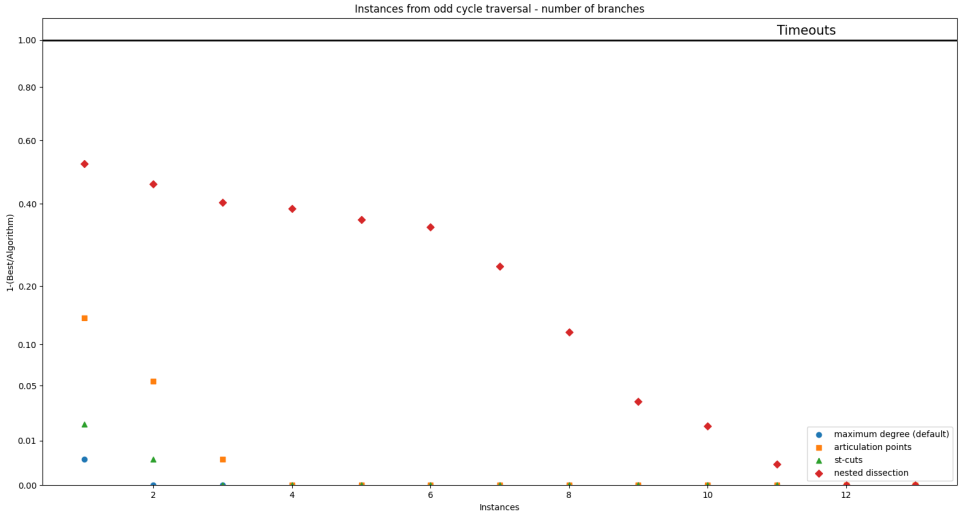


(b) number of branches

Figure 6: Performance plots of decomposing branching strategies on graphs obtained from OCT instances



(a) runtime



(b) number of branches

Figure 7: Performance plots of decomposing branching strategies on all instances

Next, we compare the branching strategies following our second approach which is to branch on vertices so that reduction rules become applicable afterwards. The experimental results are depicted in Figures 8 to 12.

Using the branching strategy based on dominance reduction almost always reduces the total number of branching steps needed for solving PACE and DIMACS instances to a minor extend. Unfortunately, the dominance reduction is not applied in the branch and reduce algorithm since it is fully contained in the unconfined reduction. Therefore, finding suitable vertices that enable the application of dominance reduction upon removal comes with additional time overhead.

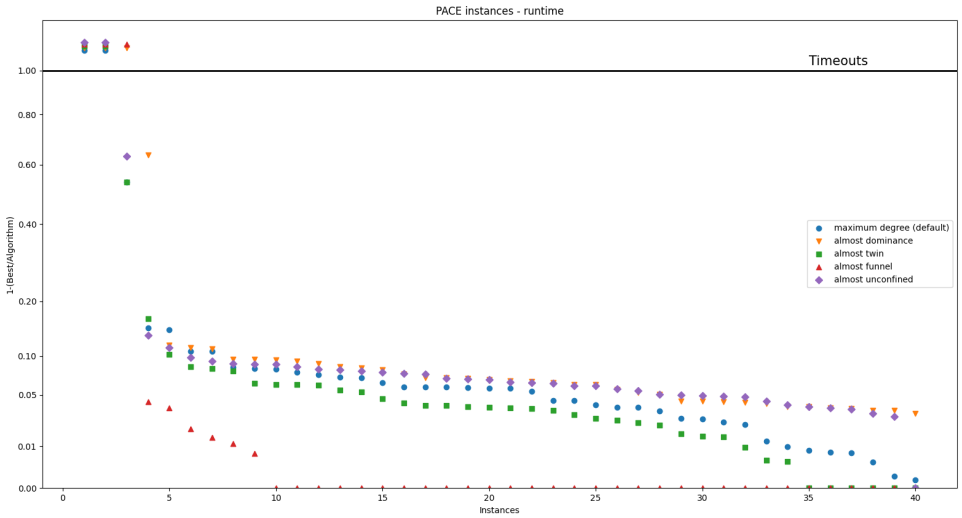
The branching strategy based on unconfined reduction reduces the total number of branching even by a larger amount than the strategy based on dominance reduction. Also, vertices considered for branching can be found during kernelization while searching unconfined vertices. However, this still produces a non negligible time overhead. In total, the branching strategies utilizing dominance and unconfined reduction perform worse than the default branching strategy on PACE instances. On DIMACS instances the latter strategy has a small advantage over default branching.

