RWTH Aachen

Department of Computer Science
Technical Report

Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm

Felix Reidl, Fernando Sánchez Villaamil

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2009-10

RWTH Aachen · Department of Computer Science · ——— —-

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm

Felix Reidl, Fernando Sánchez Villaamil

Dept. of Computer Science
RWTH Aachen University, Germany
Email: {`felix.reidl, fernando.sanchez`}`@rwth-aachen.de`

Abstract. Kneis, Langer and Rossmanith presented a new exact algorithm for the `Maximum Independent Set` problem [3]. As part of the proof of the upper runtime bound millions of special cases where automatically generated. In this technical report we present a verification method that checks the correctness of this case distinction. We focus on the theoretical aspects of this verification and give a general overview of its implementation.

1 Introduction

`Maximum Independent Set` is one of the most well-known NP-hard problems in the field of theoretical computer science. Due to its importance, several exact algorithms have been developed for it; the first non-trivial one dating back to 1977: Tarjan and Trojanowski [8] achieved a runtime bound of $O^*(1.261^n)$. Jian [4] and Robson [7] improved this further to $O^*(1.235)$ and $O^*(1.228)^n)$, respectively. In the same paper, Robson was also able to proof an upper bound of $O^*(1.211^n)$ using exponential space. Fomin, Grandoni and Kratsch [2] employed their Measure & Conquer technique to devise a new algorithm for `Maximum Independent Set` with a runtime bounded by $O^*(1.2201^n)$. Only recently, Kneis, Langer and Rossmanith [3] used a similar approach with an even better bound of $O^*(1.2132^n)$ by automatically analyzing millions of special cases. Here we will present a verifier for the computerized part of that analysis and proof the correctness of the same by Kneis, Langer and Rossmanith independently.

We will proof the theoretical foundations of the verifier first and then provide some details on the actual implementation in `C++`.

2 Preliminaries

The proof conducted in [3] uses so called graphlets, which represent small sections of a bigger graph and thus convey a picture of the extended neighborhood around a central vertex.

Definition 1 (Graphlet). *Let $G = (I \cup O, E)$ be a graph and $v \in I$ such that $I = \{v\} \cup I_1 \cup I_2$ with $I_1 = N(v)$, $I_2 = N^2(v)$, $O = N^3(v)$ and $\deg(u) = 1$ for $u \in O$. Moreover, let $\deg(v) \geq deg(u)$ for all $u \in I$. We call (G, v)* graphlet *of radius 2 with* inner vertices *I, consisting of orbits I_1, I_2 and the root vertex, and* outer vertices *O. The set of edges between I and O are called* anonymous edges*, where $a(u) = |\{(u, w) \in E(G) \mid w \in O\}|$ denotes the number of anonymous edges incident to a given vertex $u \in I$.*

Definition 2 (Subgraphlet). *Let* (G, v) *and* (H, r) *be graphlets with inner vertices* I *and* J*.* (H, r) *is subgraphlet of* (G, v) *if there exists a mapping* $f : V(H) \to V(G)$ *which conforms to the following conditions:*

1. $(u, w) \in E(H) \Rightarrow (f(u), f(w)) \in E(G)$
2. $f(r) = v$
3. $f(J_i) = I_i$ *for* $i \in \{1, 2\}$
4. $a(f(u)) \leq a(u)$ *for all* $u \in V(H)$

As the algorithm's performance on graphlets is invariant under isomorphism, we are interested only in one representative for each isomorphism class. Isomorphism for graphlets is a natural extension of graph isomorphism, additionally preserving the root label and membership of vertices to I_1, I_2 or O.

Definition 3 (Graphlet isomorphism). *Two graphlets* (G_1, v_1) *and* (G_2, v_2) *are* isomorphic *iff* (G_1, v_1) *is a subgraphlet of* (G_2, v_2) *and vice versa. The class of all isomorphic graphlets to a given graphlet* G_1 *is written as* $[G_1]$*.*

For the scope of this paper, the definition of graphlets is still too broad — we are only interested in a certain subset of graphlets with limited size, because only these are relevant for the computerized part of the proof. We will call these graphlets **valid graphlets**.

Definition 4 (Valid graphlet). *A graphlet* (G, v) *is called* valid *iff for all* $u \in V(G)$ *the degree fulfills the constraint* $3 \leq deg(u) \leq 4$ *and each vertex in* I_1 *is connected to at least one vertex in* I_2*. Furthermore, the bounds* $1 \leq |I_1| \leq 4$ *and* $1 \leq |I_2| \leq 7$ *must hold. We will call the representative set of all valid graphlets* $\mathcal{S}$*.*

Remark 1. In the following it will be assumed that all the vertices in every graphlet are the numbers from 0 to the number of vertices minus one, where 0 is the root vertex and all the vertices in I_1 are smaller than the vertices I_2. This is also consistent with the implementation.

The aim of this verifier will be to generate $\mathcal{S}$ and compare this set against the certificate given for the proof. First we will prove the correctness of the generating methods of this verifier in Section 4. Afterward we will present some details of the actual implementation and the tools used to verify the certificate in Section 5. In essence, this method and its implementation are meant to be as readable as possible while still being fast enough to generate the enormous number of graphlets needed. The following outlines our approach:

1. Generate all graphlets:
   (a) Generate all *base graphlets* (see Definition 5).
   (b) Add edges to each vertex which does not yet have minimal degree in every possible way.
   (c) Add edges in every possible way to the graphlets generated in Step 1b while no vertex has more than the maximal degree.
2. Perform a full isomorphism test on all graphlets generated in step 1, i.e. leave only unique graphlets in the set.
3. See if the remaining graphlets from Step 2 are all found in the certificate.
4. Calculate all the branching vectors and compare them to the certificate.

