**Jagiellonian University**
Department of Theoretical Computer Science

**Artur Salawa**

# Implementation of exponential algorithms for the independent set problem

Bachelor's Thesis

Supervisor: dr hab. inż. Krzysztof Turowski

August 2023

# Contents


**1 Introduction** **3**
- 1.1 Important notions . . . 3
- 1.2 Background . . . 4
- 1.3 General method for estimating an upper-bound . . . 6

**2 Simple algorithms** **8**
- 2.1 Naive Algorithm . . . 8
  - 2.1.1 Correctness . . . 8
  - 2.1.2 Computational complexity . . . 9
- 2.2 Mis1 . . . 9
  - 2.2.1 Correctness . . . 9
  - 2.2.2 Computational complexity . . . 10
- 2.3 Mis3, Mis4 and Mis5 . . . 11
  - 2.3.1 Special case: graphs with $\Delta(G) \leq 2$ . . . 11
  - 2.3.2 General case algorithms . . . 13
  - 2.3.3 Correctness . . . 14
  - 2.3.4 Computational complexity in Mis3, Mis4 and Mis5 . . . 15

**3 Advanced algorithms** **16**
- 3.1 Mis2 . . . 16
  - 3.1.1 New definitions and lemmas . . . 16
  - 3.1.2 Algorithm . . . 17
  - 3.1.3 Case A and Case B . . . 18
  - 3.1.4 Case C . . . 18
  - 3.1.5 Case D . . . 19
  - 3.1.6 Case E . . . 20
  - 3.1.7 Case F . . . 20
  - 3.1.8 Case G . . . 21
  - 3.1.9 Case H . . . 21
  - 3.1.10 Case I . . . 21
  - 3.1.11 Case J . . . 22
  - 3.1.12 Case K . . . 22
  - 3.1.13 Case L . . . 23
  - 3.1.14 Case M . . . 23

3.1.15 Overview . . . . . 24
3.2 MisFolding . . . . . 24

**4 Implementation and benchmark** **28**
4.1 Implementation . . . . . 28
4.1.1 KOALA NetworKit . . . . . 28
4.1.2 Implementation details and expectations . . . . . 28
4.1.3 Correctness . . . . . 29
4.2 Benchmark results . . . . . 30

**Appendices** **36**

# Chapter 1

# Introduction

## 1.1 Important notions

Before we begin we should establish common set of rules and definitions that we are going to use throughout the article. We only consider undirected graphs without loops or multiple edges. Most of the definitions are taken from [33] but also from [3], [6], [9] and [37]. Some definitions are refined to fit to our specific needs. For now, we only introduce the most important concepts. The remaining ones will be added later in respective places.

**Definition 1.1** (graph)**.** *A* ***graph*** $G = (V, E)$ *consists of* $V$*, a nonempty set of* ***vertices*** *(or* ***nodes****) and* $E$*, a set of* ***edges****. Each edge has two vertices associated with it, called its* ***endpoints****. An edge is said to* ***connect*** *its endpoints.*

This type of graph is also called *undirected graph*.

**Definition 1.2** (adjacent nodes)**.** *Two vertices* $u, v \in V$ *in an undirected graph* $G = (V, E)$ *are called* ***adjacent*** *(or* ***neighbors****) in* $G$ *if* $u$ *and* $v$ *are endpoints of an edge* $e \in E$ *of* $G$*.*

*Such an edge* $e$ *is called* ***incident*** *with the vertices* $u$ *and* $v$*.*

**Definition 1.3** (degree of a vertex)**.** *The* ***degree of a vertex*** *in an undirected graph is the number of edges incident with it. The degree of the vertex* $v$ *is denoted by* $\deg(v)$*.*

Frequently, we need to take a closer look on what is exactly happening in a small part of a graph. We also have definitions for that:

**Definition 1.4** (subgraph)**.** *A subgraph of a graph* $G$ *is another graph formed from a subset of the vertices and edges of* $G$*.*

**Definition 1.5** (induced subgraph)**.** *An induced subgraph of a graph is a subgraph formed from a subset of vertices and from all of the edges that have both endpoints in the subset. Induced subgraph formed from vertices* $V'$ *of graph* $G(V, E)$ *we denote by* $G[V']$*.*

We are finally ready to define the main object of this study, the independent set.

**Definition 1.6** (independent set, stable set)**.** *A subset $I \subset V$ is called an **independent set** (in some literature also called a **stable set**) if there is no edge in the subgraph induced by $I$.*

**Definition 1.7** (maximum independent set)**.** *An independent set is **maximum** if there are no larger independent sets in a graph in terms of cardinality.*

**Definition 1.8** (Maximum Independent Set (MIS) problem)**.** *In the **Maximum Independent Set (MIS) problem**, we are given an undirected graph $G = (V, E)$. The task is to find an independent set $I$ such that $G[I] = (V', E')$ and $E = \emptyset$ and $V'$ have maximum cardinality.*

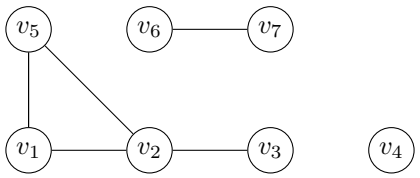



Figure 1.1: An example graph $G_0$

For example, the maximum independent set presented in Figure 1.1, is equal to 4. Independent sets of that cardinality are: $\{v_4, v_6, v_3, v_1\}$, $\{v_4, v_6, v_3, v_5\}$, $\{v_4, v_7, v_3, v_1\}$, $\{v_4, v_7, v_3, v_5\}$.

## 1.2 Background

The maximum independent set was first solved only for special cases. For the general problem, for some time only the obvious naive solution $O^*(2^n)$ was known. More about special cases later.

The maximum independent set is closely related to the minimum vertex cover problem. If $I$ is a maximum independent set of $G$ then $G \setminus I$ is a Minimum Vertex Cover. The maximum independent set problem is also related to the clique problem. The maximum independent set of $G$ in a complement graph of $G$ will be a maximum clique. These problems and their solutions can be easily reduced to each other. In 1972 Tarjan the first to find a solution $O^*(1.286^n)$ for the maximum clique problem.

Few years later, the same author was the first one to tackle the general maximum independent set problem and published his work in 1977 [35]. The computational complexity of this algorithm is $O^*(3^{n/3}) = O^*(1.2599^n)$. Unfortunately, the algorithm in pseudocode form takes 5 pages and has 35 cases. This would make an implementation challenging, and we decided against it. As a side note in 1965 it was proved that every graph contains at most $3^{n/3}$ maximal independent sets [28].

Later in 1986 Jian [21] improved the complexity to $O^*(1.2346^n)$ and in the same year Robson [32] obtained an even better result with $1.2278^n$. We didn't mention it, but so far all algorithms use only polynomial memory. Robson also introduces a second algorithm that uses exponential memory and achieves a time of $O^*(1.2109^n)$. Algorithms of this kind are not very popular because in real life memory is quite an expensive resource.

Another breakthrough happened in 2006 [13]. The authors used a new technique called folding and even with a simple algorithm they reached the result of $O^*(1.2202^n)$. This was a considerable shift towards simplicity compared to a series of previous publications that focused on several branching cases and still performed worse.

There were a few more recent publications. In 2009 [22] a new concept of satellites was introduced to reach a time of $1.2132^n$. Article [5] from 2010 further proposes another solution that improves to bound to $O^*(1.2114^n)$.

Finally, the latest result [39] from 2017 breaks the barrier of $1.2^n$ and reaches the time $O^*(1.1996^n)$.

The maximum independent set problem is a special case of the maximum weight independent set problem where each weight is equal to 1. Some graph classes have solutions for that generalized version (and, by implication, also for the original, unweighted problem). That is: chordal graphs [18], [14]; perfect graphs [19]; claw-free graphs [26], [30]; graphs of bounded clique-width [10] and $P_5$-free graphs [24].

Maximum independent set problem is NP-hard [15] in general case but also with strong restrictions. For example: planar graphs with a maximum vertex degree at most 3 [16], triangle-free graphs [29] are still NP-hard. The maximum independent set problem is NP-complete for: planar graphs of maximum degree 6 [17], 3-regular planar graphs [27], unit disk graph [8], [11]. Maximum independent set problem is W[1]-hard in the class of graphs without induced 4-cycle [20].

Polynomial time approximation schemes (PTAS) algorithms for maximum independent set problem exist for: planar graphs [1], [38]; unit disk graphs [31, 11], disk graphs with disk representation[12]; pseudo-disks [7].

Fixed parameter tractable (FPT) algorithms parameterized by the solution size exist for even-hole-free graphs [20], dart-free graphs, cricket-free graphs and some others [4]. The maximum independent set problem also has a 2-approximation algorithm for the unit disk graph.

The maximum independent set is $O(n^c)$-inapproximable unless P = NP for some positive constant $c$ [2]. It was improved later in [40] which stated that: there exist some $\epsilon > 0$ such that if NP $\neq$ ZPP, then there does not exist Las Vegas $O(n^\epsilon)$ algorithm that works in expected polynomial time.

## 1.3 General method for estimating an upper-bound

To determine complexities of algorithms we will be using these two asymptotic notions, deriving from the Landau big-O notation:

**Definition 1.9** (Big $O$). $\boldsymbol{O(g(n))}$ *is defined as:*
$O(g(n)) = \{f(n)$: *there exist positive constants* $c$ *and* $n_0$ *such that* $0 \leq f(n) \leq cg(n)$ *for all* $n \geq n_0\}$

**Definition 1.10** (Big $\Theta$). $\boldsymbol{\Theta(g(n))}$ *is defined as:*
$O(g(n)) = \{f(n)$: *there exist positive constants* $c_1, c_2$ *and* $n_0$ *such that* $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ *for all* $n \geq n_0\}$

**Definition 1.11** (Big $O^*$). $\boldsymbol{O^*(g(n))}$ *is defined as:*
$O^*(g(n)) = f(n)$ *if* $f(n) = O(g(n)poly(n))$, *where* $poly(n)$ *is a polynomial.*

Throughout this work, we determine upper bounds on the worst-case running times of our exact exponential algorithms as functions $O^*(\alpha^n)$ and some real constant $\alpha \geq 1$.

Flow control statements in branching algorithms split entire procedure into so-called reduction rules and branching rules. These rules must have polynomial complexity.