On sparse networks both strategies perform better than default branching on some of the instances. A reason for this may be that the average vertex degree on those graphs is much lower than on denser graphs. Moreover, vertices with high degree are quickly removed by branching at an early stage of the algorithm. Thus, verifying whether a vertex is almost dominated or almost unconfined is much faster in relation to overall kernelization time since fewer adjacencies have to be checked.

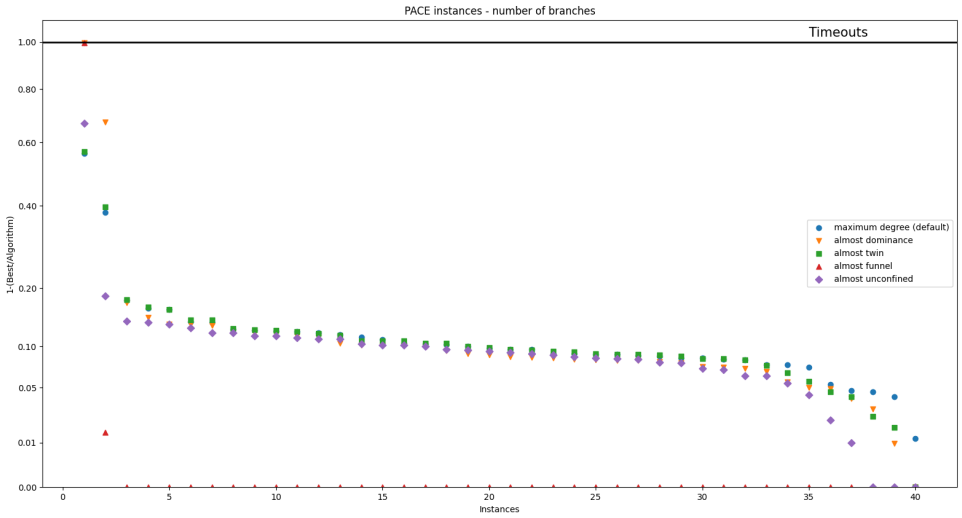
The branching strategy based on the twin reduction rule overall performs slightly better than the default strategy on some of the PACE and DIMACS instances and slightly worse on others. This is due to the fact that on those graphs almost twins are rarely found. However, finding vertices that enable twin reduction produces very little overhead. Also, applying twin reduction is very efficient, since the number of vertices in the graph is reduced by four to five. For those reasons, this branching strategy has an advantage over default branching on sparse networks. Since the average degree on those graphs is very low, vertices considered for branching are found more often.

Our last branching strategy targeting the funnel reduction overall requires the least number of branching steps to solve PACE instances. On most PACE instances this strategy also reduces the required runtime. However, the reduction in branching steps does not compensate for the time overhead caused by finding almost funnels on all of the instances. On most of the DIMACS instances, this branching strategy performs roughly equal to the default branching strategy. On sparse networks this branching strategy also seems to be competitive to default branching; it is, however, outperformed by the twin reduction on every instance. A possible explanation for this is that funnel reduction removes the common neighbors of both parts of the funnel. On denser graphs the common neighborhood of a funnel is potentially larger. Thus, funnel reduction is more effective on those graphs than on sparse networks.

The experiments using branching strategies following our second approach demonstrate that branching in order to enable reduction rules can effectively reduce the total number of branches needed to solve an instance. Contrary to our first approach, this also works on denser graphs. These branching strategies also yield more constant results with fewer outliers. Unfortunately, in many cases the reduction in branching steps does not compensate for the additional overhead caused by finding suitable branching vertices. Nevertheless, we believe that runtime can be further tuned.