Definition 5 (Base graphlet). *A graphlet B is called a* base graphlet *if the following conditions are met:*

1. *For each vertex u in I_1 there is at least one vertex $v \in I_2$ so that $(u, v) \in E(B)$.*
2. *If any edge of the base graphlet is removed either the above condition is violated or some other condition for a graphlet is broken.*
3. *$a(v) = 0$ for all $v \in V(B)$.*

The set of all base graphlet with fixed orbits I_1, I_2 is denoted by $\mathcal{B}_{|I_1|,|I_2|}$

At first it could appear as if only trees could fulfill this property. This not the case, as can be seen in Figure 1 which depicts representatives for all base graphlets with four vertices in orbit one and seven in orbit two.

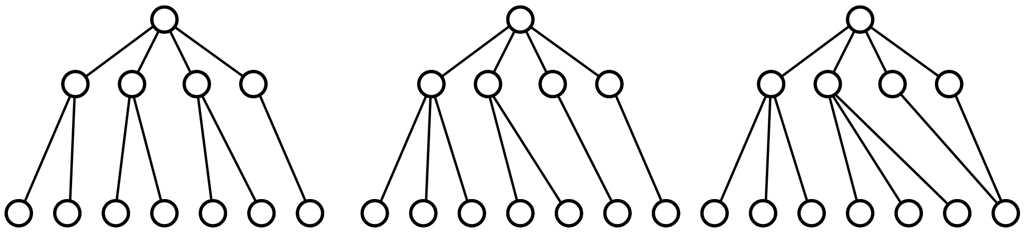

Fig. 1. Base graphlets with four vertices in I_1 and seven in I_2

3 Recursive Graphlet Generation

A common aspect of all algorithms used in the steps outlined in Listing 2 is that they recursively take a graph (some early stages cannot be called graphlets) and output a set of graphs build from that input by adding certain edges. All of them avoid redundant computations by reducing the number of isomorphic graphs (graphlets) which are taken as an input. These similarities suggest that a theoretical generalization might simplify the forthcoming proofs in Section 4, therefore we introduce the following definitions and lemmata.

Definition 6 (Isomorphic Distance). *Let G_1 and G_2 be graphlets with equal sized orbits. If there exists a graph $G \in [G_1]$ which is a subgraph of G_2 then the isomorphic distance is*

$$d(G_1, G_2) = min\{|E(G) - E(G_1)| \ : \ G \in [G_2]\}$$

else

$$d(G_1, G_2) = \infty$$

Lemma 1. *For two graphlets $G_1 \leq G_2$, iff $G_1 \simeq G_2$ then $d(G_1, G_2) = 0$.*

Proof. We will proof this in two steps.

$\Rightarrow$ If $G_1 \simeq G_2$ then there exists a mapping $f(G_2) = G_1$ which means that under that mapping $|E(G_1) - E(f(G_2))| = 0$. Since the distance can not me smaller that zero this has to be the minimum and therefore $d(G_1, G_2) = 0$.

$\Leftarrow$ If $d(G_1, G_2) = 0$ it means that there is a graphlet $G \in [G_2]$ for which $|E(G) - E(G_1)| = 0$. This means that G and G_1 are equal, since they have got by the definition the same amount of vertices and we have shown that the same edges too. This means that $G_1 \simeq G \simeq G_2$. □

Lemma 2. *$d(G_1, G)$ is the same for all $G \in [G_2]$.*

Proof. This follows directly from the fact that $[G] = [G_2]$ for all $G \in [G_2]$. □

Lemma 3. *$d(G, G_2)$ is the same for all $G \in [G_1]$.*

Proof. Say

$$d(G_1, G_2) = min\{|E(G) - E(G_1)| \ : \ G \in [G_2]\} = |E(I) - E(G_1)|$$

for some $I \in [G_2]$. Let G' be an isomorphic graphlet to G_1 and f the mapping for which $f(G_1) = G'$. It is then clear that

$$\begin{aligned} d(G_1, G_2) &= min\{|E(G) - E(G_1)| \ : \ G \in [G_2]\} \\ &= min\{|E(f(G)) - E(f(G_1))| \ : \ G \in [G_2]\} \end{aligned}$$

but since $[G_2] = [f(G_2)]$ this implies

$$\begin{aligned} d(G_1, G_2) &= min\{|E(f(G)) - E(f(G_1))| \ : \ G \in [G_2]\} \\ &= min\{|E(G) - E(G')| \ : \ G \in [G_2]\} = d(G', G_2) \end{aligned}$$

which is what we wanted to proof. □

Definition 7 (Expanding Algorithm). *An* expanding algorithm *$\mathcal{A}$ over a set of graphs $\mathcal{G}$ is a set $\{e(G), s(G)\}$ consisting of an expansion function $e : \mathcal{G} \to 2^{\mathcal{G}}$ and a selection function $s : \mathcal{G} \to \{0, 1\}$ conforming to $H \simeq G \Rightarrow s(H) = s(G)$. The* history *$H_A(G) \subseteq \mathcal{G}$ of $\mathcal{A}$ on a given graph G is recursively defined as*

1. $G \in H_A(G)$
2. $H \in H_A(G) \Rightarrow e(H) \subseteq H_A(G)$

and the output *$O_A(H_A(G))$ is simply defined as*

$$O_A(H_A(G)) = \{H \in H_A(G) \mid s(H) = 1\}$$

Definition 8 (Monotone Expansion Function). *Let $\mathcal{G}$ be a set of graph and let $e : \mathcal{G} \to 2^{\mathcal{G}}$ be a function. We call $e(G)$* monotone expanding *over $\mathcal{G}$ if*

$$\forall H \in e(G) : |E(G)| < |E(H)|$$

holds for all graphs $G \in \mathcal{G}$.

