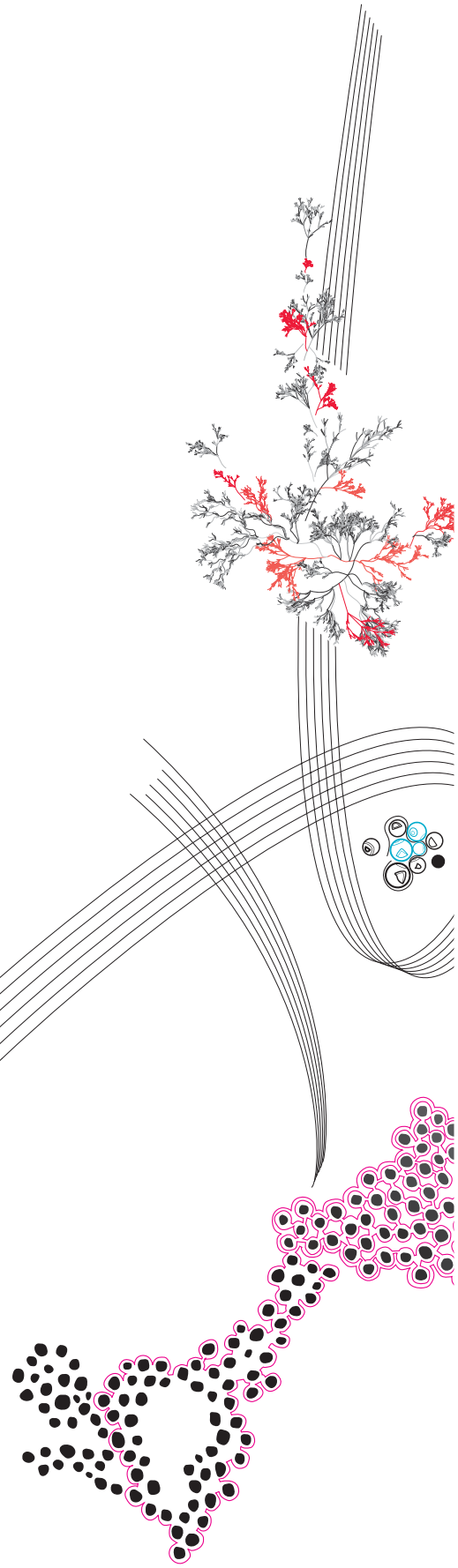

MSc Thesis Applied Mathematics

An implementable Three-in-a-Tree algorithm to accelerate Perfect Graph detection

H.A. Hof

Daily supervisor: dr. M. Walter
Graduate supervisor: prof. dr. M.J. Uetz
Committee member: dr.ing. A.B. Zander

August, 2023

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Acknowledgements

I would like to express my deepest gratitude to my supervisor Matthias Walter for his input and the many fruitful discussions about the contents of this thesis. I would also like to thank the other members of my assessment committee. I would like to thank my father for showing me the beauty of mathematics from a young age. I am very grateful for the support that both my parents gave me throughout my studies. I would also like to thank all my friends and fellow mathematics students for a great atmosphere to discuss mathematics while also enjoying our time in Enschede.

An implementable Three-in-a-Tree algorithm to accelerate Perfect Graph detection

H.A. Hof*

August, 2023

Abstract

In this thesis we implement the algorithm of Chudnovsky et al. (2005) to determine if a given graph contains an odd-hole or anti-hole. In particular this decides whether a given graph is perfect according to the strong perfect graph theorem proved by Chudnovsky et al. (2006). A main bottleneck of this algorithm is the detection of pyramids in a given graph. We introduce the three-in-a-tree problem and develop new sub-algorithms to make the decision algorithm of Lai et al. (2020) self-standing and not based on other highly theoretical results. This new algorithm allows us to actually implement the three-in-a-tree algorithm and also leads to a reduced running time of detecting pyramids by three orders of magnitude.

Keywords: Perfect graphs, Berge graphs, Pyramids, Three-in-a-tree, Induced subgraph detection, Odd-holes

*Email: hugohof1996@gmail.com

Contents

1 Introduction

Graphs are one of the most important concepts in discrete optimisation. Studying graph-related problems is therefore of great interest in this field. Perfect graphs are graphs where for every subgraph the chromatic number and the size of the largest clique are the same. In such graphs the largest clique and stable set problems are, amongst others, solvable in polynomial time [16], while these are NP-hard problems in general.

For a complete overview of basic graph theory we refer to the book by Diestel. For a graph $G = (V, E)$ with vertices V and edges E, we refer to the number of vertices as $|V| = n$, and the number of edges as $|E| = m$.

An odd-hole of a graph is an induced cycle of odd length. In order to determine whether a graph is perfect Berge conjectured that a graph is perfect if and only if the graph and its complement are odd-hole free [2]. A graph that contains no odd-hole and no odd anti-hole is referred to as a Berge graph. In 2005 the Strong Perfect Graph theorem proved Berge's conjecture [8]. During this same period of time an algorithm was developed to determine if a graph is Berge [6], and consequently a perfect graph.

This algorithm to determine perfect graphs essentially relies on three different sub-algorithms. First, various forbidden substructures of the graphs are found, which indicate a graph is never perfect. For all graphs without these substructures it is then possible to determine whether they contain an odd-hole. Because these forbidden substructures are not present, during the second and third sub-algorithms we are able to detect an odd hole or anti-hole. A schematic overview of the algorithm is shown in Figure 1.

In Section 3 the algorithm to detect perfect graphs is introduced along with implementation strategies. For this research the algorithm to detect perfect graphs was implemented, as far as we can tell for the first time. We are then able to review not only the proven upper-bounds of the algorithm, but also its effectiveness on various randomly generated graphs.

Both the algorithm to determine substructures, as well as the second and third algorithms to find holes in the remainder of the graphs have bottlenecks that lead to a running time of $O(n^9)$ as seen in Figure 1. There is, however only one forbidden structure with this running time, whilst all others have a running time of $O(n^6)$ or lower. This substructure is a so called pyramid. As suggest by the original authors in a later paper [5], these pyramids are closely linked to another graph problem, namely three-in-a-tree.

The three-in-a-tree problem is a decision problem that decides whether for a given graph and three vertices, there exists an induced tree connecting the three vertices. While the relation between pyramids and three-in-a-tree problems was already described in [5], they were not able to develop an algorithm that was fast enough to compete with the original $O(n^9)$ running time. However, a way faster algorithm for the three-in-a-tree problem was developed in 2020 by Lai et al. [18].

This new algorithm decides for a graph and three given vertices whether it contains a three-in-a-tree for these vertices in time $O(m \log n)$, or as the authors describe it *near-linear* regarding the fact that $O(m \log n) = \tilde{O}(m)$. This new algorithm consequently leads to an algorithm with a running time of $O(n^5 \log n)$ to find pyramids.

We first review the idea behind the algorithm of Lai et al. in Section 4. Several aspects of this algorithm rely on various yet to be implemented highly theoretical results of other papers, such as dynamic and incremental SPQR-trees [13], $O(\log n)$ dynamic spanning forests [17] and top space forests [1]. Continuing our efforts to implement an algorithm to detect perfect graphs, we develop various new sub-algorithms for the three-in-a-tree algorithm that are actually implementable in Section 5. These new algorithms lead to an

overall running time of $O(n^3)$ to decide the three-in-a-tree problem, and a running time of $O(n^6)$ to detect pyramids. This slightly slower overall running-time is justified on the one side from its actual 'implementability', while on the other hand it also ties the running time of pyramids with many of the other forbidden substructures.

Afterwards, in Section 6 we show the results of our perfect graph and three-in-a-tree algorithm. We investigate both algorithms separately, and compare their performance to detect pyramids.

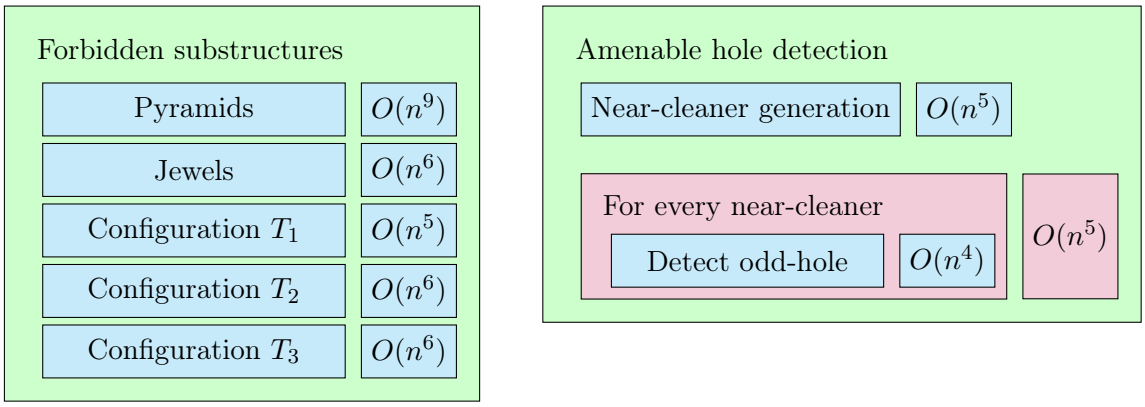

FIGURE 1: Running time of sub-algorithms in perfect graph detection of Chudnovsky et al. (2005)

.

1.1 Related literature

The algorithm to detect Berge graphs can only determine if a given graph contains either an odd hole or an odd anti-hole. The problem to determine whether a graph contains an odd hole through a given vertex has already been shown to be NP-complete [3]. This lead to the belief that deciding whether a graph contains any odd-hole was also NP-complete. Recently, however, efficient algorithms have been found to determine whether a graph contains a (shortest) odd hole [10, 11]. Some results to bound the chromatic number of (families of) odd hole free graphs have also been found [9, 21, 22], although these are surely not as strong as the results for Berge graphs.

Moreover, research has also been on detecting even holes in graphs [12, 7, 4]. The newest algorithm to detect even holes is, just like the algorithm to detect Berge graphs, also based on a decomposition algorithm. During this algorithm another kind of substructure called a 'beetle' is detected. These beetles can also be found using a three-in-a-tree algorithm [4], and thus by our implementation.