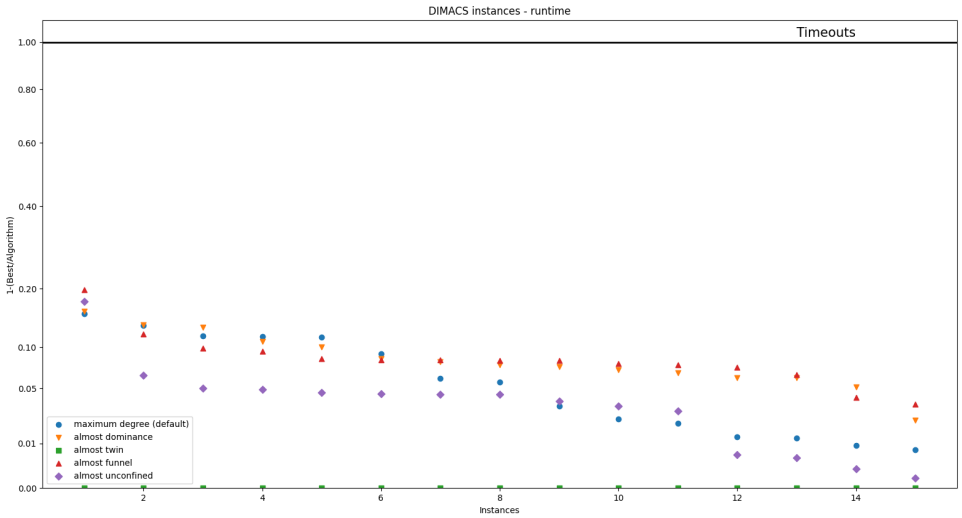


(a) runtime

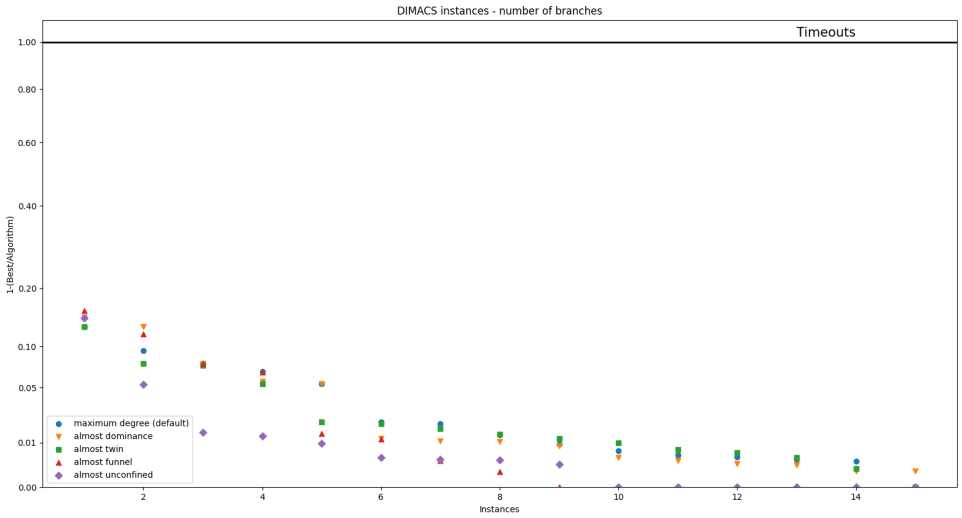


(b) number of branches

Figure 8: Performance plots of branching strategies targeting reductions on PACE instances