Definition 9 (Monotone Expanding Algorithm). *Let $\mathcal{A} = \{e(G), s(G)\}$ be an algorithm with monotone expansion function $e(G)$ over $\mathcal{G}$ and $G \in \mathcal{G}$ be a set of graphs. If for every $G_i \in H_A(G)$ and $G_t \in O_A(H_{\mathcal{A}}(G))$ with $\infty > d(G, G_t) \geq d(G_i, G_t) > 0$ there exists a $G_{i+1} \in e(G_i)$ with $d(G_i, G_t) > d(G_{i+1}, G_t)$ we call $\mathcal{A}$ a* monotone expanding algorithm *on G.*

Definition 10 (Choice Function). *Let $\mathcal{A} = \{e(G), s(G)\}$ be a monotone expanding algorithm over $\mathcal{G}$ and $c : 2^{\mathcal{G}} \to \mathcal{G} \cup \{\bot\}, c(\mathcal{G}) \in \mathcal{G} \cup \{\bot\}$ be a* choice function*. Then we call $H_{\mathcal{A},c}(G) = \bigcup_{i\in\mathbb{N}} C_{\mathcal{A}i}$ the history of $(\mathcal{A}, c)$ over G with*

$$C_{\mathcal{A}0} = \{G\}$$
$$C_{\mathcal{A}i} = \begin{cases} C_{\mathcal{A}i-1} - c(C_{\mathcal{A}i-1}) + e(c(C_{\mathcal{A}i-1})) \text{ if } c(C_{\mathcal{A}i-1}) \neq \bot \\ \emptyset \text{ if } c(C_{\mathcal{A}i-1}) = \bot \end{cases}$$

where $c(S) = \bot$ must imply that for each subset $S' \subseteq S \Rightarrow c(S') = \bot$ holds. We call c a valid *choice function of $\mathcal{A}$ on G if $O_{\mathcal{A}}(H_{\mathcal{A},c}(G)) \simeq O_{\mathcal{A}}(H_{\mathcal{A}}(G))$.*

A choice function therefore introduces a certain order in which graphs are expanded. If $c(S) = \bot \Leftrightarrow S = \emptyset$ holds, we can immediately see that the resulting output (and even the history) of $\mathcal{A}, c$ and $\mathcal{A}$ are equal. But the possibility to not expand *every* graph is exactly what we will need later on, thus we need a criteria to decide which graph we must expand in order to obtain a valid choice function.

Definition 11 (Economical Choice Function). *Let c be a choice function of a monotone expanding algorithm $\mathcal{A} = \{e(G), s(G)\}$ on a graph G. Consider two graphs $G_i \in H_{\mathcal{A}}(G)$ and $G_t \in O_{\mathcal{A}}(H_{\mathcal{A}}(G)), G_t \neq G$ with $\infty > d(G_i, G_t) > 0$. If c chooses a G_j for some k, i.e. $G_j = c(C_{\mathcal{A}k})$, and $d(G_j, G_t) < d(G_i, G_t)$ then c is called* economical choice function *of $\mathcal{A}$ on G.*

Lemma 4. *There is always at least one choice set $C_{\mathcal{A}k}$ which contains such a G_j.*

Proof. Assume $G_i = c(A_{\mathcal{A}l})$ — as $d(G, G_t) > 0$, such a G_i must exist (remember that $C_{\mathcal{A}0} = \{G\}$). In Definition 9 it is demanded that $e(G_i)$ contains at least one graph G'_j for which the $d(G'_j, G_t) < d(G_i, G_t)$ holds, therefore G_j can be chosen from $G_{\mathcal{A}(l+1)}$.

Lemma 5 (Validity of Economical Choice Functions). *Every economical choice function c of monotone expanding algorithm $\mathcal{A}$ on a graph G is valid.*

Proof. We need to show that for each $G_t \in O_{\mathcal{A}}(H_{\mathcal{A}}(G))$ there exists a $G'_t \in O_{\mathcal{A}}(H_{\mathcal{A},c}(G))$ with $G'_t \simeq G_t$. Consider such a G_t with $G_t \not\simeq G$. Because $d(G, G_t) > 0$ — otherwise a monotone expanding algorithm could not generate G_t from G — there exists at least one graph G_0 with $d(G_0, G_t) > 0$. Then, by the definition of c, G_1 with $d(G_1, G_t) < d(G_0, G_t)$ is chosen at some point. If $d(G_1, G_t) = 0$ we are done because that implies $G_1 \simeq G_t$. Otherwise, $d(G_0, G_t) > d(G_1, G_t) > 0$ holds and can apply the definition of c again, thus constructing a chain of graphs $G_0, G_1, \ldots, G_i$ with $d(G_0, G_t) > d(G_1, G_t) > \cdots > d(G_i, G_t)$ (Lemma 4 states that this is always possible). As $d(G, G_t)$ is finite, at some point this chain must end in a graph G_j with $d(G_j, G_t) = 0$ and therefore $G_j \simeq G_t$. □

We therefore do not have to follow every path of expansion to a certain output graph G_t, like implicated in Definition 9, but only have to make sure that at *some point* a graph that reduces the distance to G_t further is chosen.

Lemma 6 (Reduced Choices). *A choice function c which only chooses H' with $H' \simeq H \in H_{\mathcal{A}}(G)$ and $c(C_{\mathcal{A}k}) = H' \Rightarrow c(C_{\mathcal{A}j}) \not\simeq H'$ for all $j < k$ is economical.*

Proof. Consider $G_t \in O_{\mathcal{A}}(H_{\mathcal{A}}(G))$ and $G_i \in H_{\mathcal{A}}(G)$ with $d(G_i, G_t) > 0$. As $H_{\mathcal{A}}$ contains at least one graph G_j with $d(G_j, G_t) < d(G_i, G_t)$, c can choose this graph or an isomorphic version of it — d is invariant under isomorphism, so each choice would satisfy Definition 11. □

Corollary 1. *A choice function c which only chooses H' with $H' \simeq H \in H_{\mathcal{A}}(G)$ and $c(C_{\mathcal{A}k}) = H' \Rightarrow \nexists H'' \in R_k$ and $H'' \simeq H'$ with $R_k \subset \bigcup_{i<k} c(C_{\mathcal{A}i})$ is economical.*

Proof. This follows directly as this choice function chooses a superset of graphs w.r.t. a choice function operating like proposed in Lemma 6. □

4 Generation of Graphlets

As was already stated before, the process of generating the graphlets is subdivided in three steps (see Step 1, Listing 2), the last of which outputs all valid graphlets S. Each of those steps takes the output of the preceding one, thus the correctness of the whole process follows from the correctness of each single step.