2 Background

2.1 Preliminaries

All graphs that are considered in this thesis will be simple and undirected. We denote such a graph as $G = (V, E)$, where V denotes the *vertices* and E denotes the *edges* of the graph. The size of the vertex and edge sets will be denoted by $n := |V|$ and $m := |E|$ respectively. Edges of the graph $e \in E$, can also be referred to as $e = (u, v)$, where $u, v \in V$ are the endpoints of the edge in the graph. We say that vertices u and v are *adjacent* if $(u, v) \in E$, and refer to the set of vertices adjacent to v as its *neighbours* $N(v)$. The degree

of a vertex $v \in V$ is equal to the number of neighbours it has in the graph, i.e. the size of $N(v)$. For a set S of vertices, respectively edges, we write $|S|$ to indicate its number of vertices, respectively edges.

An *induced subgraph* of a graph G is another graph $G' = (V', E')$ that is induced by taking a subset $V' \subseteq V$. The graph G' consists of all vertices V', together with all edges $(u, v) \in E$, such that $u, v \in V'$. We may also say that the set V' *induces* the subgraph G'.

A path $P = v_0, v_1, ..., v_k$ of a graph G is a walk along the edges of the graph such that every vertex on the walk is visited only once. The length k of such a path is equal to the number of edges it contains, and we refer to *even* and *odd-length paths*. Moreover, all vertices $v \in P$ have degree two, except for the *end-vertices* v_0 and v_k, which have degree one. Denote by $d_G(u, v)$ the *distance*, that is the length of the shortest path, between vertices u and v on graph G.

A cycle C of a graph is a path where the first and last vertex of the path are the same. We similarly refer the *even* and *odd-length cycles*. If the induced subgraph of the vertices on the cycle is also a cycle, meaning all vertices of this induced graph have degree exactly 2, we call the cycle C a *hole*. Similarly, an *anti-hole* is an induced cycle in the complement graph $\bar{G}$. Odd holes and odd anti-holes are induced cycles that have odd length.

A *component* of a graph is defined as a connected subgraph that is not a subset of any other connected subgraph, and an *anti-component* is a component in the complement graph $\bar{G}$.

Given a set $S \subseteq V$, an *S-complete vertex* v is a vertex such that there exists an edge from v to every $s \in S$, and similarly an *S-complete edge* $e = (u, v)$ is an edge such that both u and v are S-complete.

2.2 Perfect graphs

A *complete* graph G is a graph that has an edge $(u, v) \in E$ between every pair of vertices $u, v \in V$. A *clique* of a graph is a complete subgraph induced by vertices $K \subseteq V$. The size of a clique is equal to the number of vertices $|K|$ and we call the size of the largest clique the *clique number* of the graph.

Vertices of a graph can be coloured using a *vertex colouring*, where two vertices that are adjacent must be assigned different colours. The *chromatic number* of a graph is defined as the minimum number of colours needed for such a vertex colouring. It becomes immediately clear that the chromatic number of a graph is at least as large as its clique number, as every vertex in a clique must be assigned a different colour. We are now ready introduce a very important class of graphs that follows from this observation.

Definition 2.1 (Perfect graphs)**.** A graph G is said to be *perfect* if for every induced subgraph of G its clique number is equal to its chromatic number.

An example of a perfect graph is shown in Figure 2. The chromatic number and the maximum size of a clique of the graph are 3. It is easy to see that this statement also holds for all possible subgraphs. In Figure 3 an imperfect graph is shown. The maximum size of a clique of a 5-cycle is clearly 2, while the chromatic number is 3. As we will see in the following section, it is not a coincidence that this graph is an odd-hole.

Perfect graphs have a range of useful properties. For example, for such graphs the chromatic number and clique number can be determined in polynomial time [16], while this is an NP-hard problem in the general case.

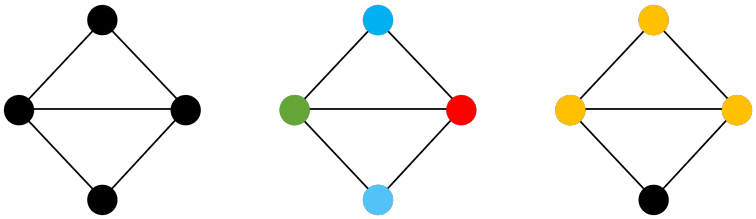

FIGURE 2: Perfect graph with vertex coloring and a max-clique in yellow

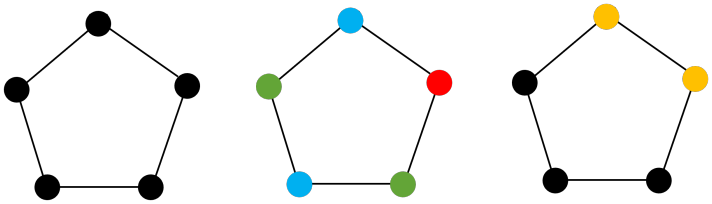

FIGURE 3: Imperfect graph with vertex coloring and a max-clique in yellow

2.3 Strong perfect graph theorem

In 1961 Berge conjectured that every minimal imperfect graph is either an odd hole or an odd anti-hole [2]. From this conjecture the following definition arose:

Definition 2.2 (Berge graphs)**.** A graph G is said to be a *Berge graph* if does not contain any odd hole or odd anti-hole.

The proof of the conjecture was finally announced in 2002 by Chudnovsky, Robertson, Seymour and Thomas [8]. Their proof spanned over 150 pages and is now known as the *Strong Perfect Graph Theorem.*

Theorem 2.1 (Strong Perfect Graph Theorem)**.** *A graph is perfect if and only if it is a Berge graph.*

Sufficiency proof. Assume we have a graph G that is perfect but not Berge. This graph then either contains an odd hole or an odd anti-hole. Note that the complement of every perfect graph is also perfect, so we assume without loss of generality that G contains an odd hole. Clearly if we take the subgraph induced by the vertices in this hole we obtain an odd cycle. The clique number in an odd cycle is 2, while the chromatic number is 3. This is in contradiction with the assumption that the graph is perfect.

The converse is way more difficult to show. For the beautiful and acclaimed proof we refer to [8]. □

One of the consequences of the theorem above is that perfect graphs can be recognised using a polynomial time algorithm for Berge graphs. In other words, if we can successfully determine whether a graph or its complement contains an odd hole, we can conclude whether it is a perfect graph. An algorithm to determine whether a graph is Berge will be further described in Section 3.

3 Berge graph detection

In this section we introduce and review a polynomial-time algorithm to detect Berge graphs. The algorithm consists of three different components that will allow us to find odd holes and anti-holes in graphs. The three components are as follows [6]:

1. **Forbidden substructures**
   We first introduce five types of induced subgraphs that will always imply the existence of an odd hole in the graph. If none of these types are present it allows us to exploit some nice properties of the graph.

2. **Amenable holes and near-cleaners**
   Secondly, the concept of near-cleaners will be introduced. When none of the forbidden substructures are present it will allow us to generate polynomially many sets such that one will be a near-cleaner for an odd hole.

3. **Odd hole detection**
   Lastly, we need an algorithm that allows us to always find an odd hole of the graph if we are given a near-cleaner that is presents in the sets above, or conclude that the graph contains no odd holes.

In order to determine whether a graph is Berge, we need to run the algorithm above twice. First we run the forbidden substructure algorithm on the graph and its complement. After that the algorithm above will successfully determine whether an odd hole is present in the graph. If that is the case, we conclude that the graph is not Berge. Repeating the algorithm for the complement graph will then also detect odd anti-holes. If those exist we also conclude that the graph is not Berge. If no odd holes and no odd anti-holes are present the graph is Berge.

In the following sections we will introduce all algorithms and implementations ideas that are needed to implement an algorithm to detect Berge graphs. During this evaluation of the algorithm, for some of the proofs and algorithms we will refer to the paper with the original algorithm by Chudnovsky et al. [6]. We focus mostly on giving implementation ideas as well as give some additional motivation for some of the ideas.

3.1 Forbidden substructures

The first part of the Berge graph algorithm consists of finding forbidden-substructures. These structures are called pyramids, jewels, and configurations T_1, T_2 and T_3. It is only after determining that a graph and its complement contain no such structures, that the algorithm using near-cleaners for amenable holes will work. In this section we will introduce the substructures, show how and with which running time we can recognise them, and show why they always lead to odd-holes.

3.1.1 Pyramids

A pyramid is the first type of an induced subgraph that we will introduce. We will introduce every one of the forbidden substructures as a graph, but it is important to always keep in mind that the induced graph cannot contain edges that are not part of the structure. This is due to the fact that every subgraph induced by a set of vertices $X \subseteq V$, contains every edge of the original graph connecting vertices from X.

Definition 3.1 (Pyramid)**.** A graph $G = (V, E)$ is a *pyramid* if it contains of three base vertices $b_1, b_2, b_3 \in V$, a top vertex $a \in V$ and paths P_i that connects a to b_i, such that:

1. Every edge $(b_i, b_j) \in E$ for $i \neq j$;

2. For paths P_i and P_j with $i \neq j$, we have $P_i \cap P_j = \{a\}$ and for every $v_k \in P_i \backslash \{b_i, a\}$ and $v_l \in P_j \backslash \{b_j, a\}$ there is no edge $(v_k, v_l) \in E$;

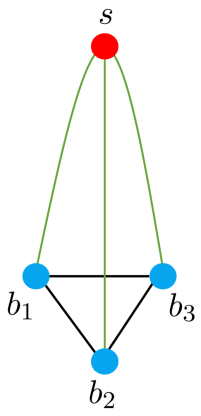

FIGURE 4: A pyramid graph, with the three base vertices in blue and the top vertex in red. Green lines indicate paths, of which at most one has length 1.

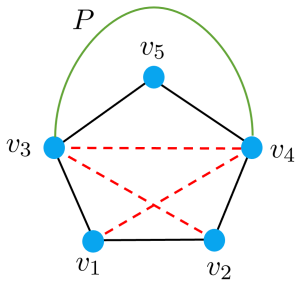

FIGURE 5: A jewel graph, red dotted lines indicate non-edges, in green path P with length at least 2.

3. At most one path P_i, $i = 1, 2, 3$ has length 1.

An example of a pyramid can be seen in Figure 4. The three blue vertices in the bottom form the triangle of the base, and the red vertex is the top vertex a. It is not difficult to prove that a pyramid always contains an odd-hole.