**Definition 1.12** (reduction rule). *A* ***reduction rule*** *simplifies a problem instance or halts the algorithm.*

Reduction rules reduce the size of a current instance. The usually appear before branching rules as they allow for problem simplification without the creation of additional subproblems.

**Definition 1.13** (branching rule). *A* ***branching rule*** *is used to solve a problem instance by recursively solving smaller instances of the problem.*

Let us define a search tree representing an execution of a branching algorithm. We build such trees as follows:

- we assign the root node of the search tree to the input of the problem,
- we recursively assign a child to a node for each smaller instance reached by applying a branching rule to the current instance state.

Let $T(n)$ count the number of leafs of that search tree. The general approach is to analyze each branching rule separately and then to use the worst-case time over all branching rules as an upper bound on the running time of the algorithm.

Let's take a branching rule $b$ which form instance of size $n$ that creates instances of sizes $n - t_1, n - t_2, \ldots, n - t_n$. We can deduce that $T(n)$ will be bounded by the following:

$$T(n) \leq T(n - t_1) + T(n - t_2) + \ldots + T(n - t_r).$$

With this linear recurrence equation we also associate a *branching vector* $b = (t_1, t_2, \ldots, t_r)$. There are well-known standard techniques to solve linear recurrences, however, in this paper we omit this discussion.

Important thing to note is that when at least one $t_i$ has $\Theta(n)$ complexity, the complexity of the entire branch is exponential. Linear recurrence associated with branching vector has an unique solution $\alpha$ but it this article we do not focus on techniques facilitating solving linear recurrences. Instead, we will use a function $\tau$ that takes elements of a branching vector as an argument and returns $\alpha$.

The computational complexity of a branch determined by a branching vector $b = (t_1, t_2, \ldots, t_r)$ is equal to $O^*(\alpha^n)$ where $\tau(t_1, t_2, \ldots, t_r) = \alpha$. Traditionally we estimate $\alpha$ up to four decimal places. A more detailed description is available in [23].

# Chapter 2

# Simple algorithms

## 2.1 Naive Algorithm

The easiest solution to the maximum independent set problem is the naive (otherwise known as the bruteforce) approach. MisNaive algorithm simply checks every possible induced subgraph of the graph $G$. It verifies if this is indeed an independent set and then compares it with the best currently known solution. If the currently analyzed subgraph satisfies these requirements, then the best solution is overwritten.

**Algorithm 1** MisNaive

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
1: procedure MisNaive(graph G)
2:     I ← ∅
3:     for each V' ⊆ V do
4:         if G[V'] has no edge and |V'| > |R| then
5:             I ← V'
6:         end if
7:     end for
8:     return I
9: end procedure
```

### 2.1.1 Correctness

Algorithms iterate over every single subgraph of a $G$ graph and check if it is independent and bigger than the previously found, so the returned set has to be doubtlessly independent and also maximum.

### 2.1.2 Computational complexity

There are $2^n$ subsets of the set of size $n$, and checking whether $E'$ does not contain any edge is polynomial in $n$. Therefore, the complexity is $O^*(2^n)$. It is the slowest possible time for a reasonable algorithm, however for small graphs, it could be the fastest approach due to low overhead.

## 2.2 Mis1

Let us start with a disclaimer that all algorithms presented below, bar the one presented in Section 3.2 are slower than the one published by Tarjan and Trojanowski. Despite that, they are interesting not only from the educational point of view but also as easy-to-implement points of reference for practical measurements – as it is often the case that in practice simple algorithms may outperform the more convoluted ones with better worst-case guarantees.

Let us start with a definition of neighborhood.

**Definition 2.1** (neighborhood)**.** *The set of all neighbors of a vertex $v$ of $G = (V, E)$, denoted by $N(v)$, is called the (open)* ***neighborhood*** *of $v$.*

*By $N[v]$ we denote a closed neighborhood that additionally contains a vertex $v$. In mathematical terms, it is $N[v] = N(v) \cup \{v\}$.*

More generally, by $N^d(v)$ we denote the set of nodes at a distance $d$ from $v$. In particular, $N^1(v) = N(v)$.

**Lemma 2.2.1.** *For any vertex $v \in G$, at least one vertex of $N(v)$ must belong to the maximum independent set of $G$.*

*Proof.* The proof is very straightforward. Let us assume otherwise that the set $I$ is a maximal independent set of $G$ and in $G$ exists $v$ such that no $N[v]$ belongs to $I$. We can add the vertex $v$ to $I$ and create a larger set than $I$ and also maximum. Hence, a contradiction. □

So, for any vertex $v$ we know that one of the $N[v]$ belongs to the solution. Using that observation, we can create a Mis1 algorithm that will hopefully have a better computational complexity than $O^*(2^n)$.

### 2.2.1 Correctness

The algorithm chooses a vertex $v$ and creates $N[v]$ subproblems. Each subproblem is another maximum independent set problem that will have to be solved. We know that some verticle $y$ of $N[v]$ has to be chosen to the independent set, and therefore, one of the subproblems plus the chosen vertex $y$ will give an optimal solution. Eventually, subproblems will be reduced to the size of 0, and for them in fact, the empty set is the maximal independent set. After recursion finishes, it is going to return some maximal independent set.

Generally, we branch over some cases, reduce the original problem to new subproblems and then analyze them exactly as the original problem. This approach is known as *Branch and Reduce*.

**Algorithm 2** MIS1

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
1: procedure MIS1(graph G)
2:     if |V| = 0 then
3:         return ∅
4:     end if
5:     choose arbitrary vertex v of minimum degree in G
6:     S ← {MIS1(G \ N[y]) ∪ {y} : y ∈ N[v]}
7:     return any element of S with the most elements
8: end procedure
```

### 2.2.2 Computational complexity

For MIS1 algorithm, just once, we will do a full analysis of the complexity, including solving a recurrence.

Let each node of the execution tree represent one call of the recursive function. We will be counting the number of nodes. Let $n$ be a number of vertices of graph $G$ and $T(k)$ be a function counting the maximum size of a branching tree for a graph $G$ consisting of $k$ vertices. We can obtain the following recurrence relation for each node:

$$T(n) \leq 1 + \sum_{y \in N[v]} T(n - d(y) - 1)$$

The first 1 in the equation stands for one call before branching. The sum is over $N[v]$ because we branch over the neighbors of chosen $v$. And finally, $n - d(y) - 1$ is the size of the reduced subproblem. From the above recursive relation, we shall bound running time in $O^*$ terms.

From the choice of $v$ (we are choosing minimum degree vertex) we have $n - d(y) - 1 \leq n - d(v) - 1$ and from the monotonic property of $T$ we get the following:

$$\begin{aligned} T(n) &\leq 1 + \sum_{y \in N[v]} T(n - d(y) - 1) \\ &\leq 1 + \sum_{y \in N[v]} T(n - d(v) - 1) \\ &= 1 + T(n - d(v) - 1) \sum_{y \in N[v]} 1 \\ &= 1 + T(n - d(v) - 1)(d(v) + 1) \end{aligned}$$

Let $s = d(v) + 1$. We obtain the following.

$$T(n) \leq 1 + sT(n - s)$$

We can expand $T(n-s)$, and then use $T(0) = 1$ and properties of geometric series

$$\begin{aligned} T(n) &\leq 1 + sT(n-s) \\ &\leq 1 + s + s^2 T(n-2s) \\ &\leq 1 + s + s^2 + \ldots + s^{n/s-1} + s^{n/s} T(0) \\ &= \frac{1 - s^{n/s+1}}{1-s} \\ &= O^*\left(s^{n/s}\right) \end{aligned}$$

Since $s^{n/s} = (s^{1/s})^n$, it is enough to find the maximum of $f(s) = s^{1/s}$ for $s \in \mathbb{N}$. Indeed, $f$ will take the maximum value for $s = 3$ which consequently leads to $O^*(3^{n/s}) \approx O^*(2^{0.5283n}) \approx O^*(1.4422^n)$ running time of our algorithm. This is certainly better than NaiveMis's $O^*(2^n)$.

## 2.3 Mis3, Mis4 and Mis5

### 2.3.1 Special case: graphs with $\Delta(G) \leq 2$

We start by introducing a special case $\Delta(G) \leq 2$ and then use this algorithm in a general solution that improves running time.

It is easy to observe that in graphs with a property $\Delta(G) \leq 2$ we have three types of structures.

- isolated vertices with degree 0,
- paths with lengths $\geq 2$
- cycles with lengths $\geq 3$

Each of these structures is a separate connected component, so we can solve the maximum independent set for them separately and then add results together.

**Definition 2.2** (component)**.** *A **component** of a graph $G$ is its maximal connected induced subgraph.*

**Algorithm 3** PolyMis

**input:** a graph $G = (V, E)$ with $\Delta(G) \leq 2$
**output:** the maximum independent set of $G$

```
1: procedure PolyMis(graph G)
2:     I ← ∅
3:     for each v ∈ V do
4:         visited[v] ← false
5:     end for
6:     for each v ∈ V such that deg(v) = 0 do
7:         I ← I ∪ {v}
8:         visited[v] ← true
9:     end for
10:    for each v ∈ V such that deg(v) = 1 ∧ visited[v] = false do
11:        P ← path containing v
12:        add ⌈||P|/2⌉ not neighboring vertices from P to I
13:        for each v ∈ P do
14:            visited[v] ← true
15:        end for
16:    end for
17:    for each v ∈ V such that visited[v] = false do
18:        C ← cycle containing v
19:        add ⌊||C|/2⌋ not neighboring vertices from C to I
20:        for each v ∈ C do
21:            visited[v] ← true
22:        end for
23:    end for
24:    return I
25: end procedure
```

For the sake of clarity, the algorithm is presented in not the most concise, but still asymptotically optimal (in terms of $O(\cdot)$ notation).

PolyMis algorithm deals with these structures type by type and consecutively adds vertices to $I$. Vertices with degree 0 can be simply added to $I$. For even and odd paths on $n$ vertices, it is easy to spot that the respective maximum independent sets would have $\lceil \frac{n}{2} \rceil$ vertices e.g. by taking every second vertex from one of its ends. Similarly, for cycles on $n$ vertices, one can find an independent set of size $\lfloor \frac{n}{2} \rfloor$.

Every part of the PolyMis algorithm works in a linear time regarding the number of vertices of $G$. It, of course, means that the whole algorithm has polynomial complexity.