Lemma 7. *Let S be a* seed graph *with two designated sets of vertices I'_1, I'_2 of sizes $i_1 = |I'_1|, i_2 = |I'_2|$ and a root vertex v, where v is connected to each vertices of I_1 but not other edges exists. Then* `generateBaseGraphs(S)` *generates all base graphlets with $I_1 = I'_1$ and $I_2 = I'_2$.*

Proof. First we will prove that the function `generateBaseGraphs(S)`, without the isomorphism test in lines 8-12, given in Listing 1.1 is equivalent to a monotone expanding algorithm $\mathcal{A}$. Afterward we will prove that its output $O_{\mathcal{A}}(H_{\mathcal{A}}(S))$ contains indeed all base graphlets of a fixed size, that is $O_{\mathcal{A}}(H_{\mathcal{A}}(S)) \simeq \mathcal{B}_{i_1,i_2}$. Finally we will prove the isomorphism test in lines 8-12 to be equivalent to an economic choice function on S.

Consider the following expanding algorithm $\mathcal{A}$:
e(G): If there exists a subset $V_1 \subseteq I_1$ of vertices which are not connected to I_2, then

$$e(G) = \{G' \mid G' = G \cup (v, w), v \in V_1, w \in I_2\}$$

otherwise, if there exists a subset $V_2 \subseteq I_2$ of vertices which are not connected to I_1, then

$$e(G) = \{G' \mid G' = G \cup (v, w), v \in I_1, w \in V_2\}$$

otherwise $e(G) = \emptyset$.

s(G): $s(G) = 1 \Leftrightarrow G$ is a base graphlet.

First we need to assure that $H_{\mathcal{A}}(S)$ contains all base graphlets. This can be verified easily: given a base graphlet B, we can generate it from S by successively adding all necessary edges via the expansion.

We want to prove that $\mathcal{A}$ is a monotone expanding algorithm. It is easy to see that $\mathcal{A}$ only adds edges to a graphlet, thereby e is a monotone expanding function, so what is left to show is that for every base graphlet $B \in O_{\mathcal{A}}(H_{\mathcal{A}}(S))$

and each graph $G \in H_{\mathcal{A}}(S)$ with $d(G, B) > 0$ there exists a $G' \in e(G)$ with $d(G', B) < d(G, B)$. Assume that in I_1 of G there exists a vertex v which is not connected to I_2. As all possible edges that can connect v to I_2 are added to G, one of those edges will decrease the distance to B, as all vertices in B *are* connected via some edge to I_2. The same argumentation holds for the case that all vertices in v already have a neighbor in I_2 and edges are added to vertices of I_2 instead. Therefore, $\mathcal{A}$ is a monotone expanding algorithm.

To see that the pseudo-code provided in Listing 1.1 (without lines 8-12) is equivalent to $\mathcal{A}$, let us think of the recursion employed there as a choice function c that simply acts upon a stack of graphs (initially empty) and the loop in lines 26-33 and lines 47-53 simply fill that stack with the graphs generated by $e(G)$ (the order is not important).

Consider the isomorphism check in lines 8-12 next, which prevents that any intermediate graph is expanded twice. Would this prevent any graph to be expanded twice, the corresponding choice function c' would, according to Lemma 6, be economical. As space is limited, not all graphs can be saved, thus c' operates in the sense of Corollary 1 — either way, c' is economical and thus $O_{\mathcal{A}}(H_{\mathcal{A}}, c) \simeq O_{\mathcal{A}}(H_{\mathcal{A}}(S)) \simeq \mathcal{B}_{i_1,i_2}$. □

Listing 1.1. Generate all base graphs of a given size

```
1  generateBaseGraphs(){
2      Graph seed;
3      connect each orbit1 vertex with the root vertex;
4      generateBaseGraphs( seed );
5  }
6
7  generateBaseGraphs( Graph G ){
8      if ( G not in graph set ) {
9          add G to graph set;
10     } else {
11         return;
12     }
13
14     // Search for an invalid orbit1 vertex
15     int v = 1;
16     while v < |orbit1|+1 {
17         if v has no edge to any vertex in orbit2
18             break;
19         v++;
20     }
21
22     if ( v < |orbit1|+1 ) {
23         // v is an unconnected vertex in orbit1.
24         // Consider all possibilities to connect it with vertices from orbit2.
25         int w = |orbit1|+1;
26         while w < |orbit1|+|orbit2|+1 {
27             if ( (v,w) is edge in G )
28                 continue;
29
30             if ( degree(w) < maxDegree && degree(v) < maxDegree )
31                 generateBaseGraphs( G + (v,w) );
32             w++;
33         }
```

```
    } else {
        // No invalid vertex was found in orbit1.
        // Search for an invalid orbit1 vertex
        int v = |orbit1|+1;
        while v < |orbit1|+|orbit2|+1 {
            if v has no edge to any vertex in orbit1
                break;
            v++;
        }

        if ( v < |orbit1|+|orbit2|+1 ) {
            // v is an unconnected vertex in orbit2.
            int w = 1;
            while w < |orbit1|+1 {
                if ( (v,w) is edge in G )
                    continue;

                if ( degree(w) < maxDegree && degree(v) < maxDegree )
                    generateBaseGraphs( G + (v,w) );
            }
        } else {
            if ( isBaseGraph( G ) ) {
                // The graph is a valid base graph
                add G to results;
            }
        }
    }
}
```

Definition 12 (Minimal Graphlet). *A minimal graphlet is a valid graphlet (see Definition 4), where no edge can be removed without the graphlet becoming invalid.*

Notice that in a minimal graphlet some vertices may have a degree greater than the minimal degree (see Figure 2)

Lemma 8. *Every valid graphlet has at least one base graphlet as a subgraphlet.*

Proof. Let G be a graphlet. First we can remove all anonymous edges. If then G is not yet a base graphlet there exists some edge e which can be removed without either changing I_1 and I_2 or leaving a vertex without a neighbor in the respective other orbit. Successively removing such edges e will at some point

