RWTH Aachen

Department of Computer Science
Technical Report

Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem

Michael Nett

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2009-9

RWTH Aachen · Department of Computer Science · April 2009

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/

Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem

Michael Nett

Dept. of Computer Science
RWTH Aachen University, Germany
michael.nett@rwth-aachen.de

Abstract. Kneis, Langer, and Rossmanith [3] proposed an algorithm that solves the maximum independent set problem for graphs with n vertices in $\mathcal{O}^*(1.2132^n)$. This bound is obtained by precisely analyzing all cases that the algorithm may encounter during execution. Since the number of cases exceeds several millions, a computer aided proof is used to generate and evaluate all cases. In this paper, we present a program that fullfills this task and give a detailed description of the principles underlying our method. Moreover, we prove that the set of generated cases includes all relevant cases.

1 Motivation

The Maximum Independent Set problem (Mis) is well known to be NP-hard. Over the past years, several exact algorithms were developed for this problem. Tarjan and Trojanowski [7] presented a method to solve it in time $O^*(1.261^n)$. This was improved by Jian [4] to $O^*(1.235)$ and by Robson [6] to $O^*(1.228)^n)$. In 2006, Fomin, Grandoni and Kratsch [2] devised a new algorithm with a runtime bounded by $O^*(1.2201^n)$.

Recently, Kneis, Langer, and Rossmanith [3] developed an intuitive algorithm that solves Mis in time $\mathcal{O}^*(1.2132^n)$. To prove this new runtime bound, however, a computer aided case distinction was applied. The number of these cases, however, is extremely large and hence demands for an efficient generation method are justified. In this paper we present an implementation of this proof and give a detailed documentation.

Throughout this paper we will try to convey an intuitive understanding of our method and subsequently analyze all involved steps in detail. Finally we will give a formal proof that our method generates the cases relevant for [3].

2 Definitions

Since this report is intended to complement the proof in [3], we will only shortly repeat the relevant definitions here.

Definition 1 ([3]). *Let $H = (V_I \cup V_O, E)$ be graph, such that $V_I \cap V_O = \emptyset$, and let $v \in V_I$ such that $V_I = N^i[v]$, $V_O = N^{i+1}(v)$ and $\deg(u) = 1$ for $u \in V_O$. Moreover, let $\deg(v) \geq deg(u)$ for all $u \in V_I \cup V_O$. We call (H, v)* graphlet *of radius i. We call V_I the* inner vertices *of (H, v) and the set of edges between V_I and V_O the* anonymous edges.

Definition 2. *Let (G, v) be a graphlet. The k-th* orbit *O_k is defined by $O_k = \{u \in V(G) \mid d(u, v) = k\}$ where $d(\cdot, \cdot)$ is the distance.*

Definition 3. *Let (G, v) be a graphlet. (G, v) has* extent *$n \in \mathbb{N}$ if and only if $O_n \neq \emptyset$ and $O_{n+1} = \emptyset$.*

Definition 4. *Let (G, v) and (G', v') be graphlets with vertex sets V_I, V_O and V'_I, V'_O respectively. A bijective mapping $\pi : V(G) \to V(G')$ is a* graphlet isomorphism *if and only if π is an isomorphism w.r.t. G and G' and additionally $\pi(v) = v'$, $\pi(V_I) = V'_I$ and $\pi(V_O) = V'_O$. If such a mapping exists, we write $(G, v) \cong (G', v')$.*

Terms surrounded by < and > refer to command line parameters used when invoking scripts or programs; identifiers surrounded by [and] refer to program names.

3 Generation

Throughout this section we will give a rough overview of our graphlet generation method. Afterwards, we will investigate the steps occurring in the generation algorithm in more detail.

In the first subsection we will specify some properties of the relevant cases. We exploit these properties in order to generate the relevant cases more efficiently. In the subsequent subsections we will discuss the relevant parameters used to setup the generation process.