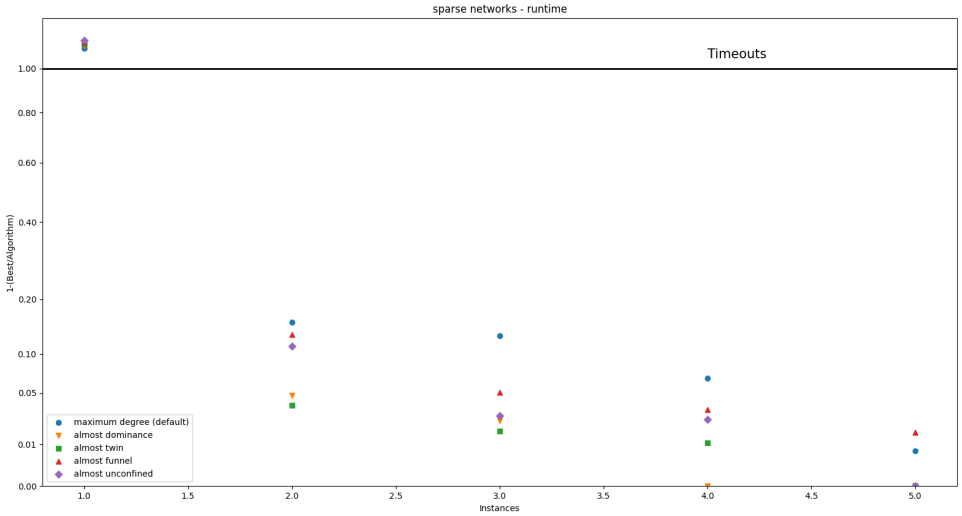


(a) runtime

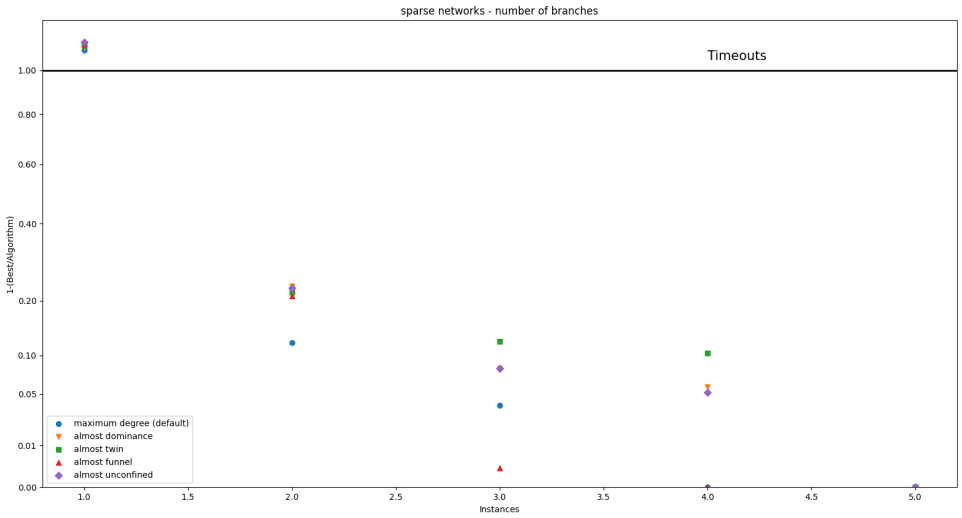


(b) number of branches

Figure 9: Performance plots of branching strategies targeting reductions on DIMACS instances