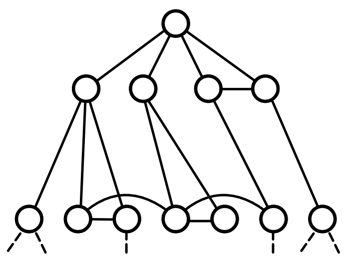

Fig. 2. A minimal graphlet with four vertices in I_1 and seven in I_2 . Dashed lines represent anonymous edges.

result in a graph which cannot be reduced further — it therefore fulfills all criteria (cf. Definition 5) of a base graphlet. As we only removed edges — anonymous or not — this resulting graphlet clearly is a subgraphlet of G. □

Lemma 9. *Every valid graphlet has at least one minimal graphlet as a subgraphlet.*

Proof. If a graphlet G is not minimal it means that there is an edge we could remove so that the graphlet remains valid. We can remove all such edges iteratively and at some point no edge will be removable anymore. This graphlet must by definition be a minimal graphlet and a subgraphlet of G. □

Before we begin with the actual proof of the correctness let us gain a little insight into the logic of the `complete(Graph G)` algorithm. The actual work in the algorithm is done by the functions `complete(Graph G, minDegreeFlags, maxDegreeFlags)` and `completeNode(Node v, Graph G, prohibited, numberAnonymousEdges, numberConections)` which call each other recursively. The logic behind that split is that first `complete(Graph G, minDegreeFlags, maxDegreeFlags)` chooses a vertex whose degree is smaller than the minimal degree. For the vertex it has chosen it calculates how many edges are missing till achieving minimal degree. It then enumerates all possible partitions between anonymous edges and non-anonymous edges on that number and calls `completeNode(Node v, Graph G, prohibited, numberAnonymousEdges, numberConections)` on each one of them. This function will then enumerate every possible way to map this partition on edges on the graphlet `G` and call `complete(Graph G, minDegreeFlags, maxDegreeFlags)` on the new generated graphlets (Unless an isomorphic graphlet was already expanded). This will go on like this until an expansion generates a minimal graphlet; a minimal graphlet can not be expanded so that another minimal graphlet is generated, which means we can stop the recursion.

Listing 1.2. Give every vertex at least minimal degree

```
void complete( Graph G ) {

    nodesWithMaxDegree = All nodes with minimal degree;
    nodesWithMinDegree = All nodes with maximal degree;

    complete( G, nodesWithMinDegree, nodesWithMaxDegree );
}

void complete( Graph G, minDegreeFlags, maxDegreeFlags ) {

    // Search a vertex which has a degree smaller
    // than the minimal degree.

    Node v = First node in the first orbit;
    while( v < totalVertices ) {
        v++;

        if (degree(v) > minDegree){
            continue;
        }

        // Denotes how many edges are needed until the vertex v
```

```
        // reaches minimal degree.
        int degreeLeft = minDegree − degree(v);

        prohibited = {u : (v, u) ∈ E(G)} + self + maxDegreeFlags;

        if ( v in orbit1 ) {
            // Vertex in orbit one cannot have anonymous edges
            completeNode( v, G, prohibited, minDegreeFlags, maxDegreeFlags, degreeLeft, 0
                );
        } else {
            // Enumerates the different possibilities for
            // a vertex to have anonymous edges.
            for (int i = 0; i <= degreeLeft; ++i) {
                completeNode( v, G, prohibited, minDegreeFlags, maxDegreeFlags, (
                     degreeLeft − i), i);
            }
        }
    }

    The Graph is complete, write it to file ;
}

completeNode(Node v, Graph G, prohibited, minDegreeFlags, maxDegreeFlags,
     numberAnonymousEdges, numberConections) {

    minDegreeFlags += {v};

    if (numberConections == 0) { // Set only anonymous edges.
        E = G;
        E.setAnonymousEdges(v, numberAnonymousEdges);
        if E is not known
            complete(E);
    } else {

        foreach valid set S of nodes the size of numberConnections {
            newMinDegreeFlags = minDegreeFlags;
            newMaxDegreeFlags = maxDegreeFlags;

            if the set has nodes that are in the prohibited set
                continue;

            E = G;
            E.setAnonymousEdges(v, numberAnonymousEdges);
            for each edge (v.i) in S {
                E.setEdge(v,i);
                if (degree(i) >= minDegree {
                    newMinDegreeFlags += {i};
                    if(degree(i) == maxDegree)
                        newMaxDegreeFlags += {i};
                }
            }

            if E can still be minimal and E ist not known
                complete (E);
        }
    }
}
```

First we will proof some lemmas for which we will have a careful look at the pseudo code that we will later use as small part of a more formal proof of the correctness of the algorithm.

Lemma 10. *The sets* `minDegreeFlags` *and* `maxDegreeFlags` *are always consistent with the graphlet* `G` *on any call of* `complete(Graph G, minDegreeFlags, maxDegreeFlags)`.

Proof. We will proof this by induction:

Base Case The first time the function is called it is called by `complete(Graph G)` which calculates this sets directly from the graphlet `G`.

Induction Step In `complete(Graph G, minDegreeFlags, maxDegreeFlags)` the flag sets are not changed. When `completeNode(Node v, Graph G, prohibited, minDegreeFlags, maxDegreeFlags, numberAnonymousEdges, numberConections)` is called the first thing we do is mark the vertex `v` as having minimalDegree. It can not have maximal degree since we are assuming that the maximal degree is greater than the minimal. In case the node does not get minimal degree, i.e. there is no subset with is not prohibited in the loop at line 54, then no recursive call is made, which makes what happens to the flags irrelevant. Before we make any change to the graphlet we copy the sets for the new graphlet on lines 55 and 56. If a non-prohibited subset is found the if-clauses at lines 65 and 67 will look what happens with other node on the edge. This makes the new sets in the variables `newMinDegreeFlags` and `newMaxDegreeFlags` correct on the recursive call to `complete(Graph G, minDegreeFlags, maxDegreeFlags)`.