Using this algorithm as a subprocedure for solving the general case allows us to branch only on vertices with higher degrees.

### 2.3.2 General case algorithms

We proceed with three quite similar algorithms. We will discuss their correctness and complexity collectively.

**Algorithm 4** MIS3

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
1:  procedure MIS3(graph G)
2:      if ∃v ∈ V : deg(v) = 0 then
3:          return MIS3(G \ {v}) ∪ {v}
4:      end if
5:      if ∃v ∈ V : deg(v) = 1 then
6:          return MIS3(G \ N[v]) ∪ {v}
7:      end if
8:      if Δ(G) ≥ 3 then
9:          choose arbitrary vertex v of maximum degree in G
10:         A ← MIS3(G \ N[v]) ∪ {v}
11:         B ← MIS3(G \ {v})
12:         return largest of the sets: A, B
13:     else
14:         return maximum independent set using Algorithm 3
15:     end if
16: end procedure
```

**Algorithm 5** MIS4

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
1:  procedure MIS4(graph G)
2:      if Δ(G) ≥ 3 then
3:          choose arbitrary vertex v of degree deg(v) ≥ 3 in G
4:          A ← MIS4(G \ N[v]) ∪ {v}
5:          B ← MIS4(G \ {v})
6:          return largest of the sets: A, B
7:      else
8:          return maximum independent set using Algorithm 3
9:      end if
10: end procedure
```

**Algorithm 6** MIS5

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
1: procedure MIS5(graph G)
2:     if Δ(G) ≥ 3 then
3:         choose arbitrary vertex v of maximum degree in G
4:         A ← MIS5(G \ N[v]) ∪ {v}
5:         B ← MIS5(G \ {v})
6:         return the largest of the sets: A, B
7:     else
8:         return maximum independent set using Algorithm 3
9:     end if
10: end procedure
```

### 2.3.3 Correctness

Let's start with the introduction of *standard branching*. This is the default branching method that will be used throughout all the following algorithms. Later on, we will refine it and introduce mirror branching.

**Lemma 2.3.1.** *Let $\alpha$ be an algorithm finding the maximum independent set. Then:*

$$\alpha(G) = max\{\alpha(G \setminus \{v\}), 1 + \alpha(G \setminus N[v])\}$$

It translates into "An algorithm will find the maximum independent set by either discarding $v$ from $G$ or selecting $v$ to the maximum independent set and discarding $N[v]$ from the independent set". $N[v]$ vertices can no longer be a part of a solution after adding $v$ to the independent set. There are no other options of choice, so at least in one of them, an algorithm will find the maximum independent set.

Standard branching can be applied on any vertex in graph $G$ and the correctness will hold. In the case of MIS3, MIS4 and MIS5 an algorithm selects a vertex with a degree of at least 3 be it either the smallest or largest degree depending on the algorithm. Algorithms also save both results for selecting and not selecting $v$ to $A$ and $B$ and then select the set with the most elements.

These three algorithms also slightly differ in the way of dealing with small degree vertices. MIS4 and MIS5 do not do anything special and eventually solve graphs with $\Delta \leq 2$ with a polynomial algorithm. MIS3 is somewhat unique because it deals with vertices of degree 0 and 1 right away. MIS3 adds vertices of degree 0 and 1 to the independent set. Vertices of degree 0 trivially must be added to the maximum independent set. Selecting vertices of degree 1 is also always optimal, as selecting its neighbor – $N(v)$ would lead to the removal of at least $v, N(v)$ and possibly another vertex connected to $N(v)$.

Ultimately, we are left with a graph with a property $\Delta \leq 2$. The polynomial algorithm [**alg:poly**] finishes the job.

### 2.3.4 Computational complexity in Mis3, Mis4 and Mis5

Now, after this introduction, let us try to bound Mis3, Mis4 and Mis5 with that method. Algorithms has only one branching rule when following $\text{Mis}(G \setminus N[v])$ or $\text{Mis}(G \setminus \{v\})$ down the tree. For vertex $v$ this reduces the size of the tree accordingly by $d(v) + 1$ and 1. This implies the recurrence:

$$T(n) \leq T(n - (d(v) + 1)) + T(n - 1)$$

The worst case scenario is going to be when $\deg(v) = 3$, so we get

$$T(n) \leq T(n - 4) + T(n - 1)$$

Instead of solving it like for Mis1 algorithm, we can use the branching vectors method to bound complexity. The branching vector for these three algorithms is $(4, 1)$, which can be computed to $\tau(4, 1) = 1.3803$. Mis3, Mis4, Mis5 all are bounded by $O^*(1.13803^n)$. We will see how they to each other and Mis1 in Chapter 4.

# Chapter 3

# Advanced algorithms

## 3.1 Mis2

In previous chapter we observed that even seemingly more sophisticated algorithms that Mis1 did not achieve better computational complexity. In this chapter we introduce even more rules but most importantly we improve the only branching rule we so far know – standard branching. In the worst case it could generate a branching vector $(1, 4)$. Algorithms here are going to improve this bound, but first, we need to establish more theory.

### 3.1.1 New definitions and lemmas

**Lemma 3.1.1.** *If no maximum independent set of $G$ contains vertex $v$ then every maximum independent set of $G$ contains at least two vertices of $N(v)$.*

*Proof.* We assume that no maximum independent set $I$ of $G$ contains vertex $v$. As a result at least one of the vertices of $N(v)$ must belong to the every $I$. We well show that actually even two vertices must belong to $N(v)$. Let us assume that only one vertex $w$ of $N(v)$ belongs to $I$. Since no other $N(v)$ vertices are in the independent set we can safely shift the selection of vertex added to the independent set from $v$ to $w$. Cardinality of the independent set stays the same so it also must be maximal. However, lemma assumed that no maximum independent set of $G$ contains $v$ hence, a contradiction. □

**Definition 3.1** (clique)**.** *A **clique** is a graph in which every two vertices are adjacent. It is frequently an induced subgraph with the mentioned property.*

**Definition 3.2** (vertex's mirror)**.** *A vertex $w \in N^2(v)$ is called a **mirror** of $v$ if $N(v) \setminus N(w)$ is a clique.*

We denote the set of vertex mirrors of $v$ by $M(v)$.

**Lemma 3.1.2.** *Let $\alpha$ be an algorithm finding the maximum independent set. Then*

$$\alpha(G) = \max\{\alpha(G \setminus \{v\} \setminus M(v)), 1 + \alpha(G \setminus N[v])\}$$

*Proof.* If $G$ has a maximum independent set containing v then $\alpha(G) = 1+\alpha(G \setminus N[v])$ and the lemma is true. Otherwise suppose that no maximum independent set of $G$ contains $v$. From Lemma 3.1.1 we know that if no maximum independent set contains $v$ then every maximum independent set contains at least 2 vertices in $N(v)$. From Definition 3.2 we also know that $N(v) \setminus N(u)$ is a clique and therefore at most one vertex from it can be selected to the independent set.

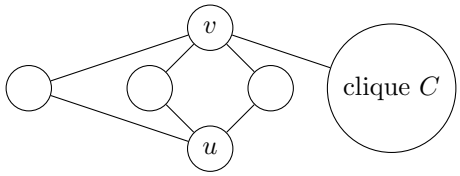



Figure 3.1

The remaining vertex to be selected to the maximum independent set must be in $N(v) \setminus (N(v) \setminus N(u)) = N(v) \cap N(u)$. It will be a neighbor of $w$, and hence, if $v$ is not selected to the independent set, neither is $w$. □

This procedure we will later call simply *mirror branching*.

### 3.1.2 Algorithm

Mis2 algorithm is quite long as it has multiple cases. Many branching rules allow for attaining better computational complexities. The most important thing, however, is to optimize the currently slowest branching rule.

We split the Mis2 algorithm into 3 parts:

- part 1 contains cases for vertices with degrees of 0, 1 and 2,
- part 2 solely focuses on vertices with degree 3,
- part 3 solves remaining cases for vertices with degrees higher than 3.

Additionally, there are ▷'s throughout the code that indicate comments. We will use them during the description of the algorithm to differentiate cases. Also in contrast to the previous chapters, we will be discussing both correctness and computational complexity at the same time.

**Algorithm 7** MIS2

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
1:  procedure MIS2(graph G)
2:      if |V| = 0 then
3:          return ∅                                              ▷ Case A
4:      end if
5:
6:      if ∃v ∈ V : deg(v) ≤ 1 then
7:          return MIS2(G \ N[v]) ∪ {v}                           ▷ Case B
8:      end if
9:
10:     if ∃v ∈ V : deg(v) = 2 then
11:         {u1, u2} ← N(v)
12:         if {u1, u2} ∈ E then
13:             return MIS2(G \ N[v]) ∪ {v}                       ▷ Case C
14:         else
15:             if |N^2(v)| = 1 then
16:                 return MIS2(G \ N[v]) ∪ {u1 u2}               ▷ Case D
17:             else
18:                 A ← MIS2(G \ N[v]) ∪ {v}
19:                 B ← MIS2(G \ {v} \ M(v))
20:                 return the largest of the sets: A, B          ▷ Case E
21:             end if
22:         end if
23:     end if
```

### 3.1.3 Case A and Case B

The first two cases are reduction rules. Case A solves the empty problem. Case B deals with vertex $v$ when its degree is 1. As we discussed earlier, such vertices always can be added to the solution. Both of these cases have only a polynomial impact on the total complexity.

### 3.1.4 Case C

In case C we are dealing with induced triangle. Since we are past Case A and B we know for a fact that vertex $v$ has the lowest degree in a graph. Also, both $u_1$ and $u_2$ will have degrees at least 2. Picking vertex $v$ to the independent set is at least as good as $u_1$ or $u_2$ because choosing any of these three consequently prevents all others from being added to the solution. Then we recursively solve subproblem $G \setminus N[v]$.

This is another reduction rule because we were not creating additional instances.