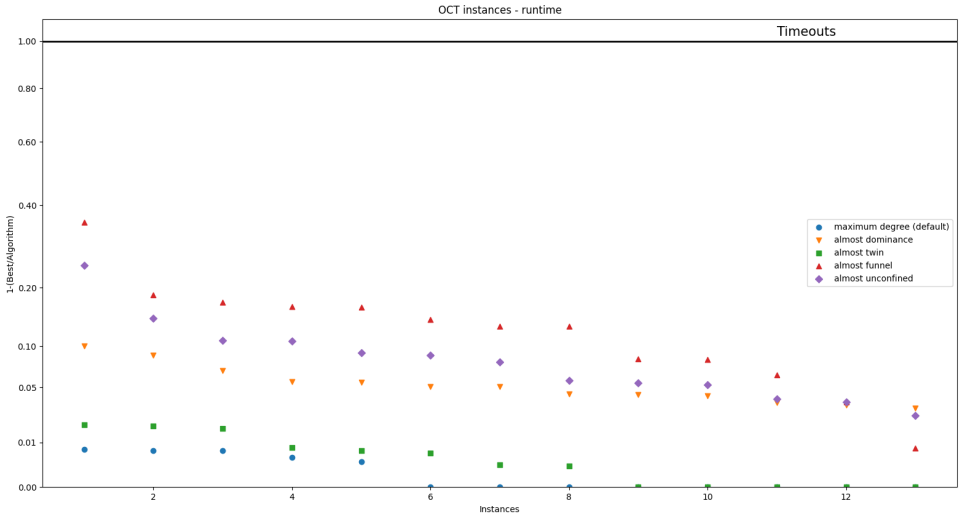


(a) runtime

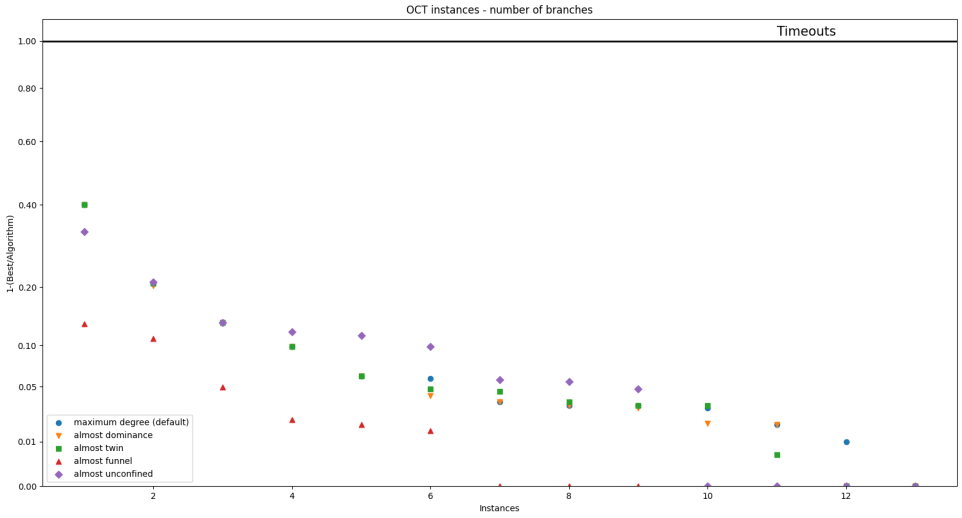


(b) number of branches

Figure 10: Performance plots of branching strategies targeting reductions on sparse networks

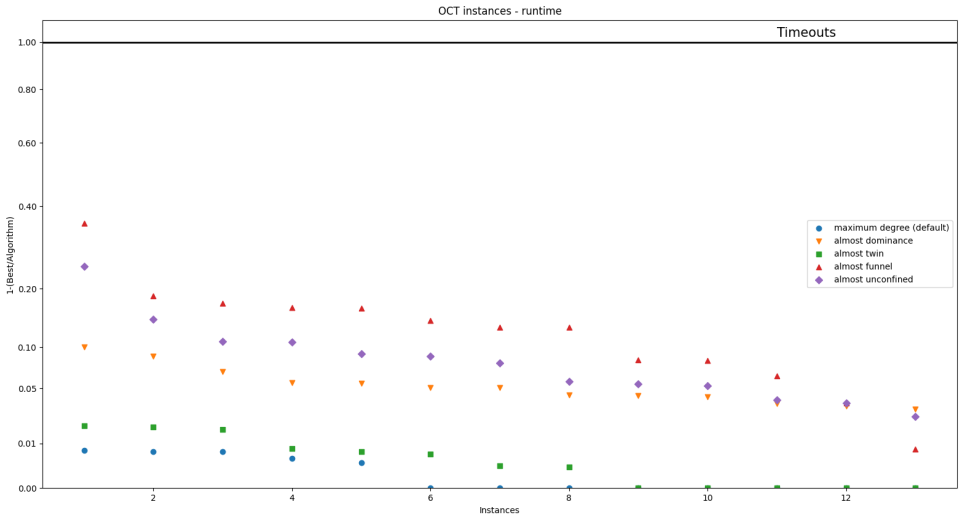


(a) runtime

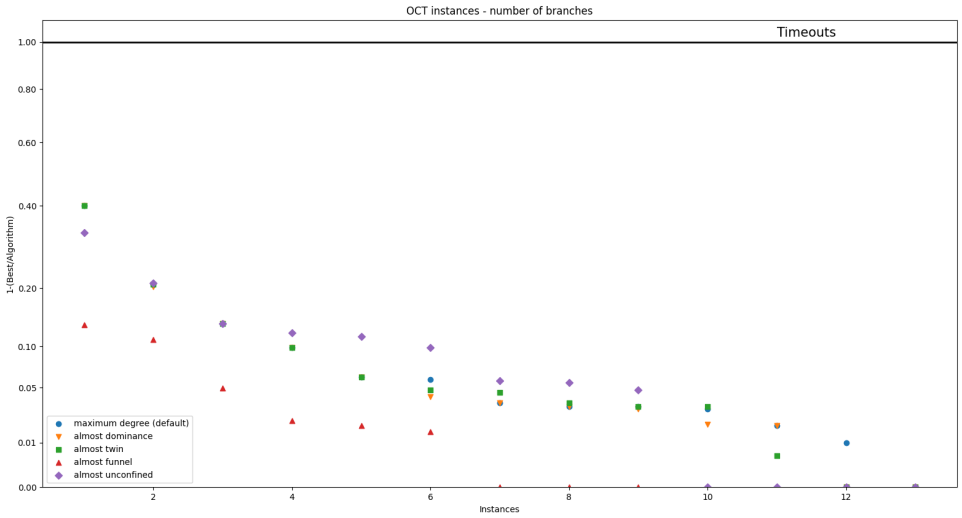


(b) number of branches

Figure 11: Performance plots of branching strategies targeting reductions on graphs obtained from OCT instances