□

Lemma 11. *When a vertex* `v` *is selected in* `complete(Graph G, minDegreeFlags, maxDegreeFlags)` *it achieves minimal degree in every possible way.*

Proof. If we have already selected a vertex `v` we are at line 24, where the remaining number of edges *degreeLeft* is calculated. Let $|M| = degreeLeft$ be a set of valid edges which are to be added to v. If v is a vertex in the first orbit then no anonymous edges can be in M. This makes the call on line 30 correct. Otherwise it does not matter what amount of the edges in M are anonymous, the right amount will be selected at some point by the for-loop on line 34, since no valid set M can have a node which is in the prohibited set. □

Lemma 12. *If the function* `complete(Graph G)` *given in Listing 1.2 is called upon every base graphlet every minimal graphlet is generated.*

Proof. To prove the correctness of the algorithm, we will first proof that the algorithm is a monotone expanding algorithm, which generates every minimal graphlet G where $d(B, G) < \infty$ when called on a base graphlet B. We will demonstrate that there are always graphlets generated to which the distance to the desired graphlet is always smaller than before. This can be repeated until the distance becomes zero, which is equivalent to the graphlets being isomorphic.

First some comment on the formalization: The function that will be called on every base graphlet will be `complete(Graph G)` but this function is only called once for every base graphlet. This function will then call `complete(Graph`

`G, minDegreeFlags, maxDegreeFlags)`, which will either stop the recursion or call `completeNode ( Node v, Graph G, prohibited, numberAnonymousEdges, numberConections)` which itself will call `complete(Graph G, minDegreeFlags, maxDegreeFlags)` again. Since these two function call each other recursively we can consider them as one single function, replacing calls to `completeNode(Node v,Graph G, prohibited, numberAnonymousEdges, numberConections)` by "inlining" that function. We will call this simply the completion algorithm.

e(G): We want to show that part of the functionality of the completion algorithm can be regarded as a monotone expanding algorithm. $e(G)$ will then be the whole function except line 40 and the if-clauses where the recursion is stopped if the graphlet is already known or an edge has been added which makes it impossible for the graph to be minimal.

1. It is clear that the only changes the function can make to any graphlet is adding edges; either it does not select any vertex in the loop between line 15 and 20, which means that on the graphlet G upon which the function was called $e(G) = \emptyset$ or it does select a vertex. In the second case it will add some edge to the graphlet either on line 49, 62 or 64. This means that $e(G)$ is a monotone expansion function.
2. For the algorithm to be a monotone expanding algorithm $e(G)$ has to work in a specific way: For every graphlet G_s and minimal graphlet G with $0 < d(G_s, G) < \infty$ there has to be a graphlet $G' \in e(G_s)$ for which $d(G', G) < d(G_s, G)$.
   From the definition of the distance function we know that for $d(G_s, G) = k < \infty$ a mapping f has to exist where $|E(G_s) - E(f(G))| = k$, where G_s is a subgraphlet of $f(G)$. Since we are assuming that $0 < d(G_s, G)$ we know G_s can not be a minimal graphlet, since any minimal graphlet has to have a distance to G of either 0 or ∞. Let then v be the vertex that is selected at the beginning of `complete(Graph G, minDegreeFlags, maxDegreeFlags)`. Now the algorithm will add edges to v in every possible way so that it achieves minimal degree, this will cause at some point the distance between $d(G', f(G))$ and $d(G_s, f(G))$ to decrease. From this follows that $d(G', G) = d(G', f(G)) < d(G_s, f(G)) = d(G_s, G)$ since $f(G) \simeq G$ which is what we wanted to show.

s(G): This function has to select at least all the minimal graphlets. On every call of `complete(Graph G, minDegreeFlags, maxDegreeFlafs)` line 40 is executed. There the minimality of each vertex is tested, which is a necessary condition for minimal graphlets, and if every vertex has at least minimal degree the graphlet is written to a file, which is the same as the graphlet being part of the output. Notice under this definition $s(G)$ is called on every graphlet in the history, which make $s(G)$ a valid selection function in $\mathcal{A}$.

choice function: The choice function consist of the lines 50 and 72. This selects for every graphlet G which is generated at some point on the algorithm without the choice function (the same functions only with this if-clauses removed) at least one graphlet $G' \in [G]$. It also throws away every graphlet the function generates which has an edge that can be removed, since that means that whatever graphlet is generated by expanding such a graphlet it can not be minimal. This in conjunction with Lemma 6 suffices to show that this choice function does not falsify the result.

Now we see that the algorithm is a monotone expanding algorithm and that $e(G)$ will expand some graphlet in the history to decrease the distance to any minimal graphlet G on any graphlet whose distance to G is finite. But this follows directly from Lemma 8, since from this lemma we can deduce that for a base graphlet B the distance $d(B, G) < \infty$. Since the distance is finite and always decreases at some point is has to become zero. This means that an isomorphic version of G is generated in any case. □

Let us move on to `addPossibleEdges(Graph G, Node v)`. This function is pretty straightforward compared to the previous one; it is executed on every minimal graphlet with the first vertex in the first orbit as the second argument to generate all valid graphlets. The algorithm either adds an edge to `v` — if possible — and call itself recursively on the vertex `v` again, or do nothing with `v` and call itself recursively on the next vertex (in the order of the labeling). By doing that it adds every subset of the set of all edges that can be added to a minimal graphlet.