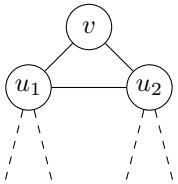



Figure 3.2: Case C example. Only vertices $u_1, u_2$ can have more edges to $V \setminus \{v, u_1, u_2\}$

### 3.1.5 Case D

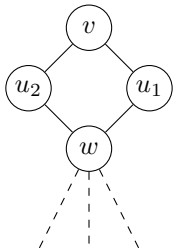



Figure 3.3: Case D example. Only vertex $w$ can have more edges with $V \setminus \{v, u_1, u_2\}$

Here we have a cycle $\{v, u_1, u_2, w\}$ and only $w$ can be connected with the rest of a graph. From these vertices, we have 3 options on how to select them to the independent set but only one of the options is optimal:

1. choose $v$ and $w$. This choice adds two vertices to the independent set and blocks $N[w] \cup \{v\}$ from being chosen down the line in the recursion tree,
2. choose only $v$ and something from $N^3(v)$. Let this vertex from $N^3(v)$ be $x$. This choice adds two vertices to the independent set and blocks $N[x] \cup \{v, u_1, u_2\}$ from being chosen later,
3. choose $u_1$ and $u_2$. This choice also adds two vertices to the independent set and blocks $\{v, u_1, u_2, w\}$ from being chosen later.

The common part of these approaches is that we always add two vertices to the independent set and always remove all four vertices from $G$. However, in options 1 and 2 we also remove additional vertices from graph $G$. From that, we can see that the optimal solution for that case is adding $u_1$ and $u_2$ to the independent set and calling a recursive function for an obtained graph.

This was another reduction rule and thus is not interesting in the sense of computational complexity.

### 3.1.6 Case E

Earlier with Lemma 3.1.2 we proved that when discarding $v$ from the graph it is also safe to discard $M(v)$. Using this rule an algorithm either selects $v$ to the maximum independent set or discards both $v$ and $M(v)$ depending on which attempt gives better results.

If the algorithm selects $v$ to the independent set then with $N(v)$ it removes 3 vertices from a graph. Otherwise, it will remove $v$ and $M(v)$. Fortunately, $M(v) = N^2(v)$. Let us take any vertex $w \in N^2(v)$. It must be a neighbor to either $u_1, u_2$ or both. Then $N(v) \setminus N(w)$ will be $\{u_1\}, \{u_2\}$ or $\emptyset$. In any case, it is a clique so $w \in M(v)$.

After discarding $\{v\} \cup M(v)$ vertices $u_1, u_2$ will have no neighbors and will be immediately selected to the maximum independent set by reduction rule of the Mis2 for vertices with degree 0. $N^2(v)$ has size at least 2 so it that case algorithm will remove at least 5 vertices from graph. Altogether we get a branching vector $(5, 3)$ and complexity $1.1939^n$.

**Algorithm 8** Mis2

```
24:  if there exists v ∈ V such that deg(v) = 3 then
25:      {u1, u2, u3} ← N(v)
26:      if G[N(v)] has no edge then
27:          if v has a mirror then
28:              A ← Mis2(G \ N[v]) ∪ {v}
29:              B ← Mis2(G \ {v} \ M(v))
30:              return the largest of the sets: A, B                ▷ Case F
31:          else
32:              A ← Mis2(G \ N[v]) ∪ {v}
33:              B ← Mis2(G \ N[u1] \ N[u2]) ∪ {u1, u2}
34:              C ← Mis2(G \ N[u1] \ N[u3] \ {u2}) ∪ {u1, u3}
35:              D ← Mis2(G \ N[u2] \ N[u3] \ {u1}) ∪ {u2, u3}
36:              return the largest of the sets: A, B, C, D          ▷ Case G
37:          end if
38:      else if G[N(v)] has one or two edges then
39:          A ← Mis2(G \ N[v]) ∪ {v}
40:          B ← Mis2(G \ {v} \ M(v))
41:          return the largest of the sets: A, B                    ▷ Case H
42:      else
43:          return Mis2(G \ N[v]) ∪ {v}                             ▷ Case I
44:      end if
45:  end if
```

### 3.1.7 Case F

We have a vertex with a degree 3 and set $N(v)$ has no edges and $v$ has a mirror. Since $v$ has a mirror, we just perform another mirror branching. In

this case, the branching vector is equal to $(4, 2)$ because in the case of vertex $v$ being selected to the independent set we can discard exactly 4 vertices from the $V(G)$. On the other hand, if $v$ is not selected then we discard $v$ and $M(v)$. It can be shown that $\tau(4, 2) < 1.2721$.

### 3.1.8 Case G

We have a vertex with a degree 3 and set $N(v)$ has no edges but $v$ does not have a mirror. We consider all the cases. The algorithm can:

- select $v$ – subproblem has 4 vertices less,
- discard $v$, select $u_1$, discard $u_2$, select $u_3$ – subproblem has at least 8 vertices less, because we reduce the problem by $\{v, u_1, u_2, u_3\} \cup N(u_1) \cup N(u_2)$. We are using an assumption that there are no more vertices with degrees 0,1 or 2 left,
- discard $v$, discard $u_1$, select $u_2$, select $u_3$ – the same as above,
- discard $v$, select $u_1$, select $u_2$ – similarly as above but without removing $u_3$, Here it might be surprising that we don't discard vertex $u3$ like in previous cases. However, here we allow for both selecting and discarding $u_3$ further down the recursion tree.

This leaves us with $(4, 7, 8, 8)$ branching vector and $\tau(4, 7, 8, 8) < 1.2406$.

### 3.1.9 Case H

A case with degree 3 and one or two edges between $N(v)$. Here is another example where we use mirror branching. Complexity analysis, however, splits this case into two:

- $d(v) = 3$ and there is one edge between $N(v)$
  Let this edge be $\{u_1, u_2\}$ then vertex $u_3$ will have at least two neighbors in $N^2(v)$ and they will be mirrors of $v$ since $\{u_1, u_2\}$ forms a clique. Not selecting $v$ also guarantees not selecting these two,
- $d(v) = 3$ and there are two edges between $N(v)$
  Let them be $\{u_1, u_2\}, \{u_1, u_3\}$. As proven in the lemmas, discarding $v$ will leave $u_1$ and $u_3$ to be selected. That removes at least 5 vertices in total.

In both cases selecting vertex $v$ to an independent set simplifies the problem by removing $N[v]$ - that is 4 vertices. This leads to branching vectors $(4, 3)$ and $(4, 5)$. Both of them are worse than the $(4, 2)$ that we encountered before.

### 3.1.10 Case I

$N(v)$ forms a clique. Here we can simply select vertex $v$ since selecting either one of $u_1, u_2, u_3$ would yield a worse result. It was a reduction rule.

**Definition 3.3** (graph $k$-regular)**.** *A graph is* ***k-regular*** *when all of its vertices have degree $k$.*

**Algorithm 9** MIS2

```
46:
47:     if Δ(G) ≥ 6 then
48:         choose arbitrary vertex v of maximum degree in G
49:         A ← MIS2(G \ N[v]) ∪ {v}
50:         B ← MIS2(G \ {v})
51:         return the largest of the sets: A, B                    ▷ Case J
52:     end if
53:
54:     if G has multiple components then
55:         choose arbitrary component C of G
56:         return MIS2(G[C]) ∪ MIS2(G \ C)                         ▷ Case K
57:     end if
58:
59:     if G is 4-regular or 5-regular then
60:         choose arbitrary vertex v of G
61:         A ← MIS2(G \ N[v]) ∪ {v}
62:         B ← MIS2(G \ {v} \ M(v))
63:         return the largest of the sets: A, B                    ▷ Case L
64:     end if
65:
66:     choose arbitrary adjacent vertices v and w with deg(v) = 5 and deg(w) =
    4 in G
67:     A ← MIS2(G \ N[v]) ∪ {v}
68:     B ← MIS2(G \ {v} \ M(v))
69:     return the largest of the sets: A, B                        ▷ Case M
70: end procedure
```

### 3.1.11 Case J

This case is pretty simple and similar to what we saw before in the other algorithms. Because this algorithm has many cases we can save up on the complexity by considering vertices of degree at least 6. The algorithm splits the problem into subproblems where either $v$ is included in the solution or not. This gives branching a vector $(7, 1)$ and $\tau(7, 1) < 1.2554$.

### 3.1.12 Case K

This case focuses on splitting disconnected graphs into subproblems solving single components. Let's say we have a graph with $n$ vertices and it contains connected components containing $c \geq 1$ (actually $c \geq 4$ but it does not matter)

vertices. Then we will get the following recurrence relation:

$$T(n) \leq T(n-c) + T(c)$$

Since the recurrence gives only a polynomial bound, this has no impact on the overall complexity.

### 3.1.13 Case L

In case L we either encounter a 4-regular graph or a 5-regular graph.

We will prove the following lemma:

**Lemma 3.1.3.** *On its execution path for any graph $G$, Algorithm 9 encounters at most one $r$-regular graph for each $r$.*

*Proof.* Let us say an algorithm, reached a point where it sees $r$-regular graph $R_1$. It must be a connected component since it is guaranteed after the previous Case. Assume that we remove some vertices from this graph and we get another $r$-regular graph $R_2$. Again, it has to be a connected component. It is also must be an induced subgraph of the original $r$-regular graph. Number of edges for each vertex is fixed and equal to $r$. It means that there can't be any edges between graphs $R_1$ and $R_2 \setminus R_1$. But $R_2$ is a subgraph of $R_1$ which means that $R_1$ and $R_2 \setminus R_1$ are separate components. Hence, a contradiction. □

Thus, we have just proved that we can only reach this case once for a 4-regular graph and once for a 5-regular graph. It means that this case can be ignored in the complexity discussion as it has at most a polynomial impact.

### 3.1.14 Case M

We are at the end. We are left with a graph in which must exist at least one vertex with degree 4 and at least one vertex with degree 5. We also know that the graph is a connected component so there must be a pair $(v, w)$ such that $\deg(v) = 5$ and $\deg(w) = 4$ and $w \in N(v)$ ($v, w$ have different degrees and are neighbors). With this setting an algorithm performs mirror branching on $v$. This leads to a branching vector of $(1, 6)$. It can be calculated that $\tau(1, 6) < 1.2852$.