(a) runtime



(b) number of branches

Figure 12: Performance plots of branching strategies targeting reductions on all instances

5.4 Conclusion

In this Thesis, we examined multiple branching strategies for a branch and reduce maximum independent set algorithm. Essentially, we evaluated two different approaches. The first approach is to branch on vertices that decompose the graph. Following this approach we presented three different branching strategies. All of them proved to be very effective on sparse networks, since those instances can easily be decomposed by branching. However, this approach does not work very well on denser graphs with low skewness (i.e. graphs with a more symmetric degree distribution). The reason for this is that on those instances small and at the same time reasonably balanced separators are rarely found. Furthermore, those graphs contain many vertices of high degree. Hence, the greedy strategy of branching on vertices of maximum degree (i.e. the default strategy) is overall more competitive than on sparse graph. In summary, we ascertained that decomposing a graph by branching can drastically reduce the total number of branching steps needed to solve an instance, and therefore boost the runtime of the algorithm greatly.

Our second approach concerns branching on vertices so that reduction rules become applicable afterwards. For sparse networks, branching on vertices that prevent twin reductions performs better than the default strategy in almost all cases. The reason for this is that the average degree in those graphs is low, and therefore almost twins are found quite often. Stallmann, Ho and Goodrich showed that the dominance and unconfined reduction rules are highly effective on dense graphs with a high degree variation [**?**]. Therefore, the reduction step prior to branching is key to minimizing the complexity of those instances. In fact, our experiments show that using branching strategies which enable the application of the dominance and unconfined reduction rules in many cases decreases the total number of branching steps on graphs displaying those properties (e.g. DIMACS an PACE instances). Unfortunately, in our implementations, the reduction in branching steps does not compensate for the additional time overhead of finding suitable branching vertices. Here, the situation is different with the branching strategy utilizing funnel reduction, which has proven to be the best strategy for some of the PACE and DIMACS instances. In conclusion, we believe that targeting reduction rule is very promising for reducing the number of branching steps on highly connected dense graphs.

6 Future Work

In this section we discuss possible options to further develop and optimize our branching strategies described in this thesis. Moreover, we outline new ideas for alternate approaches for branching strategies.

As already mentioned in Section 5, the aim of this thesis was not to optimize certain strategies down to the last detail. Instead, we examined a range of different strategies following multiple approaches in an attempt to comparatively evaluate the potential of those solutions. Thus, for most of our implementations, there is a lot of leeway to optimize runtime. For instance, we did not implement our own variant of a preflow push algorithm, but instead used an implementation from the KaHIP library [**?**]. Consequently, the current graph has to be converted to the format used by the KaHIP library at each branching step, which creates additional overhead.

Our test results show that branching on articulation points is ineffective most of the time. However, there are two instances for which this strategy produces significantly better results than default branching. Unfortunately, we did not find a satisfying explanation for these outliers. Analyzing different graph parameters on the initial kernel of those instances showed that both outliers seem to have a higher average betweenness centrality than other instances of their respective graph class. However, when looking at all instances together, we could not find any correlation between the effectiveness of this branching strategy and betweenness centrality. Nevertheless, we believe that exploring relations between the effectiveness of branching strategies and graph characteristics can help optimizing existing strategies or even finding new ones. Stallmann, Ho and Goodrich previously proposed a similar approach involving reduction rules [**?**]. They showed that there is a correlation between certain graph parameters and the effectiveness of the reduction rules. Furthermore, they presented an improved variant of the branch and reduce algorithm by Akiba and Iwata [1] where they only apply a subset of reduction rules based on simple graph parameters of the instance. This way, they reduced kernelization time and significantly speeded up the algorithm.