3.1 Relevant cases

By the proof given in [3] we know that all cases the algorithm considers for branching are fully reduced. Moreover, Theorem 3 from this paper allows us to restrict the relevant cases to graphlets with the following properties:

- $d(u) \geq 3$ holds for all $u \in V(G)$, since vertices of degree smaller or equal 2 are removed by the domination- and folding-rule.
- There is no $u \in O_1$, such that u has no neighbor in O_2, since in this case v would dominate u.
- There is no $u \in V(G)$, such that $d(u) > d(v)$, since the algorithm always branches on a vertex of maximum degree and we assume that v is the vertex on which the algorithm branches.
- $d(v) = 4$.
- (G, v) has radius 2.
- Since Kneis, Langer, and Rossmanith showed that, for a case where $|O_2| \geq 8$, the algorithm's performance is sufficient, the generated cases comply to $|O_2| \leq 7$.

Therefore our objective boils down to: Generate all graphlets (G, v) with radius 2 and $d(v) = 4$, where $d(u) \in \{3, 4\}$ for all vertices $u \in V(G)$, $|O_2| \leq 7$ and all vertices in O_1 have at least one neighbor in O_2.

3.2 Parameters

The behaviour and output of the generation algorithm are modified by three mandatory parameters <minDegree>, <maxDegree>, and <extent>. Because of the restrictions above the <extent> is already fixed at 2 and therefore hard-coded in the program.

The <minDegree> and <maxDegree> specify the minimum and maximum degree that any vertex in the generated cases may have. The algorithm by Kneis, Langer, and Rossmanith branches on graphlets which do not have vertices of degree 1 or 2. Therefore <minDegree> is set to 3. Since cases with $d(v) \geq 5$ where investigated manually, we only need to generate graphlets with a <maxDegree> of 4.

3.3 Process overview

Listing 1.1 visualizes the steps used to generate the relevant cases. We describe the applied steps in a succinct manner. Afterwards, however, we give detailed information on the effect and implementation of the steps.

Listing 1.1. Process overview in pseudo-code.

```
generateStars(minDegree, maxDegree)
makeIntraOrbit1Edges(minDegree, maxDegree)
pickRepresentatives()

appendTrees(minDegree, maxDegree)
foldLeaves(maxDegree)
pickRepresentatives()

makeIntraOrbit2Edges(minDegree, maxDegree)
pickRepresentatives()

appendAnonymousEdges(minDegree, maxDegree)
```

1. We initially invoke `generateStars(minDegree, maxDegree)`. This generates a set $\mathcal{S}$ of star-shaped graphlets. These are all graphlets (G, v) with extent 1, where $d(v) \in \{\texttt{minDegree}, \ldots, \texttt{maxDegree}\}$ and that do not contain any edges between vertices in O_1 (cf. Figure 2).
2. The invocation of `makeIntraOrbit1Edges(minDegree, maxDegree)` generates all relevant graphlets by connecting vertices in the first orbit of the graphlets in $\mathcal{S}$ from the previous step.
3. Afterwards, the `pickRepresentatives()` step is applied for the first time. In this step we determine the isomorphy classes in the set of graphlets generated so far. Then we choose one representative from each class and continue to work on these representatives only, thus reducing the number of graphlets processed further.
4. The call to `appendTrees(minDegree, maxDegree)` generates graphlets by appending new vertices to the vertices on the highest orbit of each graphlet generated so far. This step generates all possible graphlets of extent two, where each vertex in O_2 has exactly one neighbor in O_1 and no further incident edges. Moreover, $minDegree \leq \deg(u) \leq maxDegree$ for all $u \in O_1$ still holds after this step.

5. In the next step, the `foldLeaves(maxDegree)`, the new vertices from step 4 are merged with each other in all possible ways. Doing so, we generate all graphlets of extent two such that $G[O_2]$ contains no edges and without any anonymous edges. This step provides a very inexpensive method of pruning the search-tree (cf. Section 3.4).
6. The invocation of `makeIntraOrbit2Edges(minDegree, maxDegree)` has the same effect as `makeIntraOrbit1Edges(minDegree, maxDegree)`, but on O_2 instead of O_1. Hence, this step generates all graphlets of extent two without anonymous edges.
7. Finally, we add all possible valid combinations of anonymous edges, by calling `appendAnonymousEdges(minDegree, maxDegree)`.