This is the worst estimation encountered so far but fortunately, we can improve it. We want to optimize a branch case where $v$ is discarded. We need to look deeper down the recursion tree to see what happens. After $v$ is discarded vertex $w$ loses one neighbor and reduces its degree to 3. This can potentially lead to any of the cases for a degree of 3. Cases F, G, H, and I have respective branching vectors: $(4, 2)$, $(4, 3)$ $(4, 5), (4, 7, 8, 8)$. As the algorithm also passes through original node that was giving $(1, 6)$ it consequently corresponds to one of these branching vectors: $(4+1, 2+1, 6) = (5, 3, 6)$, $(4+1, 3+1, 6) = (5, 4, 6)$, $(4+1, 5+1, 6) = (5, 6, 6)$, $(4+1, 7+1, 8+1, 8+1, 6) = (5, 8, 9, 9, 6)$. The slowest of is a branching vector $(5, 8, 9, 9, 6)$ and $\tau(5, 8, 9, 9, 6)$ gives 1.2786. Interestingly, the algorithm did not change but more careful analysis led to a better result. Indeed, the complexity of the algorithm is only as good as we can prove it.

### 3.1.15 Overview

Here is an overview of all cases of Mis2 algorithm and their complexities. The bold value represents the worst-case branching scenario and therefore overall complexity.

| general case | specific case | branching vector | branching factor |
|---|---|---|---|
| | $\deg(v) = 0$ | — | 1 |
| | $\deg(v) = 1$ | — | 1 |
| $\deg(v) = 2$ | $N(v)$ connected | — | 1 |
| | $\lvert E[N(v)]\rvert = 0 \wedge \lvert N^2(v)\rvert = 1$ | — | 1 |
| | $\lvert E[N(v)]\rvert = 0 \wedge \lvert N^2(v)\rvert \neq 1$ | (5,3) | 1.1939 |
| $\deg(v) = 3$ | $N(v)$ no edge $\wedge$ $v$ has mirror | (4,2) | 1.2721 |
| | $N(v)$ no edge $\wedge$ $v$ no mirror | (4,7,8,8) | 1.2406 |
| | $N(v)$ one or two edges | (4,3) | $< 1.2721$ |
| | $N(v)$ three edges | — | 1 |
| | $\Delta(G) \geq 6$ | (7,1) | 1.2554 |
| | $G$ has multiple components | — | 1 |
| | $G$ is 4 or 5 regular | — | 1 |
| | $G$ has 4/5 degree vertices | (5,8,9,9,6) | **1.2786** |

## 3.2 MisFolding

This algorithm solves the maximum independent set problem asymptotically faster and is much shorter than Mis2. It utilizes new technique called vertex folding. Additionally, it is designed around improving the complexity of the slowest part of the algorithm and all unnecessary cases are not included. Mis2 could similarly be stripped from some cases and computational complexity would not change. Real world computation time however, would likely increase because simple cases can be frequently solved quickly.

In this section, we will be discussing the vertex folding technique and algorithm correctness. Curious readers can read complexity analysis from [13].

Let us start with some additional definitions:

**Definition 3.4** (anti-edge)**.** *An **anti-edge** is a pair of vertices that belong to the graph but are not adjacent. Anti-edges exist for every pair of vertices that are not connected by an edge.*

**Definition 3.5** (anti-clique)**.** *A **anti-clique** is a graph in which every two vertices are not adjacent. It is frequently an induced subgraph with the mentioned property. **Anti-triangle** is the anti-clique on* 3 *vertices.*

**Definition 3.6** (foldable vertex)**.** *A vertex $v$ is **foldable** if $N(v) = \{u_1, u_2, \ldots, u_{\deg(v)}\}$ contains no anti-triangles.*

**Definition 3.7** (An procedure for folding a vertex)**.** *Folding of a given vertex $v$ of $G$ is the process of transforming $G$ into a new graph $\tilde{G} = \tilde{G}(v)$ by following these steps:*

1. *adding a new vertex $u_{i,j}$ for each anti-edge $\{u_i, u_j\} \in N(v)$*
2. *adding edges between each $u_{i,j}$ and the vertices in $N(u_i) \cup N(u_j)$*
3. *adding edges between new vertices to make a clique*
4. *remove $N[v]$ (old vertices)*

Note that when we fold a vertex $v$ of degree either zero or one, we simply remove $N[v]$ from the graph.


Figure 3.4: An example of folding vertex $v$

It is also worth noting that in our modified algorithm it is necessary to also remember how exactly new folded vertices were formed. Selecting a folded vertex to a maximum independent set actually selects two vertices after a reverse operation of unfolding them. Folded vertices were originally not connected, so we have a guarantee that we can select them both.

Now let us take a look at the algorithm Algorithm 10

**Algorithm 10** MisFolding

**input:** a graph $G = (V, E)$
**output:** the maximum independent set of $G$

```
 1: procedure Mis5(graph G)
 2:     if |V| = 0 then
 3:         return ∅
 4:     end if
 5:     if G has multiple components then
 6:         choose arbitrary component C of G
 7:         return MisFolding(G[C]) ∪ MisFolding(G \ C)
 8:     end if
 9:     if exist v, w ∈ V such that N[w] ⊂ N[v] then
10:         return MisFolding(G \ {v})
11:     end if
12:     if ∃ a foldable vertex v : deg(v) ≤ 4 and N(v) contains at most 3 anti-
    edges then
13:         choose one such vertex v of minimum degree
14:         create graph G̃ by folding vertex v in graph G
15:         F ← MisFolding(G̃)
16:         if F contains u, one of the newly folded vertices then
17:             return F \ {u} ∪ {two vertices from G for which u was added }
18:         else
19:             return F ∪ {v}
20:         end if
21:     end if
22:     choose arbitrary maximum degree vertex v of G
23:     A ← MisFolding(G \ N[v]) ∪ {v}
24:     B ← MisFolding(G \ {v} \ M(v))
25:     return MostNumerous(A,B)
26: end procedure
```

There are just a few cases:

- *(line 2)* graph is empty. It simply solves an empty case. We have seen this before.
- *(line 5)* algorithm splits graph into multiple components. We have also seen this in Mis2.
- *(line 8)* we have two vertices $u, w$ with a property $N[w] \subset N[v]$. This is a generalized Case C from the Mis2 discussion. We need to prove that it is always safe to discard $v$ and leave $w$ to be selected or not. $v$ and $w$ are neighbors so both can't be selected to the independent set. If $v$ belonged to the maximum independent $I$ set then we have a guarantee that no other vertex from $N[v]\{v\}$ belongs to $I$. But we know that $N[w] \subset N[v]$ so also no other vertex from $N[w]\{v\}$ belongs to $I$. Since that is the case, we can

always shift our selection of vertex to the maximum independent set from $v$ to $w$ and so $v$ can be discarded.

- *(line 12)* the first and also the last interesting case. An algorithm picks a minimum degree vertex $v$. Then it performs folding according to the algorithm discussed in this section.

MisFolding procedure is called recursively on the modified graph. After an algorithm returns, the returned set is investigated. If it contains one of the recently folded vertex it means that we have to replace it with the corresponding original vertices from the original graph. Then we return an obtained set. Otherwise, select $v$ to the independent set and return. Folding a vertex is a general procedure but in this algorithm, authors perform it on very specific cases. This selection of vertices to be folded provides the lowest complexity.

*(line 22–25)* an algorithm is performing mirror branching on one of the remaining vertices.

# Chapter 4

# Implementation and benchmark

## 4.1 Implementation

All algorithms were implemented using C++20 for an open-source KOALA NetworKit library [36].

### 4.1.1 KOALA NetworKit

Short descriptions found in the README file in the cited repository accurately describe the building blocks of the project.

- **NetworKit** is an open-source general-purpose network analysis and graph mining library written in C++. Graphs are represented in a very compact form while the efficiency of changes to their structure, such as node and edge additions or deletions, is preserved. The memory-saving design is related to the main aim of the library: the analysis of large-scale random graphs and real-world networks.
- **KOALA** is an open-source library of C++ templates, developed at the Gdansk University of Technology, Department of Algorithms and System Modeling. Its main part consists of an implementation of a broad set of procedures in the fields of algorithmic graph theory and network problems in discrete optimization.

The project also invokes functions of **Boost** library and uses **GoogleTest** for unit testing. Additionally, **nauty** package is used for generating and reading random test graphs.

### 4.1.2 Implementation details and expectations

Major changes were added in the following files:

- `include/mis/IndependentSet.hpp` – header file with all necessary function declarations
- `cpp/mis/IndependentSet.cpp` – source file with helper functions for all algorithms as well as the implementation of MisNaive
- `cpp/mis/ExactRecursiveIndependentSet.cpp` – source file with implementation of all recursive algorithms
- `test/testIndependentSet.cpp` – simple GoogleTest tests added mainly to improve user understanding and experience as well as provide some basic testing
- `benchmark/benchmarkIndependentSet.cpp` – an independent program that allows for extensive benchmark testing of all algorithms using graphs generated from [25]

Almost all algorithms except MisNaive intensively use recursive calls. For that reason, we decided to avoid copying `NetworKit::graph` and instead, we modify it before a recursive call and revert changes after. Generally speaking, `Networkit::graph` structure does not perform well in situations where the graph is heavily modified. Algorithms also can not take full advantage of a reduced size of a problem because of how NetworKit is implemented. There is no efficient way to resize a graph (and resize back) for smaller subgraphs of the original problem.

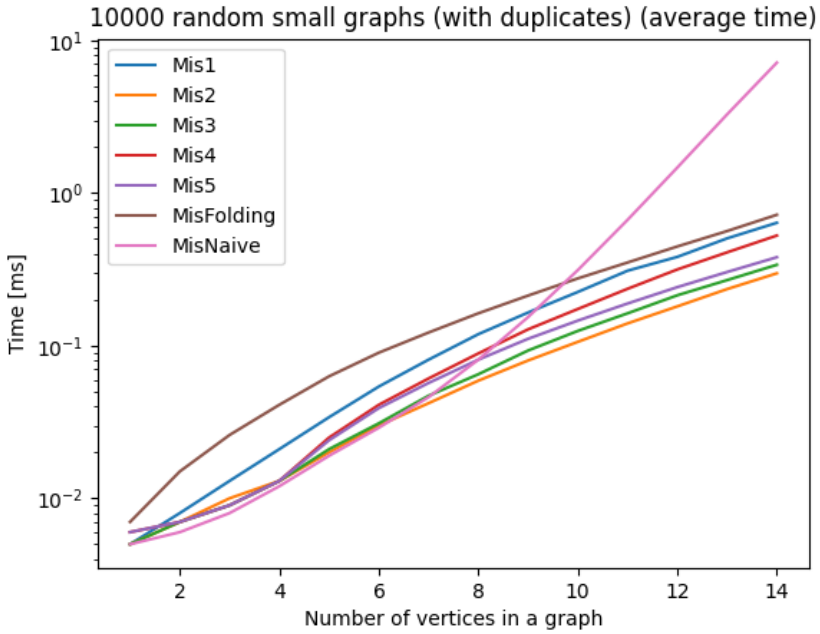
For these reasons, it is worth noting that we can expect high overhead for MisFolding algorithm since it modifies graphs the most. Besides that, a polynomial factor of complexity, which was completely ignored in the article, for MisFolding is quite high. It is to be expected that Mis2 to perform the best for relatively small graphs but hopefully will be outperformed by MisFolding for larger ones. Mis3, Mis4, Mis5 will likely perform almost the same. Mis1 will probably be the second slowest after MisNaive. MisNaive will even have to be skipped in computationally demanding benchmark tests due to unworldly long execution times but it should be the fastest for very small graphs.