Theorem 3.1 (Pyramids contain odd-holes)**.** *Every pyramid with base b_1, b_2, b_3, a top vertex a and three paths P_1, P_2, P_3 as in Definition 3.1, contains an odd-hole.*

Proof. It is immediately clear that each cycle b_m- P_m -P_n -b_n-b_m is also a hole from the definition of a pyramid. We only need to ensure that at least one of these holes is an odd-hole. Because we have three paths P_i, $i = 1, 2, 3$ we know that there exist $k, l \in \{1, 2, 3\}$ such that $k \neq l$ and P_l and P_k have the same parity. In both cases b_k- P_k -P_l -b_l-b_k has odd length and is an odd-hole. □

Note that importantly the paths that connect the base vertices to the top of the pyramid are *not* necessarily the shortest paths between those vertices. Therefore, it is not immediately clear how to test for induced pyramids in a given graph, as simply trying all paths between all b_i's and a will not have a sufficiently low running time bound. Because we only need to determine whether a pyramid exists as an induced subgraph, we instead only look for a vertex-minimal pyramid. A pyramid induced by a set $K \subseteq V$ of a graph $G = (V, E)$ is vertex-minimal if there exists no other set $K' \subseteq V$ that also induces a pyramid and $|K'| < |K|$. Clearly, such a vertex-minimal pyramid exists if there exists an induced pyramid in a graph. Such a vertex-minimal pyramid has a special structure that can be used to determine if a graph contains a pyramid. Due to this special structure, there exists an algorithm to detect pyramids in time $O(n^9)$. Moreover, these vertex-minimal pyramids can be represented using pyramid frames.

Definition 3.2 (Pyramid frame)**.** A *pyramid frame* consists of the four pyramid vertices b_1, b_2, b_3 and a, as well three *middle vertices* m_1, m_2, m_3 and three *start vertices* s_1, s_2, s_3, such that:

1. Vertex s_i is the second vertex on the path P_i from a to b_i

2. Vertex m_i is the vertex in the middle of the path P_i from a to b_i, possibly closer to b_i. To clarify, denoting with $d_P(v, w)$ the distance on path P from v to w, we have $d_{P_i}(a, m_i) - d_{P_i}(m_i, b_i) \in \{0, 1\}$.

Note that in a pyramid there might be one vertex b_i that is a neighbour of a, and in this case $b_i = s_i = m_i$. If the distance from b_i to a is two, we have $m_i = s_i$. In all other cases the three vertices are uniquely defined. For vertex-minimal pyramids with a given frame we can always replace paths P_i with shortest sub-paths through the set of vertices not neighbouring all b_j's and s_j's for $j \neq i$. This statement is clarified below. The proof of the following Theorem can be found in [6].

Lemma 3.2 (Vertex-minimal pyramid frames - 2.1 in [6].)**.** *We are given a graph $G = (V, E)$ and a set $S \subseteq V$ that induces a vertex-minimal pyramid with a frame as is Definition 3.2. Let P_i be the path from b_i via m_i to s_i on the pyramid with sub-paths S_i between s_i and m, and T_i between m and b_i. Define P_j, P_k similar for $i \neq j \neq k$, $i, j, k \in \{1, 2, 3\}$. If G' is the graph induced by the vertices that are not a neighbour of s_j, b_j with $j \in \{1, 2, 3\}, j \neq i$, then for every shortest path S_i' between s_i and m, and T_i' between m and b_i in G':*

1. *The path P_i' induced by the vertices of $S_i' \cup T_i$, together with P_j, P_k form a vertex-minimal pyramid.*

2. *The path P_i' induced by the vertices of $S_i \cup T_i'$, together with P_j, P_k form a vertex-minimal pyramid.*

Because of Lemma 3.2, we can replace both paths S_i and T_i on a pyramid with a shortest path S_i' and T_i' respectively and still obtain a vertex-minimal pyramid. These shortest sub-paths do now allow us to determine if a graph contains a pyramid. This is done in a few steps to make the algorithm run in the promised running time of $O(n^9)$. First we need to find construct the pyramid frame by enumerating b_i's and s_i's for $i = 1, 2, 3$ and determining all promising paths $P_i(m_i)$ for all possible m_i's.

Corollary 3.2.1. *If a graph $G = (V, E)$ contains a pyramid with given partial frame $b_1, b_2, b_3, s_1, s_2, s_3, a$, we can generate three sets $\mathcal{P}_i$ each containing $O(n)$ paths, such that there are $P_1 \in \mathcal{P}_1$, $P_2 \in \mathcal{P}_2$, $P_3 \in \mathcal{P}_3$ that together form a pyramid using Algorithm 1 in time $O(n^3)$.*

Proof. Correctness follows directly from Lemma 3.2. Finding all shortest paths in a graph can be done in time $O(n^3)$ [15]. The loop over s_i is done $O(1)$ times, the loop over $m \in M \cup \{b_i, s_i\}$ is done $O(n)$ times. Things like checking that the pats S_i and T_i only share one vertex, and induce a path together can surely be done in time $O(n)^2$. □

The algorithm above generates possible paths between base vertices and the top-vertex, that could potentially form a pyramid. In particular, if a vertex-minimal pyramid exists, there are paths $P_1(m_1), P_2(m_2), P_3(m_3)$ that form a vertex-minimal pyramid by Lemma 3.2. What remains is to find a method to test whether these paths are disjoint and do not have any edges between them except between the base vertices. We could just simply check for every combination of m's whether this is the case, but that will be way too slow. If paths $P_1(m_1)$ and $P_2(m_2)$ do obey the statement above, we call vertices m_1 and m_2 a *good* pair. We will first generate all good pairs as follows.

Lemma 3.3 (Good path pairs)**.** *Given two lists of paths $\mathcal{P}_i$ and $\mathcal{P}_j$ containing paths for each $m_i \in M \cup \{b_i, s_i\}$ and $m_j \in M \cup \{b_j, s_j\}$ respectively, we can determine all good pairs (m_i, m_j) in time $O(n^3)$ using Algorithm 2.*

Algorithm 1 Promising paths

Input: Graph $G = (V, E)$ and vertices $b_1, b_2, b_3, s_1, s_2, s_3, a$
Let $M = V \backslash \{b_1, b_2, b_3, s_1, s_2, s_3\}$.
for each s_i, $i = 1, 2, 3$ **do**
    Let $L = \{v \in V | v \text{ is a neighbour of } b_j, s_j, j \neq i\}$.
    Let $F = V\ L + \{b_i, s_i\}$.
    Find a shortest path $S_i(m)$ between s_i and m for all possible $m \in M$.
    Find a shortest path $T_i(m)$ between m and b_i for all possible $m \in M$.
    for every possible $m \in M \cup \{b_i, s_i\}$ **do**
        if $m = b_i$ **then**
            $P_i(m) = \{b_i\}$.
        else if $m = s_i$ **then**
            $P_i(m) = \{s_i, b_i\}$.
        else
            Check if m is not adjacent to b_k, s_k for $k \neq j$.
            Check if both sub-paths $S_i(m)$ and $T_i(m)$ exist.
            Check if $S_i(m) \cap T_i(m) = \{m\}$.
            Check if $S_i(m) \cup T_i(m)$ induces a path.
            if All statements above hold **then**
                Let $P_i(m)$ be the path induced by $S_i(m) \cup T_i(m)$.
            else
                Let $P_i(m)$ be empty.
            end if
        end if
    end for
end for

Algorithm 2 Good path pairs

for each $m_i \in M \cup \{b_i, s_i\}$ **do**
    if $P_i(m_i)$ does not exist **then**
        continue.
    end if
    Colour each vertex in $M \cup \{b_i, s_i\}$ that belongs to, or is a neighbour of $P_i(m_i)$ black, and all other vertices white.
    for each $m_j \in M \cup \{b_j, s_j\}$ **do**
        if path $P_j(m_j)$ contains only white vertices **then**
            Label (m_i, m_j) as a good pair.
        end if
    end for
end for

Proof. By colouring vertices the algorithm still obviously performs as expected. For the running time note that there are $O(n)$ vertices m_i's and m_j's. Because the paths are predetermined in Algorithm 1, looping through the vertices and checking their colour only takes $O(n)$ per pair (m_i, m_j). Consequently, the total running time is as claimed. □

Now that we can determine possible paths as well as find good pairs of paths, we only need to find a list of possible pyramid frames, and formulate the complete algorithm.

Lemma 3.4 (Pyramid frame generation)**.** *Given a graph $G = (V, E)$ we can generate a list of all possible pyramid frames $b_1, b_2, b_3, s_1, s_2, s_3$ such that*

1. *the sets $\{b_i, s_i\}$ and $\{b_j, s_j\}$ $j \neq i$ are disjoint, and have only one edge, between b_i and b_j;*
2. *there exists a top-vertex a adjacent to all s_i's, and to at most one b_i. If a is adjacent to b_i then $b_i = s_i$.*

in time $O(n^7)$ using Algorithm 3.

Algorithm 3 Pyramid frame generation

```
1: for vertex b_1 ∈ V do
2:     Determine all pairs of neighbours of b_1 that are adjacent. And store the bases
   b_1, b_2, b_3 as a set B.
3:     for each base B ∈ B do
4:         for vertex a ∈ V\B and neighbours s_1, s_2, s_3 of a do
5:             Check if the sets {b_i, s_i} and {b_j, s_j} j ≠ i are disjoint.
6:             if there is an edge between b_i and s_j then
7:                 Check if b_j = s_j.
8:             end if
9:             Check if at most one b_i is adjacent to a, and in that case if b_i = s_i.
10:            if all above statements are true then
11:                b_1, b_2, b_3, s_1, s_2, s_3 is a possible pyramid frame.
12:            end if
13:        end for
14:    end for
15: end for
16: return all possible pyramid frames
```

Proof. The algorithm loops through all possibilities of $O(n^7)$ vertices, with a constant time to check if they form a pyramid base. □

Theorem 3.5 (Pyramid detection)**.** *Given a graph $G = (V, E)$ we can determine in time $O(n^9)$ whether the graph contains a pyramid using Algorithm 4*

Proof. Here we will only discuss running time of the algorithm. For the proof of correctness we refer to [6]. Generating pyramid frames using Algorithm 3 takes $O(n^7)$ time and results in $O(n^6)$ sets through which we loop in line 2. Determining all promising pyramid paths takes $O(n^2)$ time. Determining all good pairs takes $O(n^3)$ time. Finding the combinations of good pairs also takes $O(n^3)$ time. This is because for a pair m_i, m_j we can now check in $O(1)$ whether they are a good pair, which also leads to a time $O(1)$ for checking a triple. In total the algorithm therefore takes $O(n^9)$ time.