Depending on the used parameters, the memory consumption of the procedure easily exceeds the resources of a conventional computer. Therefore, we made extensive use of disk storage: Between each two steps the intermediate results are stored on the hard-drive. This of course is a major performance penalty, but the obtained runtimes for our scenario were more than acceptable. Listing 1.2 shows the actual script that is used to coordinate the generation of the relevant cases. The steps described above are implemented as autonomous programs which work on sets of graphlets stored on the disk.

Listing 1.2. The script coordinating the generation process.

```
./sinit -m=$1 -M=$1 -o=stage1/init
./sedge -i=stage1 -o=stage2 -M=$1

cd stage2/
for N in *; do
 ../shash -i=$N -o=../stage3/
done
cd ..

cd stage3/
for N in *; do
 ../sfindiso -i=$N -r=$N.r -t=10000
 ../sclean -i=$N -r=$N.r -o=../stage4/$N
done
cd ..

./smerge -i=stage4/ -o=stage5/set

./sexpand -i=stage5/set -o=stage6/set -m=$2 -M=$1

./sfold -i=stage6/set -o=stage7/

cd stage7/
for N in *; do
 ../shash -i=$N -o=../stage8/
done
cd ..

cd stage8/
for N in *; do
 ../sfindiso -i=$N -r=$N.r -t=1000
 ../sclean -i=$N -r=$N.r -o=../stage9/$N
done
cd ..

./sedge -i=stage9 -o=stage10 -M=$1

cd stage10/
```

```
for N in *; do
 ../shash -i=$N -o=../stage11/
done
cd ..

cd stage11/
for N in *; do
 ../sfindiso -i=$N -r=$N.r -t=10000
 ../sclean -i=$N -r=$N.r -o=../stage12/$N
done
cd ..

./smerge -i=stage12/ -o=stage13/set

./sanon -i=stage13/set -o=output/$1-$1.$2-$1.$3-$1 -m=$3 -M=$1
```

Furthermore, the `pickRepresentatives()` function is split into three programs (`shash,sfindiso,sclean`). Note that checking for isomorphisms is absolutely necessary to restrict the number of generated graphlets. Since an exhaustive check is very expensive, we only compare graphlets with the same hash value (see Section 4). This step improves the performance of the generation process dramatically.

3.4 Process sequence

As depicted in Section 3.3, the process consists of several autonomous programs. The programs' usage and implementations are elaborated throughout the next pages. This part of the report is intended to serve as a guideline to understand the programs' implementations, as well as a manual on how to use them.

`generateStars(minDegree, maxDegree)` [`sinit`] The invocation of `sinit` requires certain parameters (cf. Listing 1.3). For every `<m>` $\leq n \leq$ `<M>` an n-Star graphlet is generated and stored in the file `<o>`, where an n-Star is defined as a grahplet (G, v), $G = (\{v\} \cup O_1, E)$ with $O_1 = \{u_1, \ldots, u_n\}$ and $E = \{\{v, u_1\}, \ldots, \{v, u_n\}\}$.

Listing 1.3. Invocation syntax for `sinit`.

```
Invocation: sinit [OPTIONS]
Generates a set of initial graphs for the generation process.

  -m, --mindegree      Specifies the minimum degree of the anchor
                       vertex.
  -M, --maxdegree      Specifies the maximum degree of the anchor
                       vertex.
  -o, --output         Selects the output file (defaults to 'init.out')
```

`makeIntraOrbit`**{1, 2}**`Edges(minDegree, maxDegree)` [`sedge`] The invocation of `sedge` requires three parameters, as described in Listing 1.5.

Let S be the set of graphlets with extent i. For any $(G, v) \in S$, all $x, y \in O_i$ are — by construction — not adjacent, cf. Listings 1.1 and 1.2. Our goal is now to add all possible sets of edges inside O_i to these graphes (see Figure 1). Note that the extent i of a given graphlet is determined automatically by the algorithm.