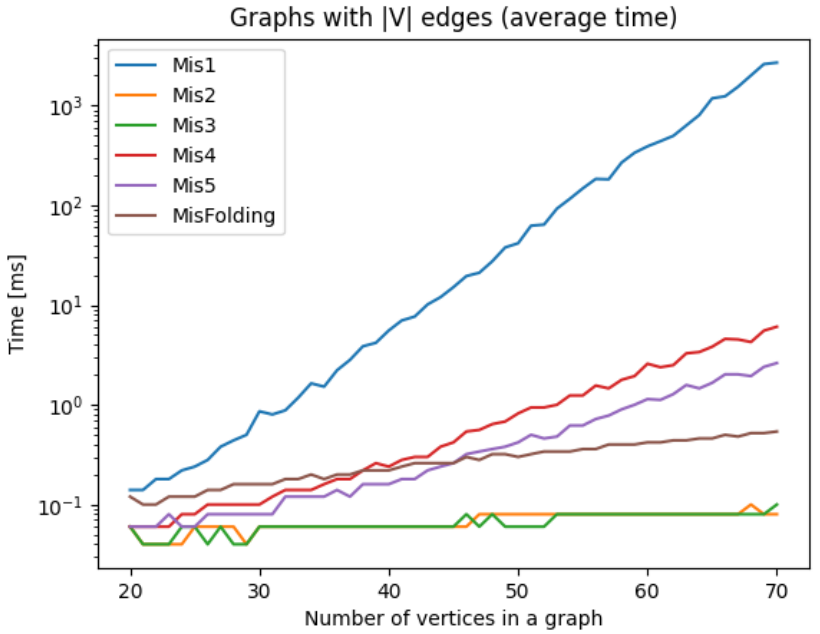
### 4.1.3 Correctness

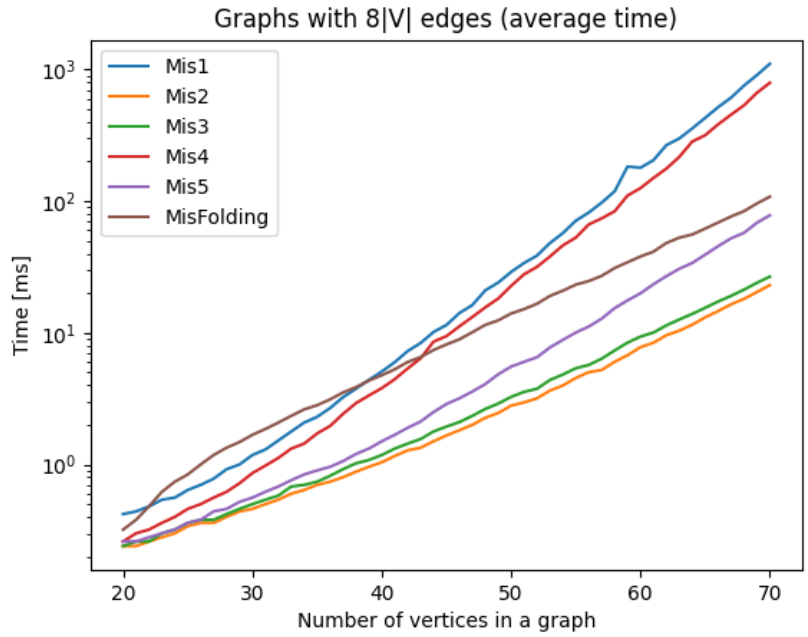
Each of the algorithm solutions is verified whether it is an independent set. It means that for every returned solution $I$ for test graph $G = (V, E)$ we make sure that there are no vertices $u, v \in I$ such that there is an edge $\{u, v\} \in E$.

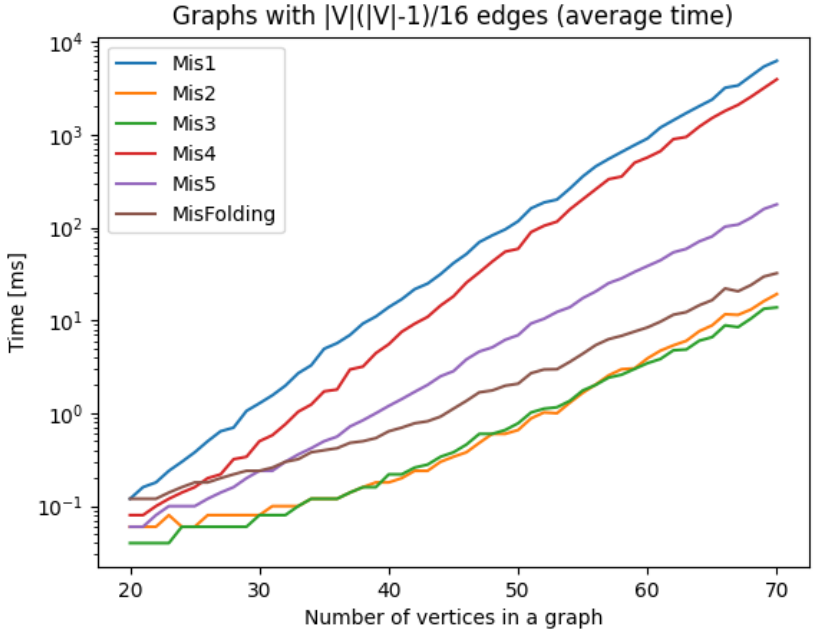
Additionally, we check whether all algorithms for the same test, return the same maximum independent set size. The indispensable algorithm here is MisNaive as it is pretty much impossible to be incorrect due to its simplicity.