□

Algorithm 4 Pyramid detection

1: Generate all pyramid frames $b_1, b_2, b_3, s_1, s_2, s_3$ using Algorithm 3.
2: **for** each pyramid frame **do**
3: Determine the sets $\mathcal{P}_i$, $i = 1, 2, 3$ each consisting of $O(n)$ promising pyramid paths using Algorithm 1.
4: **for** $1 \leq i < j \leq 3$ **do**
5: Determine all good pairs m_i, m_j using Algorithm 2.
6: **end for**
7: **for** all combinations of m_1, m_2 and m_3 **do**
8: **if** all combinations of m_i and m_j $i \neq j$ are good pairs **then**
9: **return true**.
10: **end if**
11: **end for**
12: **end for**
13: **return false**

3.1.2 Jewels

The second type of forbidden sub-structure is a jewel. An example of a jewel can be seen in Figure 5. When a graph contains an induced jewel, this will again lead to an induced odd-hole. First let us introduce the exact definition of a jewel.

Definition 3.3 (Jewel)**.** A *jewel* is a graph containing of five vertices $v_1, ..., v_5$ and a path P such that:

1. There is a cycle $v_1, v_2, v_3, v_4, v_5, v_1$;
2. There is *no* edge between v_1 and v_3, v_1 and v_4, or v_2 and v_4;
3. There exists a path P between v_1 and v_4, of which no internal vertex is a neighbour of $\{v_2, v_3, v_5\}$.

From this definition it will immediately follow that every graph that contains an induced jewel, also contains an odd-hole.

Theorem 3.6 (Jewels contain odd-holes)**.** *Every jewel consisting of the vertices $v_1, ..., v_5$ and a path P that fulfills the conditions of Definition 3.3 contains an odd-hole.*

Proof. First check whether the path P is an induced path. Otherwise, there exists a subset of the path vertices that induce a shorter path with interior vertices adjacent to $\{v_2, v_3, v_5\}$, and we take this as path P. Now, if the path $P = v_1, p_1, ..., v_4$ has even length we see that $v_1, p_1, ..., v_4, v_5, v_1$ is an odd-hole. If path P is an odd-length path, $v_1, p_1, ..., v_4, v_3, v_2, v_1$ is an odd hole. □

Similar to pyramids we need to find a sub-routine to determine jewels. A trivial enumeration method would take $O(n^7)$. We can however do it in in $O(n^6)$ [6]. At this moment this does not seem important, because finding pyramids is the bottleneck of the algorithm already. However, as we will later see the time to find pyramids will be brought down to $O(n^6)$, resulting in both of these sub-algorithms having the same running time of $O(n^6)$. Now we turn to detecting jewels in a graph.

Theorem 3.7. *Given a graph $G = (V, E)$ we can determine in time $O(n^6)$ whether the graph contains an induced jewel using Algorithm 5.*

Algorithm 5 Jewel recognition

Generate all tuples (v_2, v_3, v_5) such that (v_2, v_3) is an edge.
for each such tuple (v_2, v_3, v_5) **do**
    Let $F = \{v | v$ is not a neighbour of $v_2, v_3, v_5\}$.
    Let $V_1 = \{v | v$ is a neighbour of v_2, v_5 and not of $v_3\}$.
    Let $V_4 = \{v | v$ is a neighbour of v_3, v_5 and not of $v_2\}$.
    for every component F' of F **do**
        for every $v_1 \in V_1$ **do**
            if v_1 has a neighbour in F' **then**
                Mark (v_1, F') as *good*.
            end if
        end for
        for every $v_4 \in V_4$ **do**
            if v_4 has a neighbour in F' **then**
                Mark (v_4, F') as *good*.
            end if
        end for
    end for
    for each combination of $v_1 \in V_1$, $v_4 \in V_4$ and F' a component of F **do**
        if v_1 and v_4 are *not* neighbours, but both (v_1, F') and (v_4, F') are good **then**
            return true.
        end if
    end for
end for
return false

Proof. Assume a graph G contains an induced jewel containing vertices v_2, v_3, v_5 such that (v_2, v_3) is an edge. Then path P has no vertex that is a neighbour of v_2, v_3, v_5, and thus P is a subset of a component F' of the set F. Clearly, the algorithm then finds this component F' and vertices v_1, v_4. Moreover, because we enumerate all possible tuples (v_2, v_3, v_5) the algorithm will correctly determine that the graph contains a jewel.

On the other hand assume that the algorithm outputs that a graph G contains a jewel. Then clearly $v_1, ..., v_5$ obey Definition 3.3. We only need to show that there exists a path $P = v_1, ..., v_4$ with no internal vertex that is a neighbour of v_2, v_3, v_5. Take the component F' that has a neighbour of v_1 and v_4 in it. Clearly, the shortest path P, between v_1 and v_4 in the graph induced by the vertices of $F' \cup \{v_1, v_2\}$ is an induced path of G. Moreover, P has no internal vertex that is a neighbour of v_2, v_3, v_5. Thus, $v_1, ..., v_5$ together with P form a jewel.

To see that the running-time is indeed as claimed note that we loop over $O(n^3)$ tuples. There are $O(n)$ components, and the sets V_1, V_4 contain $O(n)$ vertices. Checking if v_1 and v_4 are neighbours and both (v_1, F') and (v_4, F') are good takes $O(1)$ time, which shows that the algorithm runs in $O(n^6)$. □

3.1.3 Configurations of types T1, T2 and T3

In this section, three other easily detectable induced subgraphs that contain an odd-hole are introduced called T_1, T_2 and T_3. We will do these in order, and T_1 is the most obvious one.

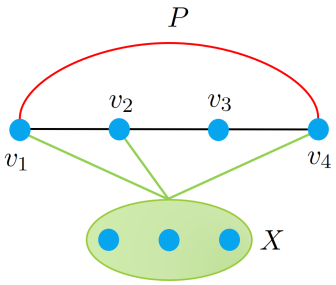

FIGURE 6: Type T_2 graph. In red path P, green oval represents set X. Lines from a vertex to a set means fully connected.

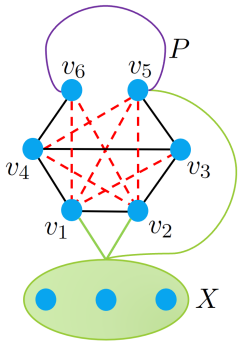

FIGURE 7: Type T_3 graph. In purple path P, green oval represents set X. Lines from a vertex to a set mean fully connected.

Definition 3.4 (Configuration T_1)**.** A configuration of type T_1 are five vertices $v_1, ..., v_5$ such that these vertices form an odd-hole.

Checking for these odd-holes can be done in the trivial way, and will also result in our graph not being a Berge graph. Therefore, we omit writing the algorithm down but still choose to show the result in a theorem below.

Theorem 3.8 (T_1 detection)**.** *Given a graph $G = (V, E)$ we can determine in time $O(n^5)$ whether G contains a T_1.*

Proof. We can simply enumerate all possible combinations of five vertices, name them $v_1, ...v_5$ and checking whether they form an odd-hole in this order. Checking for all 20 edges takes $O(1)$ time, which concludes the proof. □

Clearly, the algorithm above can be sped up slightly. When implementing we can for example generate combinations of vertices starting with all pairs of neighbours of a vertex v_1 and combine this with all pairs of non-neighbours. However, finding these odd-holes of length five is not a bottleneck of the algorithm anyhow.

We will introduce the other configurations of types T_2 and T_3 below, and give algorithms to find those and their running times. A proof that graphs containing T_2 and T_3 also contain odd-holes is omitted and we refer to the proof in [6].

Definition 3.5 (Configuration T_2)**.** A *configuration of type* T_2 is a graph consisting of 4 distinct vertices v_1, v_2, v_3, v_4 together with a set X and a path P such that:

1. The vertices v_1, v_2, v_3, v_4 induce a path.
2. Vertices v_1 and v_4 are the end-points of path P, and P has no internal vertex that is equal to or a neighbour of v_2 or v_3. Moreover, no internal vertex of P is X-complete.
3. The set X is an anti-component of all $\{v_1, v_2, v_4\}$-complete vertices.

Definition 3.6 (Configuration T_3)**.** A *configuration of type* T_3 is a graph consisting of 6 distinct vertices $v_1, ..., v_6$ together with a set X and a path P such that:

1. We have edges (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_1, v_4), (v_3, v_5) and (v_5, v_6).
2. No other edges between vertices $v_1, ..., v_6$ are allowed, except between v_3, v_6 and between v_5, v_6 which might be edges.

3. X is an anti-component of all $\{v_1, v_2, v_5\}$-complete vertices.

4. Vertices v_3 and v_4 are not X-complete

5. Path P has endpoints v_5 and v_6, has no internal vertex that is equal to v_1, v_2, v_3, v_4, no internal vertex that is a neighbour of v_1 or v_2 and no internal vertex that is X-complete

6. Vertex v_6 is not X-complete if there is an edge between v_5 and v_6.

An example of a type T_2 and type T_3 graph can be found in Figure 6 and 7 respectively. We will now shortly discuss algorithms to determine whether graph G contains vertices that induce a configuration of type T_2 or T_3.