Listing 1.3. Add all possible edges to the graphs generated in the second step

```
addPossibleEdges(Graph G, Node v) {
    if (v > numberVertices) {
        Write G to file;
        return;
    }

    if degree(v) < maximalDegree {
        foreach pair (v,w) where w > v {
            if (G.adjacent(v,w))
                continue;

            if degree(w) < maximalDegree {
                E = G;
                E.setEdge(v,w);

                if E is not known
                    addPossibleEdges(E, v);
            }
        }


        if v in second orbit {
            E = G;
            E.addOneAnonymousEdgeToNode(v);

            if E is not known
                addPossibleEdges(E, v);
        }
    }

    addPossibleEdges(G, v + 1);
}
```

Lemma 13. *Every valid graphlet is generated by* `addPossibleEdges(...)` *at some point, if it is called on every minimal graphlet with* `Node v` $= 1$ *as the second argument.*

Proof. It is obvious that if it was not for the isomorphism test on lines 16 and 26 then every set of edges that can be added to a minimal graphlet would be added and so that every valid graphlet would be generated.

We still have to show that the isomorphism test does not impede the creation of a graphlet G for every valid graphlet G_V with $G \in [G_V]$. The correctness of this assertion follows from the depth-first recursion the algorithm employs.

Let G be a graphlet upon which the function would have been called with i as a second argument — assuming it is not, because an isomorphic version was already encountered before — and let (v, w) with $w > v \geq i$ be any edge we could have added to G. Let $G' \simeq G$ be the isomorphic version upon which the function was called and f the mapping for which $G' = f(G)$. The edge (v, w) becomes $(f(v), f(w)) = (u, u')$ where $u' > u$. Furthermore, let i' be the second argument on the call upon G'.

If $i' \leq u$ for all edges (v, w) then, by using the recursive call on line 31 (which is always executed), we can reach a moment where the second argument is u and the edge is added.

If $i' > u$ for some edge (v, w) then there is a subgraphlet G'' of G' upon which the function was called (else this call would not exist) for which the the second argument equaled u which means that the edge is added. There is a path that adds all the edges missing in G'' that are in G'.

This means that in every step we find a graphlet for which a path in the code exists that generates all expansion of G. Since a path is only interrupted when an isomorphic version is found this path must either complete or be interrupted because of one of the reasons stated before or because a graphlet on the path can only be expanded to graphlets that already exist on this path. ◻

From the previously proved lemmas the following central theorem follows:

Theorem 1. *Every valid graphlet is generated by first generating the base graphlets, then calling* `complete Graph(Graph G)` *on every base graphlet and lastly calling* `addPossibleEdges(Graph G, 1)` *on every graphlet generated by* `completeGraph(Graph G)`.

5 Implementation

In this section we will outline our implementation of the above algorithm written in C++ as well as the tools we developed to compare the output of our verifier with the certificate. We also introduce a data structure which efficiently spots isomorphic graphlets[1] and thus contributes significantly to the good performance of the verification process.

[1] To be precise: graphlets and subgraphs of graphlets, as the `generateBaseGraphs()` works on not yet complete graphlets

5.1 Subset Representation and Enumeration

The algorithm `addPossibleEdges(Graph G, Node v)` uses sets of vertices for certain optimization, which we model by using unsigned integers, assuming that the program will be run on an architecture with at least 32 bit (which should be commonly enough). Each position in the binary representation of the integer is seen as a flag which denotes whether a specific vertex is part of the set or not.

If we regard the bit-strings as being ordered from right to left, the first position will be the root node, then come all the positions for the vertices in the first orbit and then all the ones for the vertices in second orbit.

This way of representing set allows to speed up some operation, for example testing whether a certain set is a subset of another or whether two sets have at least one element in common, both of which become simple integer operations.

To enumerate subsets we use a bithack known as Gosper's Hack, which is described int the 0. fascicle of the 4. book of The Art of Computer Programming [6].

5.2 Data Structure: Graphlet Trie

Listings 1.1, 1.2 and 1.3 require a way to determine whether a given graphlet was already encountered during the recursion of the algorithm. The following abstract data structure concretises this premise:

Definition 13 (Graphlet set). *A graphlet set is any data structure which provides the operations* insert *and* contains *for graphlets. These operations must conform to the following contracts:*

insert	*Inserts a given graphlet into the set.*
contains	*Returns true if the graph or an isomorphic version was inserted before, otherwise false.*

As space is quite limited while the amount of graphlets grows exponentially w.r.t the number of vertices, the following less restrictive definition is introduced (this corresponds to Corollary 1)

Definition 14 (Limited graphlet set). *A limited graphlet set is any data structure which provides the operations insert and contains conforming to the following contracts:*

insert	*Inserts a given graphlet into the set if there is enough space.*
contains	*Returns true if the graph or an isomorphic version is contained, otherwise false.*

Furthermore it must be deducible whether the limited graphlet set can save further graphs or not, i.e. by initializing the structure with a certain amount of memory or the maximum number of graphlets that should be stored.

Under this definition, even a trivial data structure which never saves graphlets and always returns false when asked whether a certain graphlet is contained is a valid *limited graphlet set* and our algorithm would work correctly, albeit slowly, with it. But of course the intent is to reduce the amount of redundant computations, so a good *limited graphlet set* will try to save as many graphlets as possible and to recognize a lot of isomorphisms.

While a a simple set data structure would work reasonable well to store graphlets, the amount of comparisons that would have to be made for a single isomorphism check would outweigh the gain by far. Therefore, we introduced the graphlet trie which pre-sorts graphlets into buckets by generating a *characteristic sequence* for each of them:

Definition 15 (Graphlet characteristic sequence). *A characteristic sequence $I(G)$ of a given graphlet G is any sequence of integers that conforms to*

$$I(G) = I(H) \text{ for all } H \simeq G$$

A simple example for a characteristic sequence is a sorted list containing the degrees of each vertex or – trivially – the sequence of length one containing always a zero. If any method to generate such a characteristic sequence is supplied, checking for isomorphic versions of a graph must only be conducted on the bucket identified by that sequence as the equality of the characteristic sequences is, by definition, a necessary precondition.

Note that even if the characteristic sequences of two isomorphic graphs would differ or if any other kind of (logical) error would occur, the above is still a valid *limited graphlet set*; it would not dismiss any graphlet that was not encountered yet. Furthermore, if used as a basis for isomorphism tests, a mistake in the implementation of the characteristic sequence can only cause a false negative, never a false positive.