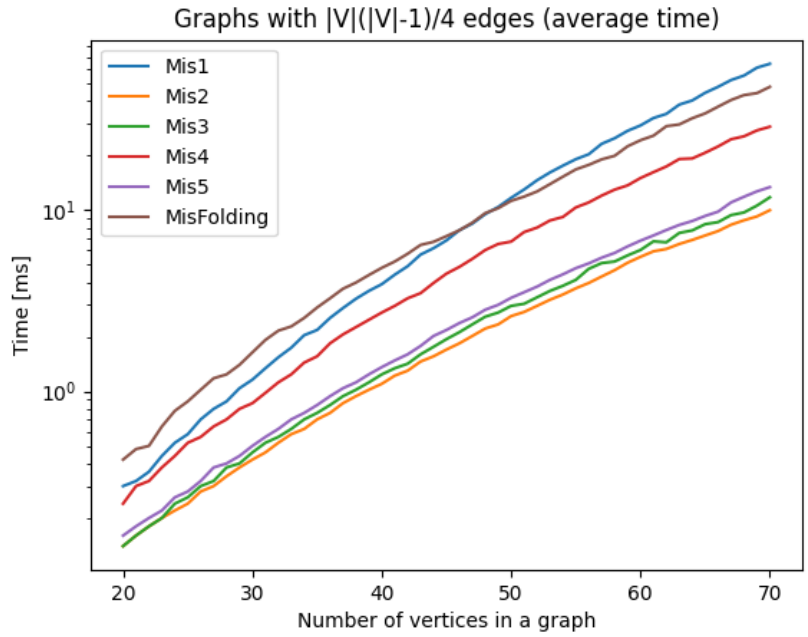
## 4.2 Benchmark results

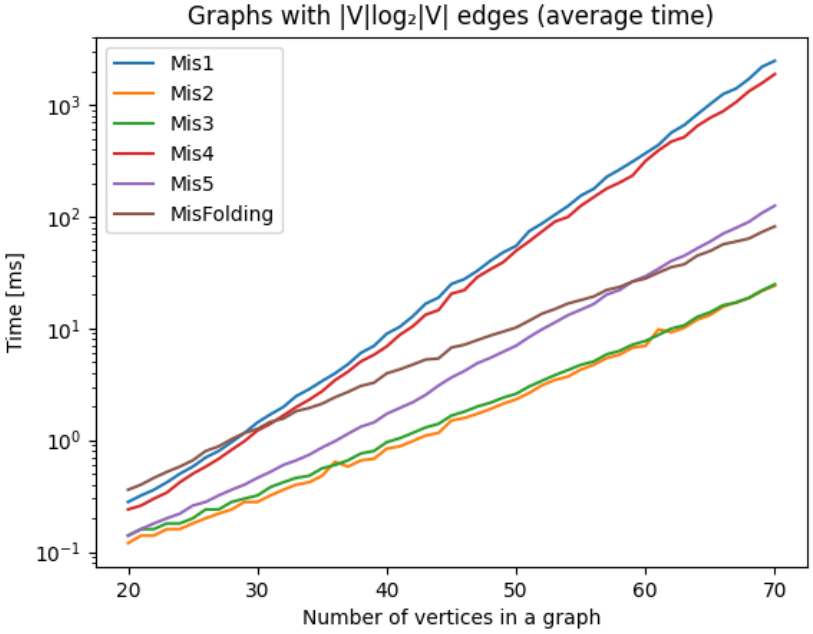


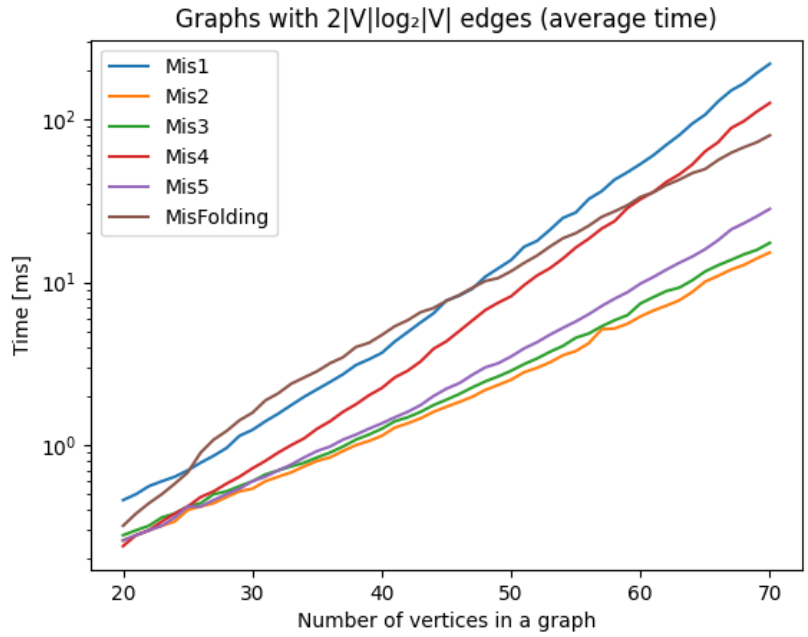



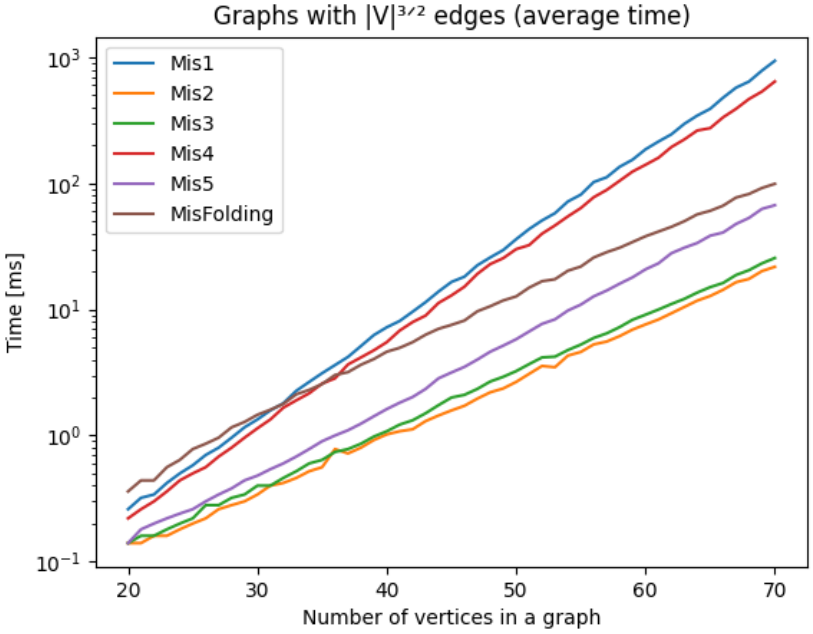

Graphs with 8|V| edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
$10^3$
$10^2$
$10^1$
$10^0$
20
30
40
50
60
70
Number of vertices in a graph

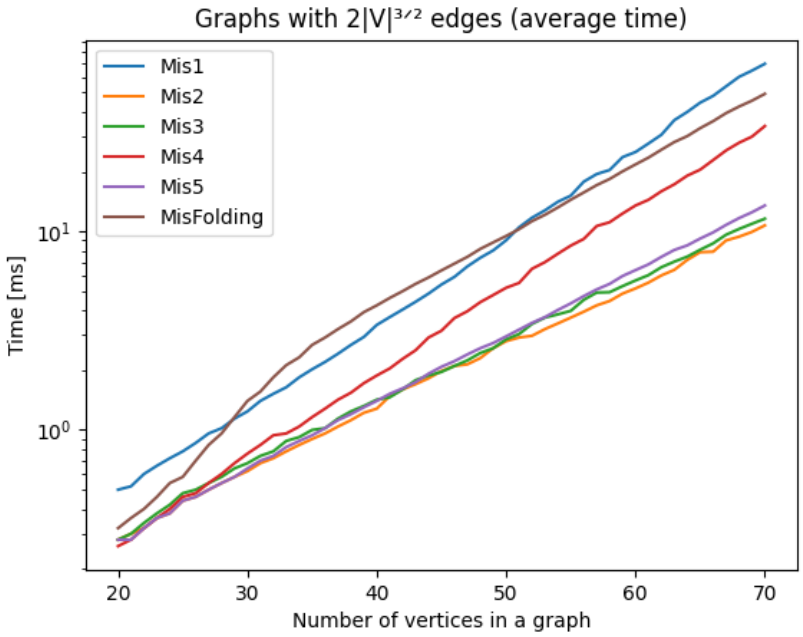

Graphs with |V|(|V|-1)/16 edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
$10^4$
$10^3$
$10^2$
$10^1$
$10^0$
$10^{-1}$
20
30
40
50
60
70
Number of vertices in a graph

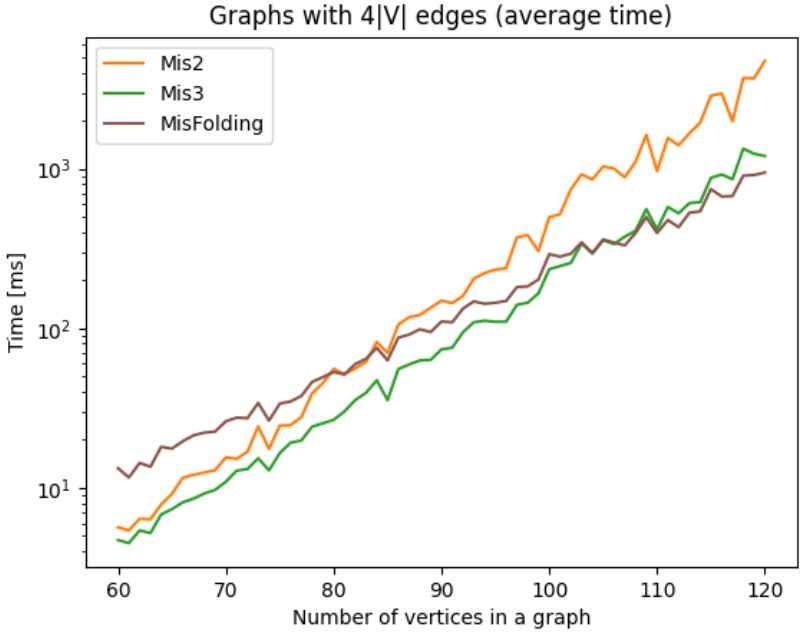

Graphs with |V|(|V|-1)/4 edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
$10^1$
$10^0$
20
30
40
50
60
70
Number of vertices in a graph

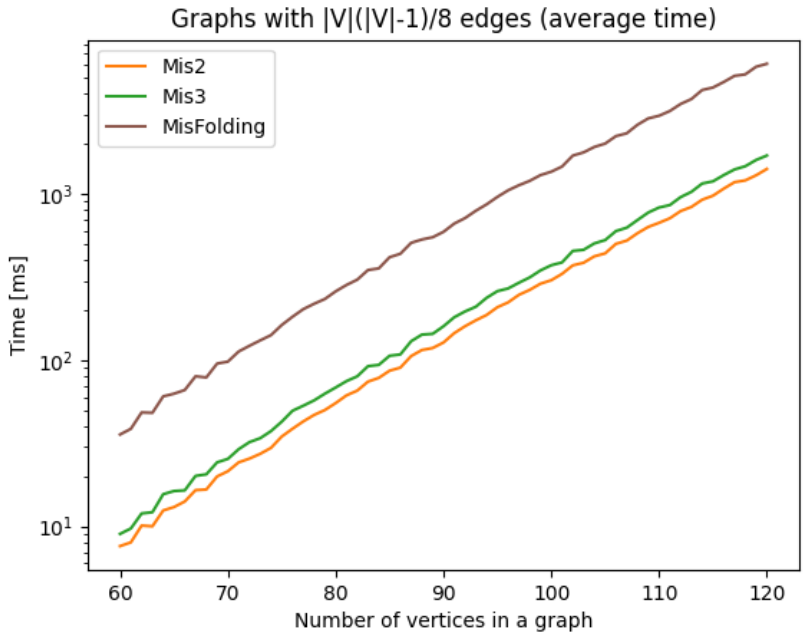

Graphs with |V|log₂|V| edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
$10^3$
$10^2$
$10^1$
$10^0$
$10^{-1}$
20
30
40
50
60
70
Number of vertices in a graph

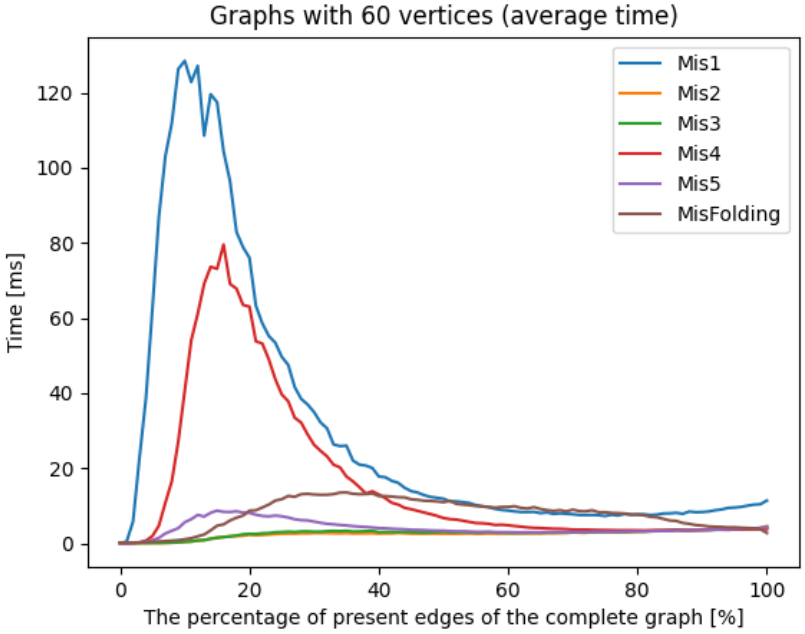

Graphs with 2|V|log₂|V| edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
$10^2$
$10^1$
$10^0$
20
30
40
50
60
70
Number of vertices in a graph