Theorem 3.9 (T_2 detection)**.** *Given a graph $G = (V, E)$ we can determine in time $O(n^6)$ whether this graph contains a configuration of type T_2 using Algorithm 6.*

Algorithm 6 T_2 detection

```
Find all vertices v_1, v_2, v_3, v_4 that induce a path v_1, ..., v_4.
for each path v_1, ..., v_4 do
    Let 𝒳 be the set of all anti-components of {v_1, v_2, v_4}-complete vertices.
    for each X ∈ 𝒳 do
        Let S = {v ∈ V | v ∉ {v_2, v_3}, (v, v_2) ∉ E, (v, v_3) ∉ E, v is not X-complete}.
        if there exists a path P from v_1 to v_4 in the graph G' induced by S then
            return true.
        end if
    end for
end for
return false.
```

Proof. The algorithm is essentially brute-force and therefore correctness is trivial. There are at most $O(n^4)$ paths $v_1, ..., v_4$ and $O(n)$ sets X. Testing if *any* path P exists can for example be done using breadth-first search in time $O(n)$, which proves the running-time. □

Theorem 3.10 (T_3 recognition)**.** *Given a graph $G = (V, E)$ we can determine in time $O(n^6)$ whether this graph contains a configuration of type T_3 using Algorithm 7.*

Proof. Again the algorithm is simply a (smarter version of a) brute-force and trivially finds any configuration of type T_3 if it exists. There are $O(n^3)$ triples (v_1, v_2, v_5), $O(n)$ possible sets X, $O(n)$ possible vertices v_4. Checking if v_3, v_6 and P exist can be done in time $O(n)$, trivially for the vertices and again using for example breadth-first search to test the existence of a path P. □

The proof that every graph that contains an induced subgraph of configuration types T_2 or T_3 also contains an odd-hole is based on the Roussel-Rubio lemma [20], and an example proof using that lemma can be found in [6].

Algorithm 7 T_3 detection

Find all triples (v_1, v_2, v_5) such that $(v_1, v_2) \in E$, $(v_i, v_5) \notin E, i = 1, 2$.
for each triple (v_1, v_2, v_5) **do**
    Let $\mathcal{X}$ be the set consisting of anti-components of all $\{v_1, v_2, v_5\}$-complete vertices.
    for each $X \in \mathcal{X}$ **do**
        Let $F' = \{v \in V | (v, v_1) \notin E, (v, v_2) \notin E, v \text{ is not } X\text{-complete}\}$.
        Let F'' be the anti-component of $F' \cup \{v_5\}$ that contains v_5.
        Let M be all X-complete vertices not adjacent to v_1, v_2, v_5 neighbouring F''.
        Let $F = F'' \cup M$.
        Let $V_4 = \{v | v \text{ adjacent to } v_1, \text{ not to } v_2, v_5, \text{ with a neighbour in } F, \text{ not } X\text{-complete}\}$.
        for each $v_4 \in V_4$ **do**
            Find a vertex v_3 adjacent to v_2, v_4, v_5, not to v_1 that is not X-complete.
            Let v_6 be any neighbour of v_4 in F.
            Let G' be the graph induced by F'.
            Let P be any path from v_5 to v_6 in G'.
            if v_3, v_6 and P exist **then**
                return true
            end if
        end for
    end for
end for
return false

3.2 Cleaning algorithm

We now turn to finding odd-holes in our graph G. For this we first introduce a *clean* (odd-)hole. If we have any hole C we consider vertices neighbouring C. Such a neighbour can be the neighbour of multiple vertices in C, and we call this neighbour *C-major* if it has neighbours $v_1, v_2, v_3 \in C$ such that v_1, v_2 and v_3 do not lie on an induced path of length three. Such a C-major vertex can then be used for a shorter route between these vertices. If no vertex is C-major in a graph without pyramids and without jewels, holes are easily obtainable. In particular, this will allow us to find holes by finding the shortest path between three relatively evenly spaced vertices on C.

Definition 3.7 (Clean hole)**.** A *clean* hole is a hole such that there are no C-major vertices.

A clean hole then will have no vertices that allow a 'much shorter path' between vertices on the cycle. In general of course shorter paths could exist if two neighbours of the cycle are connected, however this will not be the case. We first need an important result. The proof of the following theorem is over five pages long and therefore we refer to [6] for the proof as well.

Theorem 3.11 (Theorem 4.1 in [6])**.** *Given a graph $G = (V, E)$ that does not contain an induced pyramid or induced jewel. Let C be a shortest clean odd-hole in G. Take two vertices $v_1, v_2 \in C$ that are not neighbours and let $L_1 \subseteq C$ and $L_2 \subseteq C$ be the two paths of the cycle joining the vertices where $|L_1| < |L_2|$. Then*

1. *L_1 is a shortest path between v_1 and v_2;*
2. *For every other shortest path P between v_1 and v_2, we have that $P \cup L_2$ also induces a clean shortest odd-hole.*

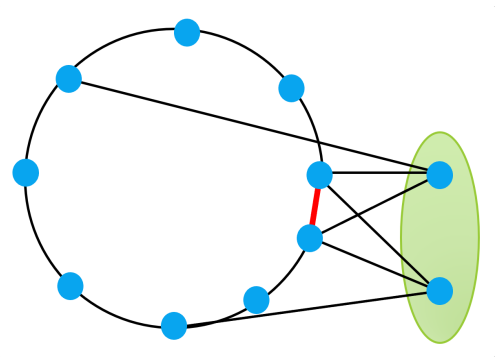

FIGURE 8: An amenable hole. In green an anti-component X of C-major vertices. The red edge indicates an X-complete edge.

From Theorem 3.11 it becomes clear that shortest clean odd-holes are easy to find. If we simply enumerate all tuples of three vertices v_1, v_2, v_3 and find shortest paths between them, if there exists a clean odd-hole we will pick three vertices on the shortest clean odd-hole eventually. By applying Theorem 3.11 multiple times, the shortest paths between these vertices then form such a clean odd-hole upon inspection.

In reality, not all shortest odd-holes are clean. Imagine there would be a way to enumerate polynomially many sets such that one set contains exactly all C-major vertices for some shortest odd-hole and no vertex of C. We call this set S a cleaner of hole C. If we would then delete each set one by one from G and test for a clean odd-hole we would clearly find C in the graph induced by $V \backslash S$ using Theorem 3.11. Unfortunately, generating sets such that one of them is a cleaner is not that easy. However, it is possible to generate sets that are almost as good.

Definition 3.8 (Near-cleaner)**.** Given a graph $G = (V, E)$ and a shortest odd-hole C, a set $S \subseteq V$ is a *near-cleaner* of C if it contains all C-major vertices, $|C \cap S| \leq 3$ and the vertices of $C \cap S$ induce a path.

In the remainder of this section we will then show how to obtain polynomially many, more precisely $O(n^5)$, sets such that one of them is a near-cleaner if the input graph contains an odd-hole and no forbidden substructures. In Section 3.3, we will show how we can find an odd-hole if we are provided with a near-cleaner.

Obtaining sets such that one of them is a near-cleaner for a shortest odd-hole C, is not possible for every shortest odd-hole C in general. We introduce a special type of odd-hole.

Definition 3.9 (Amenable hole)**.** Given a graph $G = (V, E)$ a hole $C \subseteq V$ is amenable if

1. C is a shortest odd-hole;
2. $|C| \geq 7$;
3. For every anti-connected set $X \subseteq V$ consisting of C-major vertices, there exists an X-complete edge on the cycle.

An example of an amenable hole can be seen in Figure 8. In another highly technical proof of over five pages long in [6], the following is shown:

Theorem 3.12 (Forbidden substructures lead to amenable holes - Theorem 8.1 in [6])**.** *Given a graph $G = (V, E)$, such that G and the complement graph $\bar{G}$ do not have a subset that induces a pyramid, jewel or configuration of type T_1, T_2, T_3, then every shortest odd-hole $C \subseteq V$ is amenable.*

Clearly, this is exactly what we wanted, and we now need to generate sets such that one of them is a near-cleaner for C if C is an amenable hole.

3.2.1 Generating near-cleaners

Firstly, note that if an amenable hole only has one anti-component X of C-major vertices, then by Definition 3.9 this entire component can be easily found by generating sets containing all neighbours of every pair of neighbouring vertices. One of those neighbouring sets then contains all C-major vertices, and no vertex of C, because C is a hole and two neighbouring vertices on a hole share no neighbours of C. However, when there are multiple anti-components there is no such easy solution. Still, however, if we manage to choose two vertices a and b on the cycle C the set of their common neighbours $N(a, b)$ consists of only C-major vertices. If we would manage to find all anti-components containing C-major vertices, except for one, we could combine this set with the sets of all neighbours of two neighbouring vertices and be guaranteed that one of those unions contains all C-major vertices.

Secondly, if we could pick two vertices a and b on the hole C that have distance greater than three on the cycle, then all their common neighbours are by definition C-major. Picking these vertices a, b in a smart way and extending the set of their neighbours with some other vertices will do the job. In order to do this, we need introduce relevant triples.

Definition 3.10 (Relevant triples and their accompanying sets)**.** A set of three vertices a, b, c is called a *relevant triple* if a, b are two distinct non-adjacent vertices and $c \notin N(a, b)$, however c might be in $\{a, b\}$. Moreover we also define the following sets and values regarding the anti-components of $N(a, b)$:

1. $r(a, b, c)$ is the size of the largest anti-component of $N(a, b)$ that contains at least one non-neighbour of c;
2. $Y(a, b, c) \subseteq N(a, b)$ is the union of all anti-components with size strictly greater than $r(a, b, c)$;
3. $W(a, b, c)$ is an anti-component of the graph induced by $N(a, b) \cup \{c\}$ that contains c;
4. $Z(a, b, c)$ is the set consisting of all vertices that are $Y(a, b, c) \cup W(a, b, c)$-complete;
5. $X(a, b, c) = Y(a, b, c) \cup Z(a, b, c)$.

All the sets from Definition 3.10 can be found in Figure 9. We will now work out this example to give a better understanding of what is going on. The two red vertices on the cycle are vertices a and b, the third red vertex on the right bottom is vertex c. Vertices in coloured vertical ovals are not connected because they are anti-components, while thick coloured lines between vertices and sets or sets themselves mean fully connected edges. The dashed red-line from vertex c to the purple oval means this edges is not present.