For this purpose we compute a list L of pairs of vertices whose degree is strictly less than the degree of the anchor vertex. These are exactly the pairs of vertices which we may connect without violating the maximal degree restrictions:

$$L = \{\{x, y\} \mid d(x) < d(v) \wedge d(y) < d(v)\}$$

The algorithm's behaviour is a simple (exhaustive) search for all possibilities, as depicted in Listing 1.4. In Line 7, however, we need to update the candidate list L, since adding edges might disqualify certain pairs.

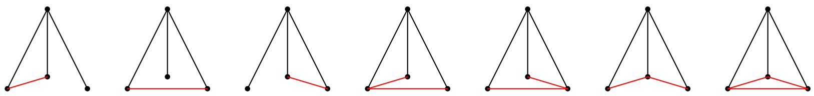

Fig. 1. Constructing edges in O_1 of a 3-star.

Listing 1.4. Sedge pseudo-code.

```
1  For all (G,v) in the input file do {
2      L = computeCandidates()
3      addEdges((G,v), L)
4  }

6  addEdges((G,v), L) {
7      updateList(L)

9      if (isEmpty(L)) {
10         writeToDisk(G,v)
11         return.
12     }

14     {x,y} = firstElementOf(L)
15     L' = L - {x,y}

17     /* Realize the edge */
18     (G',v) = (G,v) where E[G'] = E[G] + {x,y}
19     addEdges((G',v), L')

21     /* Do not realize the edge */
22     addEdges((G,v), L')
23 }
```

Listing 1.5. Invocation syntax for `sedge`.

```
Invocation: sedge [OPTIONS]
Connects the orbital vertices of a set of graphs.

  -i, --input       The graph collection to fold.
  -o, --output      A file to store the connected graphs.
  -M, --maxdegree   The maximum degree of each vertex.
```

`appendTrees(minDegree, maxDegree)` [sexpand] Calling `sexpand` requires four parameters. They are described in Listing 1.7.

Given a set of graphlets S with extent 1 this step constructs a new set S' of graphlets with extent 2 as follows: for each graphlet $(G, v) \in S$, consider its outermost orbit $O_1 = \{v_1, \ldots, v_n\}$. For every such vertex $v_i \in O_1$ we then calculate the set

$$n(v_i) := \{a \in \mathbb{N} \mid \texttt{<m>} \leq a \leq \min\{\texttt{<M>}, d(v)\}\}$$

If, for example, $n(v_i) = \{2, 3, 4\}$, the vertex v_i can have $2, 3$ or 4 neighbors in O_2 without $d(v_i)$ being smaller than `<m>`, or too high.

Using the graphlet (G, v) and some choice $a_i \in n(v_i)$ for all $1 \leq i \leq n$ we create a new graphlet (G', v) by attaching a_i new vertices to the vertex v_i. Consider a graphlet (G, v) such that $V(G) = \{v\}$. The expansions results for $< m >= 3$ and $< M >= 5$ are illustrated in Figure 2.

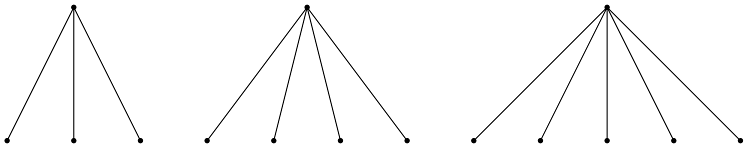

Fig. 2. Expanding a single vertex for $n(v) = \{3, 4, 5\}$.

Every possibility to choose the a_i from the $n(v_i)$ yields a graphlet. The set of the graphlets obtained by using all possible choices for the a_i are added to the set S'. S' is used as input for the next step.

For implementation details refer to the pseudo code in Listing 1.6.