Nested dissection has proven to work very well, at least on DIMACS instances. Therefore, we see a lot of potential for further optimization involving this strategy. The METIS library provides many settings that can be used to tweak the nested dissection algorithm, and thus the branching strategy. Further, it might be interesting to examine whether there is a tradeoff between the balance of the parts and the size of the separators. Also, a partitioning algorithm that produces higher quality bipartitions at the cost of computation time could improve the strategy.

Our experiments show that the branching strategy, which targets the dominance reduction, significantly reduces the total number of branches needed on PACE and DIMACS instances. Unfortunately, the overhead in computation time it takes for finding almost dominated vertices is excessively large. Moreover, it is not clear whether those vertices can be found with our proposed modification of the unconfined reduction. Potentially, this problem could be fixed by considering extending children in the order of their appearance.

We believe that the concept of branching aiming at the application of reduction rules can also be adopted for the packing branching rule. During kernelization one can find vertices whose removal forces the reduction of other vertices in order to satisfy the constraints resulting from the packing rule. This could also be utilized in the branching strategy that targets chain reductions, since packing reduction does not introduce new vertices or edges.

During the initial testing of our branching strategy using nested dissection ordering, we noticed that computing the nested dissection ordering at later stages is not advantageous. A possible explanation

is that branches at lower depth of recursion have a greater influence on the runtime of the algorithm than such at a higher depth. For this reason it might be worth investing a lot of execution time into finding branching vertices at the start of recursion compensating for the time spent by using a simple branching strategy at higher depth of recursion.

References

[1] Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theor. Comput. Sci.*, 609:211–225, 2016.

[2] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Fast algorithms for max independent set. *Algorithmica*, 62(1-2):382–415, 2012.

[3] Sergiy Butenko and Wilbert E. Wilhelm. Clique-detection models in computational biochemistry and genomics. *Eur. J. Oper. Res.*, 173(1):1–17, 2006.

[4] Randy Carraghan and Panos M. Pardalos. An exact algorithm for the maximum clique problem. 1990.

[5] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: Further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.

[6] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.

[7] Tammy M. K. Cheng, Yu-En Lu, Michele Vendruscolo, Pietro Liò, and Tom L. Blundell. Prediction by graph theoretic measures of structural effects in proteins arising from non-synonymous single nucleotide polymorphisms. *PLoS Computational Biology*, 4(7), 2008.

[8] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *J. ACM*, 56(5):25:1–25:32, 2009.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. Linear-time FPT algorithms via network flow. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1749–1761. SIAM, 2014.

[11] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 83–93. Springer, 2010.

[12] Joachim Kneis, Alexander Langer, and Peter Rossmanith. A fine-grained analysis of a simple independent set algorithm. In Ravi Kannan and K. Narayan Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, December 15-17, 2009, IIT Kanpur, India*, volume 4 of *LIPIcs*, pages 287–298. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2009.

[13] Chu-Min Li, Zhiwen Fang, and Ke Xu. Combining maxsat reasoning and incremental upper bound for the maximum clique problem. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pages 939–946. IEEE Computer Society, 2013.

[14] Chu-Min Li, Hua Jiang, and Felip Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Comput. Oper. Res.*, 84:1–15, 2017.

[15] Chu-Min Li, Hua Jiang, and Ruchu Xu. Incremental maxsat reasoning to reduce branches in a branch-and-bound algorithm for maxclique. In Clarisse Dhaenens, Laetitia Jourdan, and Marie-Eléonore Marmion, editors, *Learning and Intelligent Optimization - 9th International Conference,*

LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers, volume 8994 of *Lecture Notes in Computer Science*, pages 268–274. Springer, 2015.

[16] Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[17] George L. Nemhauser and Leslie E. Trotter Jr. Vertex packings: Structural properties and algorithms. *Math. Program.*, 8(1):232–248, 1975.

[18] Deepak Puthal, Surya Nepal, Cécile Paris, Rajiv Ranjan, and Jinjun Chen. Efficient algorithms for social network coverage and reach. In Barbara Carminati and Latifur Khan, editors, *2015 IEEE International Congress on Big Data, New York City, NY, USA, June 27 - July 2, 2015*, pages 467–474. IEEE Computer Society, 2015.

[19] Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):144, 2008.

[20] Mingyu Xiao and Hiroshi Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theor. Comput. Sci.*, 469:92–104, 2013.

[21] Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Inf. Comput.*, 255:126–146, 2017.