Our method of generating characteristic sequences is to calculate the characteristic polynomial of the adjacency matrix[2] and use the coefficients sorted by the order of their respective terms. The calculation itself is done by an iterative variant of the Berkowitz-algorithm[1] which calculates the characteristic polynomial without divisions and therefore is not susceptible to numerical errors.

We want to stress that the isomorphism check, which enumerates all feasible mappings (e.g. it does not try to map a vertex from I_1 to a vertex of I_2) for two graphlets, is robust in the sense that the mapping is applied to the graphlet and then tested for equality against the other — again this only makes false negatives possible, not false positives[3].

5.3 Full Test for Isomorphism

In order to compare the generated graphs we need to remove all non-unique graphs from our set of graphlets (see Section 5.4 for an explanation why). This is accomplished by comparing each graph from our set against all others while considering all possible isomorphic versions, thus filtering out all isomorphic variants. We again employ a limited graphlet set, but with two additional operations:

[2] We managed to gain a better discrimination of graphlets by adding a little more information to the matrix (and thus to the characteristic sequence): for each vertex v on the second orbit, $1 + a(v)$ is written on the respective diagonal entry of the matrix, thus orbit one and orbit two vertices become more 'distinguishable' in the resulting sequence.

[3] We trust that our equality test is correct.

Definition 16 (Limited graphlet checking set). *A limited graphlet checking set is a limited graphlet set which provides two additional operations flag and allFlagged:*

flag	*Marks a graphlet contained in this set*
allFlagged	*Returns whether all graphlets inside this set have been marked*

Furthermore the limited graphlet checking set must assure that if enough memory is provided every graph inserted will be stored and that the memory limit can be controlled in some manner.

To avoid missing graphs due to the limited number of graphs saved inside the *limited graphlet checking set* we have to filter iteratively: each steps loads a new chunk of graphs – small enough to fit into the *limited graphlet checking set* – from the total set and compares it to the already reduced parts of the set.

Listing 1.4. Graphlet set reduction

```
function filterGraphs( S, maxGraphsInMemory ) {

    R = {}; // Empty set

    while ( S is not empty ) {
        I = takeChunk(S, maxGraphsInMemory);
        S = S − {Graphs isomorphic to graphs in I};
        R += I;
    }

    return R;
}
```

We use the graphlet trie (c.f. Section 5.2) – which supports the above operations – for that purpose: a set of valid graphlets from the file F_0 is loaded into the trie which, since the trie will not exceed its capacity and thus not dismiss graphlets[4], are pairwise non-isomorphic. These graphs are written to the resulting file R, afterward all remaining graphs from F_0 are compared against the graphlets contained in the trie — if an isomorphic version is found, the graphlet is dismissed, otherwise it is written to a new file F_1. The next iteration loads as many graphlets as possible from F_1 into the trie and repeats the process, until no graphlets are left.

5.4 Comparison of Graphlets Sets

After obtaining a reduced graphlet set the comparison against another graphlet set can be made. Again a *limited graphlet checking set* is employed to spot isomorphic versions of each graphlet. The sets are checked iteratively by loading appropriate sized chunks from the reduced set and comparing it against all not yet encountered graphlets from the other set, see Listing 1.5:

Listing 1.5. comparing two sets

```
function boolean compareGraphs( S, T, maxGraphsInMemory ) {
```

[4] The implementation is a little more robust: the graphlet trie is initialized with endless capacity and a simple counter keeps track of the amount of graphlets inserted.

```
    while ( S is not empty ) {
        I = takeChunk(S, maxGraphsInMemory);
        S = S − I;
        Mark graphs in I which are found in T;
        T = T − {Graphs isomorphic to graphs in I};
        if (not all graphs in I marked)
            return false; // T misses graphs
    }

    if (T is not empty)
        return false; // S misses graphs

    // S equals T
    return true;
}
```

A set of valid graphlets from the first file is loaded into the trie which, since it was reduced earlier (see Section 5.3) all elements are pairwise non-isomorphic. Afterward all valid graphlets from the other file are compared against the contents of the trie, eliminating each graphlet to which a corresponding isomorphic version can be found, while the remaining graphs are saved to a file. This process is then repeated; again a chunk of graphs is loaded from the first file and compare against the, now reduced, second file. At some point, no graphs are left to be checked and the number of missing graphs in each file is reported.

5.5 Branching Vector Comparison

The final step in the process is the verification of the certificate's branching vectors. Alongside the file containing representatives of S the certificate provides a file with branching vectors of those graphs, ordered so that the i-th branching vector in the one file belongs to the i-th graph in the other. Obviously, if the verifier first testified that the graphlet file was complete and then does not find any faulty calculated branching vector, the upper bound calculated from these branching vectors is correct.

6 Conclusion

We have provided an in-depth overview of the algorithms employed to verify the results by Kneis, Langer and Rossmanith[3] and a proof for its correctness. The source code available at [5] can be examined to verify that those algorithms are correctly implemented. As the certificate has been successfully verified, the upper bound of $O^*(1.2132^n)$ for `Maximum Independent Set` seems to hold.

References

1. S. J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18:147–150, 1984.
2. F. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: A simple $O(2^{0.288n})$ independent set algorithm. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 18–25, 2006.
3. P. Rossmanith J. Kneis, A. Langer. A fine-grained analysis of a simple independent set algorithm, 2009. submitted for publication.

4. T. Jian. An $O(2^{0.304n})$ algorithm for solving Maximum Independent Set problem. *IEEE Transactions on Computers*, 35(9):847–851, 1986.
5. J. Kneis, A. Langer, and P. Rossmanith. Independent set proof homepage, 2009. `http://www.tcs.rwth-aachen.de/independentset/`.
6. D. E. Knuth. Introduction to combinatorial searching, 2008. The Art of Computer Programming, Pre-Fascicle 0.
7. J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.
8. R. E. Tarjan and A. E. Trojanowski. Finding a Maximum Independent Set. *SIAM Journal on Computing*, 6(3):537–550, 1977.