Listing 1.6. Sexpand pseudo-code.

```
1  For all (G,v) in the input file do {
2      Let O = outerMostOrbit(G,v)
3      expand((G,v), O)
4  }

6  expand((G,v), O) {
7      if (isEmpty(O)) {
8          writeToDisk(G,v)
9          return.
10     }

12     u = firstElementOf(O)
13     for all i in n(u) do {
14         (G',v) = appendFreshVertices(i, u, (G,v))
15         expand((G', v), O − u)
16     }
17 }
```

In line 14 the new graphlet (G', v) is obtained for some vertex u and some choice $i \in n(u)$ by

$$V(G') = V(G) \,\dot{\cup}\, \{v_1, \ldots, v_{|n(u)|}\} \text{ and } E(G') = E(G) \cup \{\{u, v_1\}, \ldots, \{u, v_{|n(u)|}\}\}$$

Listing 1.7. Invocation syntax for `sexpand`.

```
Invocation: sexpand [OPTIONS]
Expands a set of graphs by appending new vertices to the outermost
orbit.

  −i, −−input      The graph collection to expand.
  −o, −−output     The desired output−file.
  −m, −−mindegree  The minimum degree of a vertex.
  −M, −−maxdegree  The maximum degree of a vertex.
```

`foldLeaves(maxDegree)` [`sfold`] The invocation of `sfold` requires two parameters, as depicted in Listing 1.9.

Let S be a set of graphlets of extent $k+1 \in \mathbb{N}$ generated by the `sexpand` program. Obviously there is no graphlet in S such that two vertices in O_k have a common neighbor in O_{k+1} since for all $u \in O_{k+1}$ it holds $d(u) = 1$. We then start to fold the vertices in the outermost orbit with each other.

Definition 5. *Let $u, v \in O_{k+1}$, $u \neq v$ and $N(u) \cap N(v) = \emptyset$. We fold u, v by introducing a new vertex z and connecting it to all neighbors of u, v. Afterwards we delete u, v from the graphlet.*

For each graphlet $(G, v) \in S$ we start an exhaustive search on all possible ways to fold vertices in O_{k+1}. Refer to the following figure for an example.

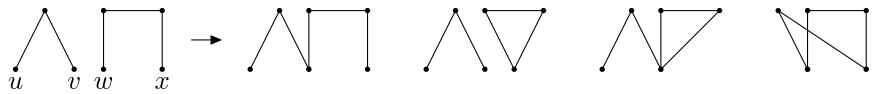

Fig. 3. The graphlets obtained by folding vertices u, v, w, x in the highest orbit in all possible ways (isomorphic graphs are omitted).

Listing 1.8. `sfold` pseudo-code.

```
1  For all (G,v) in the input file do {
2      P = pairsOfVerticesInOutermostOrbit()
3      foldVertices((G,v), P)
4  }

6  foldVertices((G,v), P) {
7      updatePairs(P)

9      if (isEmpty(P)) {
10         writeToDisk(G, v)
11         return.
12     }

14     {x,y} = firstElementOf(P)
15     P' = P - {x,y}

17     /* Do not fold the vertices */
18     foldVertices((G,v), P')

20     /* Vertices foldable? */
21     if (areDisjunctive(N(x), N(y))) {
22         (G',v) = fold((G,v), x, y)
23         foldVertices((G',v), P')
24     }
25 }
```

Note that the graphlet (G', v) in Line 22 is obtained by performing the following steps:

1. Add a new vertex u and remove the vertices x, y: $V(G') = (V(G) \mathbin{\dot\cup} \{u\}) \setminus \{x, y\}$.
2. Connect u to all neighbors of x, y: $\{x, z\} \in E(G) \vee \{y, z\} \in E(G) \Rightarrow \{u, z\} \in E(G')$.

The call to `updatePairs(P)` has two purposes. First it removes vertex pairs which cannot be folded anymore because either their neighborhoods now overlap or because of the removed vertices.

Second it creates new pairs for a new vertex — in case we applied folding in the previous round — and adds them to P.