Therefore, this purple set is the largest anti-component that contains a non-neighbour of c, and $r(a, b, c) = 3$. The set $Y(a, b, c)$ is equal to the green anti-component as it is the only one that is larger than $r(a, b, c)$. Then $W(a, b, c)$, the anti-component containing c, is the purple anti-component together with c, which is also indicated with a dark green oval. Note that here $W(a, b, c)$ is the set with size $r(a, b, c)$ together with c because c is fully connected to all other anti-components, however this does not need to be the case in general. The set $Z(a, b, c)$ consist of all oranges vertices, which are the blue anti-component

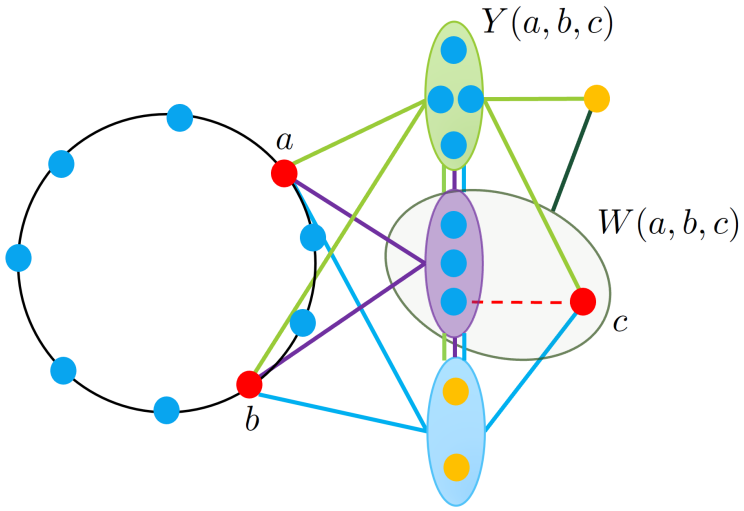

FIGURE 9: Example of a hole with a relevant triple (a, b, c) in red

and the orange vertex next to $Y(a, b, c)$. Lastly, $X(a, b, c)$ is then the union of $Y(a, b, c)$ and $Z(a, b, c)$, which contains both the green and the blue anti-component, and the orange vertex.

We can already see in this example there is exactly one anti-component Q of C-major vertices not in $X(a, b, c)$, which we will be able to determine because there must exist an Q-complete edge on C. As we will now see this is always the case for this set $X(a, b, c)$. For this we first need the following lemma.

Lemma 3.13 (Lemma 9.1 in [6]). *Given a graph $G = (V, E)$ with an amenable hole C, there exist a relevant triple (a, b, c) such that:*

1. *The set of all C-major vertices that are not present in $X(a, b, c)$ form an anti-connected set;*

2. *The intersection of $X(a, b, c)$ and the vertices of C induce a path of length at most three.*

Clearly now, if we have a graph that has an amenable hole we can generate all possible sets $X(a, b, c)$ from relevant triples. Then we will have found a set $X(a, b, c)$ which covers all except at most one anti-component of C-major vertices. By Definition 3.9 of an amenable hole there then exist two vertices $u, v \in C$ such that the neighbours $N(u, v)$ of these vertices contain the last anti-component of C-major vertices. The union of these two sets then form a near-cleaner. Because we do not know the actual hole, the idea is just to generate all possible sets $X(a, b, c)$ and $N(u, v)$ and their combinations. The following theorem formally completes the suggestions above. Note that the algorithm does not go into detail on how to obtain sets Y, Z, W, X as this is trivially done by checking neighbours in the graph G and its component $\bar{G}$.

Theorem 3.14 (Near-cleaners). *Given a graph $G = (V, E)$ that contains an amenable odd-hole C, we can generate $O(n^5)$ sets such that one of those sets is a near-cleaner in time $O(n^5)$ using Algorithm 8.*

Algorithm 8 Near-cleaners

1: Find for all pairs of neighbouring vertices $u, v \in V$ the set of common neighbours $N(u, v)$.
2: Generate all relevant triples (a, b, c).
3: Find the sets $Y(a, b, c), W(a, b, c), Z(a, b, c), X(a, b, c)$ for each triple (a, b, c).
4: **return** all combinations of $N(u, v)$ and $X(a, b, c)$.

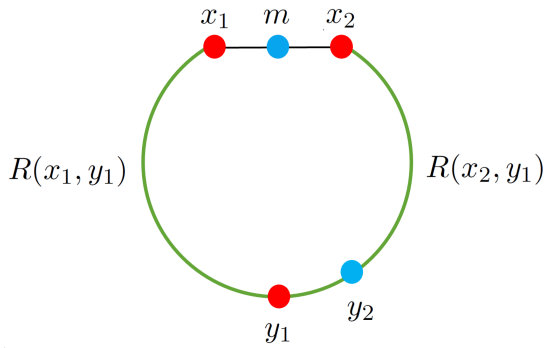

FIGURE 10: Odd hole constructed from an induced path (x_1, m, x_2) and two shortest paths to vertex y_1.

Proof. First of all clearly the algorithm outputs $O(n^5)$ sets and runs in time $O(n^5)$. Assume G contains an amenable hole C. Then by Lemma 3.13, there exists (a, b, c) such that $X(a, b, c)$ covers all but at most one anti-component, say Q, of the set of C-major vertices. By Definition 3.9, there are $u, v \in C$ such that $Q \subseteq N(u, v)$. Clearly $N(u, v) \cap C = \emptyset$, and by Lemma 3.13 $X(a, b, c) \cap C$ is an induced path of length of at most 3. Therefore $N(u, v) \cup X(a, b, c)$ contains all C-major vertices, and intersects with C in at most 3 vertices on a path. Therefore $N(u, v) \cup X(a, b, c)$ is a near-cleaner and it is found by Algorithm 8. □

3.3 Odd hole detection

From the sections above we know that we can first search any graph G for forbidden substructures to guarantee that if it contains an odd-hole it contains an amenable odd-hole. Moreover, we can then generate $O(n^5)$ sets such that one of them contains a near-cleaner if such an amenable hole exists. The only thing that remains is an algorithm to find an odd-hole in a graph G given a near-cleaner $X \subseteq V$. Obviously, we could just generate every combination of maximal three vertices from the near-cleaner, remove the rest from the graph and we are guaranteed to be left with a clean hole which we know how to find. However, this would be very slow and there is a better method. This method relies on first finding paths between all vertices with internal vertices outside of X, and afterwards 'repairing' the hole C with a 3-vertex path of vertices of the original graph. The beauty of the algorithm lies in the fact that it finds two vertices that are exactly on opposite sides of the cycle of the three vertex subset that is the intersection between the near-cleaner and C. Because these vertices are as far apart as possible, we can say a lot about the length of the respective paths and the non-existence of other paths.

Theorem 3.15 (Detect amenable hole from near-cleaner)**.** *Given a graph $G = (V, E)$ and a set $X \subseteq V$ that is a possible near-cleaner, we can decide whether the graph G contains an amenable hole with near-cleaner X using Algorithm 9 in time $O(n^4)$.*

Algorithm 9 Detect amenable hole from near-cleaner

Let $G' = (V', E')$ be the graph induced by $V \backslash X$.
Find all shortest paths $R(x, y)$ between vertices $x, y \in V$ with internal vertices in V'.
Let $r(x, y)$ be equal to the number of vertices in $R(x, y)$ or ∞ if the path does not exist.
Find all possible triples $(x_1, m, x_2) \subseteq V$ that induce a path x_1, m, x_2 in G.
for each vertex $y_1 \in V'$ **do**
    for each triple (x_1, m, x_2) **do**
        Let y_2 be the neighbour of y_1 on the path $R(x_2, y_1)$.
        Check if $R(x_1, y_1)$ and $R(x_2, y_1)$ both exist.
        Check if $r(x_2, y_1) = r(x_1, y_1) + 1 = r(x_1, y_2)(= K)$.
        Check if $r(m, y_1), r(m, y_2) \geq K$.
        if everything above is true **then**
            return true
        end if
    end for
end for
return false

Proof. We will show that the algorithm outputs true if and only if there exists an amenable hole with near-cleaner X. We found that in this proof it is easier to talk about the number of vertices in a path, and not its length. While we also omit the word length, this remark is made as not to confuse the reader.

(Suffiency)

We first show that if the algorithm outputs true, that then there does exist an odd-hole. Consider the paths $R(x_1, y_1)$ and $R(x_2, y_1)$ as also shown in Figure 10. We will show that these paths either form an odd-hole, or there is a smaller odd-hole using parts of these paths.

First note that because $r(x_2, y_1) = r(x_1, y_1) + 1$ the vertex $x_2 \notin R(x_1, y_1)$. Moreover, because x_1 and x_2 are non-neighbours and from the same argument $x_1 \notin R(x_2, y_1)$, as otherwise $r(x_2, y_1) = r(x_1, y_1) + 2$, a contradiction. Similarly, we can deduce that $y_2 \notin R(x_1, y_1)$ and vertex m is a part of neither path. Moreover, m has no neighbours on $R(x_1, y_1)$ and $R(x_2, y_1)$ except x_1 and x_2, because otherwise either $r(m, y_1)$ or $r(m, y_2)$ would be smaller than K.

Secondly, we show that the two paths $R(x_1, y_1)$ and $R(x_2, y_1)$ have no other common vertex than y_1. From the output of the algorithm we know that $r(x_2, y_1) = r(x_1, y_1) + 1$. Let us write for the paths $R(x_1, y_1) = v_1, v_2, v_3, ..., v_{K-1}$ and $R(x_2, y_1) = w_1, w_2, w_3, ..., w_K$, where $v_{K-1} = w_K = y_1$ and $w_{K-1} = y_2$. Assume to the contrary that the two paths do have a vertex in common, say $v_i = w_j$. Then, because v_i and w_j are both part of a shortest path to y_1, we know that the sub-paths $v_i, v_{i+1}, ..., y_1$ and $w_j, w_{j+1}, ..., y_1$ must have the same length. Consequently, $j = i + 1$ and vertex $v_i = w_{i+1}$. Then, $P = v_1, ..., v_i, w_{i+2}, ...w_{K-1}$ is a path from x_1, to y_2 containing $K - 2$ vertices. However, in the algorithm it was determined that $r(x_1, y_2) = K$. This is a contradiction and therefore the paths $R(x_1, y_1)$ and $R(x_2, y_1)$ intersect only at vertex y_1.

Note now that if there are also no edges between any vertices from $R(x_1, y_1)$ and $R(x_2, y_1)$ we find that the vertices of $R(x_1, y_1) \cup R(x_2, y_1) \cup \{m\}$ induce a hole of length $(K - 2) + (K - 1) + 2 = 2K - 1$, and thus the graph indeed contains an odd-hole and the

algorithm correctly outputs true.