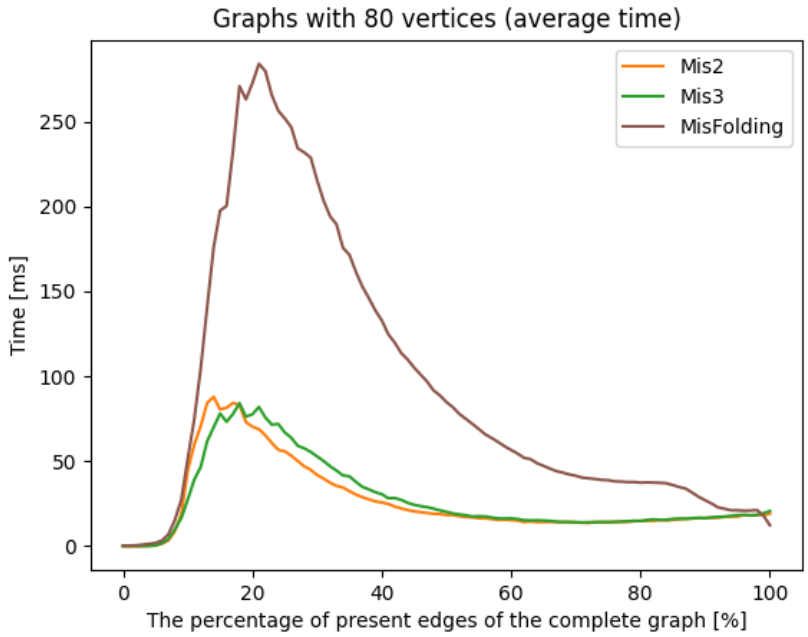

Graphs with |V|³ᐟ² edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
$10^3$
$10^2$
$10^1$
$10^0$
$10^{-1}$
20
30
40
50
60
70
Number of vertices in a graph


Graphs with 2|V|³/² edges (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
Time [ms]
10¹
10⁰
20
30
40
50
60
70
Number of vertices in a graph


Graphs with 4|V| edges (average time)
Mis2
Mis3
MisFolding
Time [ms]
10³
10²
10¹
60
70
80
90
100
110
120
Number of vertices in a graph


Graphs with |V|(|V|-1)/8 edges (average time)
Mis2
Mis3
MisFolding
$10^3$
$10^2$
$10^1$
Time [ms]
60
70
80
90
100
110
120
Number of vertices in a graph


Graphs with 60 vertices (average time)
Mis1
Mis2
Mis3
Mis4
Mis5
MisFolding
120
100
80
60
40
20
0
Time [ms]
0
20
40
60
80
100
The percentage of present edges of the complete graph [%]




From the benchmarks performed, we can see that MisNaive is the slowest already for graphs as small as 10 vertices.

Mis2 and Mis3 are almost always the fastest for graphs up to about 100 vertices. Mis3 algorithm is much simpler than Mis2 but it offers a similar performance. To have a high-performance algorithm for graphs of these sizes (without much hassle) it is probably best to implement Mis3 algorithm as well as data structures from scratch. NetworKit library has its limitations because it has to work well for all purposes. It does not perform well when graphs are heavily modified and recursion in our algorithms removes all vertices and edges.

As for graphs of higher sizes, we can see that MisFolding slowly gains an edge over Mis2 and Mis3, at least in some specific cases. Unfortunately, we were not able to test it for graphs with larger order due to the long computing time.

# Bibliography


[1] Brenda Baker. "Approximation algorithms for NP-complete problems on planar graphs". In: *Journal of the ACM* 41.1 (1994), pp. 153–180.

[2] Piotr Berman and Georg Schnitger. "On the complexity of approximating the independent set problem". In: *Information and Computation* 96.1 (1992), pp. 77–94.

[3] Béla Bollobás. *Modern graph theory*. Vol. 184. Springer Science & Business Media, 1998.

[4] Édouard Bonnet, Nicolas Bousquet, Stéphan Thomassé, and Rémi Watrigant. *When maximum stable set can be solved in FPT time*. arXiv preprint arXiv:1909.08426. 2019. URL: `https://arxiv.org/abs/1909.08426`.

[5] Nicolas Bourgeois, Bruno Escoffier, Vangelis Paschos, and Johan van Rooij. "A bottom-up method and fast algorithms for max independent set". In: *Algorithm Theory-SWAT 2010: 12th Scandinavian Symposium and Workshops on Algorithm Theory, Bergen, Norway, June 21-23, 2010. Proceedings*. Vol. 12. Springer. 2010, pp. 62–73.

[6] Sergiy Butenko. *Maximum independent set and related problems, with applications*. University of Florida, 2003.

[7] Timothy Chan and Sariel Har-Peled. "Approximation algorithms for maximum independent set of pseudo-disks". In: *Proceedings of the twenty-fifth annual symposium on Computational geometry*. 2009, pp. 333–340.

[8] Brent Clark, Charles Colbourn, and David Johnson. "Unit disk graphs". In: *Discrete Mathematics* 86.1–3 (1990), pp. 165–177.

[9] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 2022.

[10] Bruno Courcelle, Johann Makowsky, and Udi Rotics. "Linear time solvable optimization problems on graphs of bounded clique-width". In: *Theory of Computing Systems* 33.2 (2000), pp. 125–150.

[11] Gautam Das, Minati De, Sudeshna Kolay, Subhas Nandy, and Susmita Sur-Kolay. "Approximation algorithms for maximum independent set of a unit disk graph". In: *Information Processing Letters* 115.3 (2015), pp. 439–446.

[12] Thomas Erlebach, Klaus Jansen, and Eike Seidel. "Polynomial-time approximation schemes for geometric intersection graphs". In: *SIAM Journal on Computing* 34.6 (2005), pp. 1302–1323.

[13] Fedor Fomin, Fabrizio Grandoni, and Dieter Kratsch. "Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm". In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*. 2006, pp. 18–25.

[14] András Frank. "Some polynomial algorithms for certain graphs and hypergraphs". In: *Congressus Numerantium* XV (1976), pp. 221–226.

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

[16] Michael Garey and David Johnson. "The rectilinear Steiner tree problem is NP-complete". In: *SIAM Journal on Applied Mathematics* 32.4 (1977), pp. 826–834.

[17] Michael Garey, David Johnson, and Larry Stockmeyer. "Some simplified NP-complete problems". In: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*. 1974, pp. 47–63.

[18] Fănică Gavril. "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph". In: *SIAM Journal on Computing* 1.2 (1972), pp. 180–187.

[19] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric algorithms and combinatorial optimization*. Vol. 2. Springer, 1988.

[20] Edin Husic, Stéphan Thomassé, and Nicolas Trotignon. *The independent set problem is FPT for even-hole-free graphs*. arXiv preprint arXiv:1907.01083. 2019. URL: `https://arxiv.org/abs/1907.01083`.

[21] Tang Jian. "An $O(2^{0.304n})$ algorithm for solving maximum independent set problem". In: *IEEE Transactions on Computers* 100.9 (1986), pp. 847–851.

[22] Joachim Kneis, Alexander Langer, and Peter Rossmanith. "A fine-grained analysis of a simple independent set algorithm". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2009, pp. 287–298.

[23] Dieter Kratsch and Fedor Fomin. *Exact exponential algorithms*. Springer-Verlag Berlin Heidelberg, 2010.

[24] Daniel Lokshantov, Martin Vatshelle, and Yngve Villanger. "Independent set in $P_5$-free graphs in polynomial time". In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. 2014, pp. 570–581.

[25] Brendan McKay. *Graphs*. 2023. URL: `https://users.cecs.anu.edu.au/~bdm/data/graphs.html` (visited on 09/07/2023).

[26] George Minty. “On maximal independent sets of vertices in claw-free graphs”. In: *Journal of Combinatorial Theory, Series B* 28.3 (1980), pp. 284–304.

[27] Bojan Mohar. “Face covers and the genus problem for apex graphs”. In: *Journal of Combinatorial Theory, Series B* 82.1 (2001), pp. 102–117.

[28] John Moon and Leo Moser. “On cliques in graphs”. In: *Israel Journal of Mathematics* 3 (1965), pp. 23–28.

[29] Owen Murphy. “Computing independent sets in graphs with large girth”. In: *Discrete Applied Mathematics* 35.2 (1992), pp. 167–170.

[30] Daishin Nakamura and Akihisa Tamura. “A revision of Minty’s algorithm for finding a maximum weight stable set of a claw-free graph”. In: *Journal of the Operations Research Society of Japan* 44.2 (2001), pp. 194–204.

[31] Tim Nieberg, Johann Hurink, and Walter Kern. “A robust PTAS for maximum weight independent sets in unit disk graphs”. In: *Graph-Theoretic Concepts in Computer Science: 30th International Workshop, WG 2004, Bad Honnef, Germany, June 21-23, 2004. Revised Papers*. Vol. 30. Springer. 2005, pp. 214–221.

[32] John Michael Robson. “Algorithms for maximum independent sets”. In: *Journal of Algorithms* 7.3 (1986), pp. 425–440.

[33] Kenneth Rosen. *Discrete mathematics and its applications*. The McGraw Hill Companies, 2007.

[34] Robert Endre Tarjan. *Finding a maximum clique*. Tech. rep. Cornell University, 1972.

[35] Robert Endre Tarjan and Anthony Trojanowski. “Finding a maximum independent set”. In: *SIAM Journal on Computing* 6.3 (1977), pp. 537–546.

[36] Krzysztof Turowski. *KOALA NetworKit*. 2023. URL: `https://github.com/krzysztof-turowski/koala-networkit` (visited on 09/07/2023).

[37] Wikipedia. *Glossary of graph theory*. 2023. URL: `https://en.wikipedia.org/wiki/Glossary_of_graph_theory` (visited on 08/23/2023).

[38] David Williamson and David Shmoys. *The design of approximation algorithms*. Cambridge University Press, 2011.

[39] Mingyu Xiao and Hiroshi Nagamochi. “Exact algorithms for maximum independent set”. In: *Information and Computation* 255 (2017), pp. 126–146.

[40] David Zuckerman. “On unapproximable versions of NP-complete problems”. In: *SIAM Journal on Computing* 25.6 (1996), pp. 1293–1304.