Listing 1.9. Invocation syntax for `sfold`.

```
Invocation: sfold -i=input -o=output
Folds the orbital vertices of a set of graphs.

  -i, --input       The graph collection to fold.
  -o, --output      A file to store the folded graphs.
```

`pickRepresentatives()` [`shash`,`sfindiso`,`sclean`] The objective of these three programs is to pick representative graphlets from the present isomorphy classes. As a first step the `shash` program splits the present files into several new files. The file that a graphlet is saved to depends on its hash value (cf. Section 4).

Afterwards, exploiting that $(G, v) \cong (G', v') \Rightarrow h(G, v) = h(G', v')$, the `sfindiso` program searches for isomorphic graphlets, one file at a time, using a straightforward isomorphism checking algorithm. Finally the `sclean` program removes all members of an isomorphism class except for one representative.

Listing 1.10. Invocation syntax for `shash`.

```
Invocation: shash [OPTIONS]
Splits a set according to the hash value of each graph.

  -i, --input       The input file.
  -o, --output      A file to store the hash files.
```

Listing 1.11. Invocation syntax for `sfindiso`.

```
Invocation: sfindiso [OPTIONS]
Locates pairs of isomorphic graphs and writes their index into a file.

  -i, --input       The graph collection to search.
  -r, --report      A file to report the located isomorphisms to.
  -T, --time        A global time limit. After this time has passed,
                    the tool stops looking for isomorphic graphs.
  -t, --steps       The number of non-deterministic steps the
                    isomorphism checking algorithm is limited to for
                    each pair of graphs.
  -o, --offset      The index of the first graph to check.
```

Listing 1.12. Invocation syntax for `sclean`.

```
Invocation: sclean [OPTIONS]
Locates pairs of isomorphic graphs and writes their index into a
file.

  -i, --input       The graph collection to clean.
  -r, --report      The report file from the isomorphism-tracker.
  -o, --output      The desired output-file.
```

4 Hashing function

Throughout the generation process, we confine the number of considered cases to a necessary minimum by considering only representatives of isomorphism classes.

In this scenario we are able to exploit additional information about the graphlet-isomorphisms to speed up the isomorphism checking. The checking procedure, however, is still computational expensive. Therefore we introduce the following means to reduce the number of required isomorphism checks.

Let S be a set of cases. We decompose S into several sets $S_1, S_2, \ldots$ For each graphlet $(G, v) \in S$ we use a serial version of Berkowitz' algorithm [1] to determine the coefficients of the characteristic polynomial of G's adjacency matrix. Graphlets are distributed into the sets S_i according to the coefficients in their respective characteristic polynomial[1].

Let (G, v), (G', v') be graphlets. If they are isomorphic, then $G \cong G'$ also holds. Therefore their adjacency matrices are permutations of each other. Thus they must have the same characteristic polynomial.

Altogether, two isomorphic graphlets will be contained in the same set S_i. Therefore it suffices to perform pairwise isomorphy checks on graphlets from the same set.

Also, in case the implementation of Berkowitz' algorithm was incorrect, we would not miss any relevant cases.

5 Completeness

Theorem 1. *Let S be the set of graphlets generated by our algorithm for an radius of 2, a minimum degree of 3 and a maximum degree of 4. Then for every relevant case (G, v), relevant to algorithm, there is a case $(G', v') \in S$, such that $(G, v) \cong (G', v')$.*

In this section we prove that an arbitrary graphlet with extent 2 is generated by our algorithm (for given `<minDegree>`,`<maxDegree>`). Recall the generation process' overview in Listing 1.1.

To improve the proof's readability, we will not distinct between isomorphic graphlets anymore. If $(G, v) \cong (G', v')$ we treat them as equal.

Proof. Let (G, v) be a graphlet with extent 2, $d(v) = 4$ and $V(G) = \{v\} \cup O_1 \cup O_2$ with $O_1 = \{u_1, \ldots, u_4\}$, $O_2 = \{w_1, \ldots, w_n\}$ where $n \leq 7$. Furthermore $d(x) \in \{3, 4\}$ for all $x \in V(G)$ and there is no $u_i \in O_1$ that has no neighbor in O_2. We will prove that (G, v) is generated by our algorithm.