Assume now that there does exist an edge between $R(x_1, y_1)$ and $R(x_2, y_1)$, namely between vertices v_i and w_j. Suppose $i < j$. Then there is a path $P = v_1, ..., v_i, w_j, ..., w_{K-1}$ between x_1 and y_2 and $|P| \leq K - 1$. However, all internal vertices of this path are in V' and thus $r(x_1, y_2) \leq K - 1$, a contradiction. On the other hand, suppose $i > j$. Then there is a path $P = w_1, ..., w_j, v_i, ..., v_{K-1}$ from x_2 to y_1. Again, all internal vertices of P are in V' and $|P| \leq K - 1$. Another contradiction with the fact that $r(x_2, y_1) = K$. We conclude that $i = j$. Take the lowest i such that there is an edge between v_i and w_i. Then $C = x_1, m, x_2, w_2, w_3, ..., w_i, v_i, v_{i-1}, ..., v_3, v_2, x_1$ is a cycle of length $2i + 1$ without any other edges between its vertices. Thus C is an odd-hole and the output of the algorithm was correct.

(Necessity)

Now we need to show that if X is a near-cleaner for a shortest odd hole C of the graph, then the algorithm will indeed output true. Let us again refer to Figure 10 and assume that C is a shortest odd-hole of length $2K - 1$. Because the algorithm tries all combinations of triples (x_1, m, x_2) and vertices y_1, we can freely pick a combination of these vertices that is considered during the algorithm. Firstly, we ensure that all four vertices x_1, m, x_2, y_1 lie on C. Because X is a near-cleaner for an amenable hole, we know that $C \cap X$ is the subset of a three vertex path, and we pick (x_1, m, x_2) such that $C \cap X \subseteq \{x_1, m, x_2\}$. Lastly, we pick the vertex y_1 such that the sub-path P_1 of the cycle from x_1 to y_1 that does not pass x_2 has $K - 1$ vertices, and the sub-path P_2 from x_2 to y_1 has K vertices.

For this choice of (x_1, m, x_2) and y_1 consider $R(x_1, y_1)$ that is computed in the algorithm. Because $C \cap X \subseteq \{x_1, m, x_2\}$ we know that none of the internal vertices of P_1 lies in X. Thus, $r(x_1, y_1) \leq |P_1|$, as $r(x_1, y_1)$ is the length of the shortest path without internal vertices in X and P_1 is one such path. Let us construct a graph $G'' = (V'', E'')$ that is induced by the vertices $V'' = (V \backslash X) \cup \{x_1, m, x_2\}$. We know that by definition of a near-cleaner, the hole C is a clean odd-hole in graph G''. By assumption C was a shortest odd-hole and therefore by Theorem 3.11, P_1 is a shortest path in G'' between x_1 and y_1. Note that the path $R(x_1, y_1)$ is also a path of G'', as by definition its internal vertices are not a part of X. Therefore $r(x_1, y_1) \geq |P_1|$.

Because we showed that both $r(x_1, y_1) \geq |P_1|$ and $r(x_1, y_1) \leq |P_1|$, we conclude $r(x_1, y_1) = |P_1| = K - 1$. Again by Theorem 3.11, we can replace path P_1 by path $R(x_1, y_1)$ in C, and C is still a shortest clean odd-hole in G''. By the exact same argument and Theorem 3.11 we know that $r(x_2, y_1) = |P_2| = K$, because P_2 is a shortest path from x_2 to y_1. Because both P_1 and P_2 are shortest paths, we also know that shortest paths from m to y_1 or y_2 through x_1 and x_2 respectively contain K vertices. If $x_i \in X$ for $i = 1, 2$ the path from m to y_i might be longer or does not even exist. In all cases however $r(m, y_1), r(m, y_2) \geq K$, as for non-existing paths $r(u, v)$ is infinite.

For this choice of (x_1, m, x_2) and y_1 the algorithm will output true, and because the algorithm tries all combinations it will output true for every set X such that X is a near-cleaner for a shortest odd hole C of the graph.

(Running time)

Now we show that the running time is as claimed. Finding all pairs of shortest paths is done in $O(n^3)$ [15]. We generate the set of triples inducing a path in time $O(n^4)$ at most. We then loop through $O(n)$ vertices and $O(n^3)$ triples, while all checks inside the loop take constant time. □

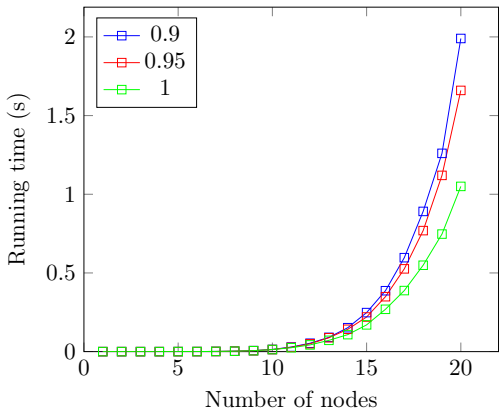

FIGURE 11: Average running time of a pyramid detection sub-algorithm for varying edge probabilities.

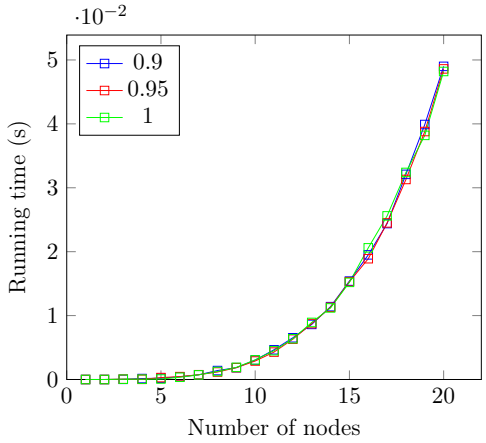

FIGURE 12: Average running time of a jewel detection sub-algorithm for varying edge probabilities.

3.4 Perfect graph detection

In the sections above a complete algorithm is given to detect perfect graphs. We first determine for a given graph G in time $O(n^9)$ whether it contains one of the five forbidden substructures. Afterwards we know that if G contains a hole C, then it contains an amenable hole C'. Therefore, we generate $O(n^5)$ sets of vertices, such that if G contains an amenable hole, one of these sets is a near-cleaner for it. For each of these sets we try to find and repair an amenable hole in time $O(n^4)$. In total this means that both the part of the algorithm where forbidden substructures are detected, as well as the remainder of the algorithm run in time $O(n^9)$. Notably, the only forbidden substructure with this running time are pyramids, while jewels and configurations of types T_1, T_2, T_3 are all found in $O(n^6)$. Of course, in practice it is beneficial to look for forbidden substructures in a different order than in which they were presented in this thesis. Pyramids are always considered as last of the five.

We implemented the entire algorithm in Python 3.8, to not only determine the theoretical bottleneck with the highest asymptotic running time, but also the bottleneck in running time that arises for real problem inputs. Here we discuss only the running times of all sub-components of our Berge detection algorithm, to motivate our choice to find a better pyramid algorithm. All other interesting computational results are discussed in Section 6.

In Figures 12, 11, 13, 14, 15 and 16, we can see the average running times for all sub-components of our Berge detection algorithm. The simulations are done for an increasing number of nodes and 25 graphs for each number of nodes. These random graphs are generated with a certain probability for each edge, known as a Erdös-Rènyi graph. The probabilities for edges here are $p = 0.9, 0.95, 1.0$. These probabilities seem relatively high, however graphs with probabilities that are a little bit lower are almost never perfect and therefore less interesting. More about this will follow in Section 6. Also note that choosing very low probabilities on the other hand, has the exact same effect as very high probabilities because the Berge algorithm runs on the graph as well as its complement.

In the figures we only take the average of completed runs for each forbidden substructure, or each sub-routine. While in general the algorithm might already stop quickly in

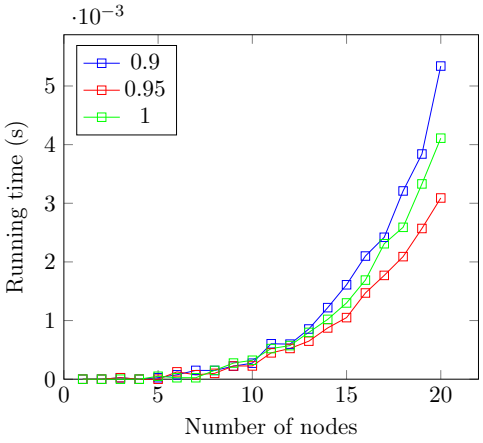

FIGURE 13: Average running time of a configuration type T_1 detection sub-algorithm for varying edge probabilities.

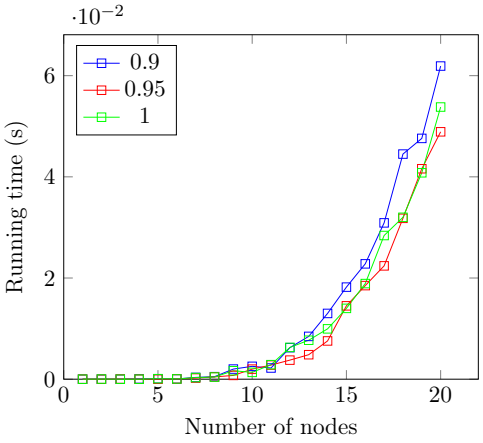

FIGURE 14: Average running time of a configuration type T_2 detection sub-algorithm for varying edge probabilities.

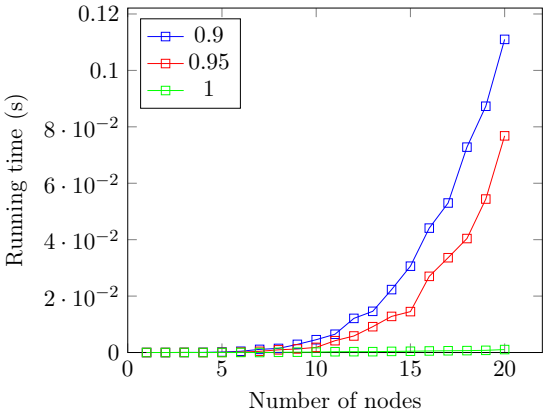

FIGURE 15: Average running time of a configuration type T_3 detection sub-algorithm for varying edge probabilities.