Let $T_1 = (G[\{v\} \cup O_1], v)$ without edges in O_1. In Line 1 the call to `generateStars` generates only the 4-star graphlet. Since $d(v) = 4$ we know that T_1 is generated in the first line.

Consider Line 2 and assume that T_1 has been generated so far. Let $T_2 = (G[\{v\} \cup O_1], v)$ be the graphlet induced by the first orbit O_1 and the anchor vertex v. Since (G, v) is a relevant case the edges in $E[G] \cap \binom{O_1}{2}$ are also added in one path of the exhaustive search tree employed by `makeIntraOrbit1Edges`. Therefore T_2 is generated by the second line.

Since, during the proof, we do not distinguish between isomorphic graphlets the `pickRepresentatives`-calls are not interesting.

Consider Line 5. Let, for some $u_i \in O_1$, $e(u_i)$ be the number of u_i's neighbors in O_2 w.r.t. (G, v). Let $T_3 = (G', v)$ where $V(G') = \{v\} \cup O_1 \cup \{x_1, \ldots, x_m\}$

[1] Due to limitations in the file system, we were forced to reduce the quality of this separation, thus obtaining a smaller number of files.

where $m = \sum_{i=1}^{4} e(u_i)$. Moreover $E(G') = \{\{v, u_1\}, \ldots, \{v, u_4\}\}$ and every u_i is connected to $e(u_i)$ unique vertices in $\{x_1, \ldots, x_m\}$. Hence for all x_i it holds $d(x_i) = 1$. So T_3 equals T_2 with new vertices of degree 1 attached to O_1.

Since the call to `appendTrees` performs an exhaustive search, at least one of the leafs in the search tree provides T_3 (under the assumption that T_2 was obtained by the previous steps).

Consider Line 6 and assume T_3 was generated by now. Let $T_4 = (G, v)$ but without anonymous edges and without edges in the second orbit. Since `foldLeaves` performs an exhaustive search on all possibilities to fold vertices, we only need to show that T_3 can be folded into T_4.

Let $x \in O_2(T_4)$ and $y_1, \ldots, y_k$ its neighbors in O_1. The y_i have $e(y_i)$ neighbors with degree 1. For each y_i, y_{i+1} we take one of their unused neighbors n_i, n_{i+1} in O_2 and fold them together. This is always possible since $N[n_i] \cap N[n_{i+1}] = \{y_i\} \cap \{y_{i+1}\} = \emptyset$. We obtain a new vertex z that is connected to y_i, y_{i+1}. We can now fold z, y_{i+2}, etc. Afterwards all y_i have been folded into a single new vertex which resembles x in T_4. We employ the same strategy for the remaining vertices in $O_2(T_4) \setminus \{x\}$. After we have done this, $T_3 = T_4$. Moreover, the number of available vertices for folding is always sufficient, since the number of edges between O_1 and O_2 are not changed during folding.

Thus, under the assumption that T_3 is generated by the previous steps, we will obtain T_4 in Line 6.

Let $T_5 = (G, v)$ without any anonymous edges. By the same argument as regarding the second line, T_5 is generated under the assumption that T_4 was generated before.

In the last step, we perform an exhaustive search on the possibilities to add anonymous edges. Assuming that T_5 was generated by the previous steps, at least one leaf in the search tree will provide (G, v).

Thus, neglecting isomorphisms, (G, v) will be generated by the algorithm. Therefore the algorithm generates all relevant cases.

Since the completeness is crucial for the validity of Kneis, Langer, and Rossmanith's work [3], Reidl & Sánchez Villaamil [5] devised a more readable — and therefore slower — program, used to verify that the presented algorithm generates all relevant cases. For information on their implementation refer to the respective technical report [5].

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. F. Reidl and F. Sánchez Villaamil. Automatic verification of the correctness of the upper bound of a maximum independent set algorithm, 2009. Available at `http://www.tcs.rwth-aachen.de/independentset/`.
6. J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.

7. R. E. Tarjan and A. E. Trojanowski. Finding a Maximum Independent Set. *SIAM Journal on Computing*, 6(3):537–550, 1977.