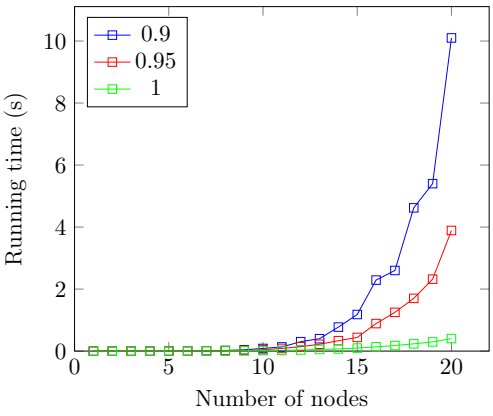

FIGURE 16: Average running time of near-cleaner and odd hole detection routines for varying edge probabilities.

many cases with low edge probabilities, because it finds a configuration of type T_1, we here consider the worst-case scenarios for our algorithm, to find bottleneck sub-routines. Therefore, for each sub-algorithm we only average over the runs in which it has fully been executed.

Interestingly, the sub-algorithms testing for configurations of type T_3 and the algorithm for detecting amenable odd-holes from near-cleaners perform much worse for lower edge probabilities, while the other sub-algorithms are almost unaffected. This comes from the fact that in both cases a lot of outer loops of the algorithms depend on some set of vertices and at least one non-edge between them.

While clearly for lower edge probabilities the bottleneck of the Berge detection algorithm lies in routines 1 and 3, for highly dense graphs the pyramid detection algorithm is the bottleneck. For graphs that are not very dense, often a substructure of types T_1, T_2, T_3 or a jewel can be found without the need to run the pyramid detection and the algorithm to clean and detect amenable holes. Therefore, highly dense graphs are particularly important and hard to decide leading us to look for a better alternative to detect pyramids.

4 Three-in-a-tree

The next half of this thesis is focused on finding an algorithm that brings down the running-time to detect pyramids to $O(n^6)$, which makes it a tied theoretical bottleneck of the forbidden substructures part of the algorithm. In this section we start by introducing the three-in-a-tree problem. As will be seen later this problem relates very closely to finding pyramids in a graph.

4.1 Problem description

For the three-in-a-tree problem we are given a simple, undirected graph G, together with three vertices that we will refer to as *terminals*. The three-in-a-tree problem then asks whether there exists an induced tree on the graph G that connects all the terminals. More formally:

Definition 4.1. (Three-in-a-tree) Given a simple, undirected graph $G = (V, E)$ and three different vertices v_1, v_2, v_3 called *terminals*. We say that the graph together with the terminals allows *three-in-a-tree* if there exists $W \subseteq V$, such that v_1, v_2, $v_3 \in W$, and the subgraph *induced* by W is a tree.

If our graph contains multiple components, either the terminals lie in different components, or we can solve the three-in-a-tree problem in a smaller graph that is the shared component of the terminals. Therefore, we assume from here on that our graph G is connected and every vertex has at least a degree of 1. We refer to a vertex with degree exactly 1 as a *leaf* of the graph. From here on we will also assume without loss of generality that the given terminals are leaves of the original graph G. Clearly, if they are not leaves we can modify the graph by adding an extra leaf vertex to each terminal. Because all terminals are leaves, for every vertex-minimal induced tree there is exactly one vertex $m \in V$ with degree $d(m) = 3$. We will refer to this vertex v_c as the *center vertex of the tree*.

Lemma 4.1 (Center vertex)**.** *Every vertex-minimal induced tree connecting the three terminals of the three-in-a-tree problem contains exactly one center-vertex of degree three. All other vertices that are not terminals have degree two.*

Proof. Clearly, all vertices that are not terminals have degree at least two, because of the minimality of the tree.

Now, first assume that there exists a vertex-minimal induced tree where all vertices have degree at most two. Clearly this tree is then a path and one of the terminal vertices is not a leaf in this tree. This is a contradiction with the terminal being a leaf of the graph. From this we conclude there exists at least one vertex with degree at least three.

Assume now that there exists a vertex with degree at least four. Because a tree is cycle-free, this means that the tree has at least four leaves. This is in contradiction with the minimality of the induced tree.

Lastly, assume that there exist more than one vertex with degree three. The tree will then contain at least four leaves, which again contradicts the minimality of the induced tree.

□

The next sections will outline the characterisation of the vertices of the graph into a *web* as described in [18].

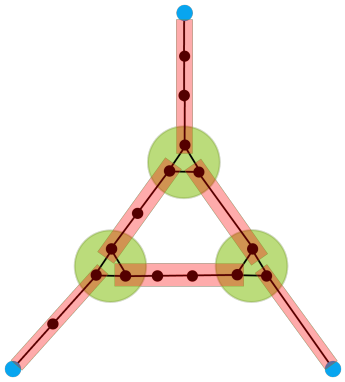

FIGURE 17: Example of a simple web, where green circles represent nodes and red rectangles are simple arcs.

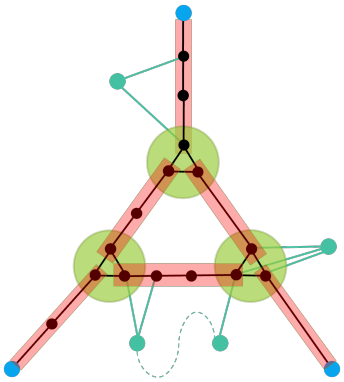

FIGURE 18: A web with $\mathbb{H}$-tamed sets of vertices on its outside in light green.

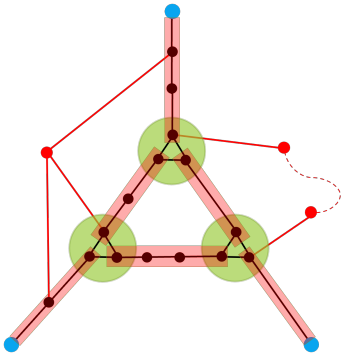

FIGURE 19: A web with $\mathbb{H}$-wild sets of vertices on its outside in red.

4.2 Graph webs

An important observation to decide whether we can find three-in-a-tree, is that the inducing vertex set of the tree can never contain three vertices from the same clique of the original graph. Because, if we would have three such vertices in the inducing vertex set, the tree would contain a cycle. This idea leads to the characterisation of a subset of the vertices into *nodes* and *arcs* which together form a *web*.

The nodes and arcs in the web behave similar as vertices and edges would in a graph; Each arc connects two nodes, which lie at the end of the arc. We stress that for the web, however, multiple arcs between two neighbours are allowed. Remember that we have a simple undirected graph $G = (V, E)$. We will first define the nodes, and a subclass of the arcs called *simple arcs*. For now we will focus on these simple arcs, but later we will define another type of arcs called *flexible arcs*.

Definition 4.2 (Node)**.** A *node* of a web consists of vertices $N \subseteq V$ such that for every two vertices $v_1, v_2 \in N$ that do not lie in the same arc we have $(v_1, v_2) \in E$.

Definition 4.3 (Simple Arc)**.** A *simple arc* of a web consist of vertices $A \subseteq V$ such that these vertices together form a minimal induced path between two nodes.

We write for an arc $A = UV$, meaning the arc has *end-nodes* U and V. For simple arcs this means that the arc A shares exactly one vertex with U and V. By now we are ready to define a web.

Definition 4.4 (Web)**.** A *web* $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ over a graph G and three terminals consists of *arcs* $A \in \mathcal{A}$, and *nodes* $N \in \mathcal{N}$, where $A, N \subseteq V$. The set X contains all vertices of the web, i.e. $X = (\bigcup_{A \in \mathcal{A}} A) \cup (\bigcup_{N \in \mathcal{N}} N)$. Every vertex of X is part of exactly one arc, and a maximum of two nodes. Moreover, the set X is connected and contains all terminals.

We look at an example of a web consisting of simple arcs and nodes as seen in Figure 17. This web consist of three nodes that are coloured green, three leaf nodes in blue, and six simple arcs coloured in red. The vertices in this web can never form a tree together connecting all three terminals. Importantly, this is always the case for every web.

Lemma 4.2 (Web contains no tree - simple web version)**.** *There exists no subset of vertices of a web that induce a tree connecting all three terminals.*

Proof. From Lemma 4.1 we know that every induced tree, containing vertices T, connecting the terminals should contain a center vertex v_c. This vertex has degree three, and can therefore not be part of a simple arc. Therefore, v_c is part of a node N. Clearly then, v_c has at most one of its neighbours of T in its arc A, and must have two neighbours in N. However, by definition all vertices in N form a complete subgraph and the graph induced by T contains a triangle. □

The observation that the vertices in a web can never create an induced tree connecting the three terminals, leads us to a clear algorithmic idea.

We want to start with some initial web, which is then grown while we maintain the fact that no three-in-a-tree exists. During each step of the algorithm we then either find a three-in-a-tree or try to increase the size of the web. Then, when neither is possible we are able to conclude that no three-in-a-tree exists. In the next section we will describe different kinds of vertices outside of the web. These classifications help to conclude whether no three-in-a-tree exists, or we can grow the web.

4.3 Tamed sets

As described in the section above, the graph that is induced from all vertices in a web can never contain an induced tree that connects the terminals. We will now consider the vertices that are not yet present in the web. We introduce the following definitions that will help distinguish those.

Definition 4.5 (Tamed set)**.** Take a web $\mathbb{H}$ over a graph G. We say a set $S \subseteq X$ of vertices in the web is *tamed* if every pair of vertices in the set share a common node or arc.

Note that from the definition above, it is not necessary that all vertices in S share a common arc or node. In our web using simple arcs, there is only one such case that can appear. Take three vertices v_1, v_2, v_3, which are all part of a different node N_i, $i \in \{1, 2, 3\}$, and arcs $A_i = \{v_j | j \neq i\}$. Clearly, every pair of vertices share an arc, but there is no common arc or node. We will refer to this special structure as a *triad* if and only if the graph induced by $\cup_{i \neq j}(N_i \cap N_j)$ induces a triangle, or more formally:

Definition 4.6 (Triad)**.** A *triad* is a sub-structure of a web consisting of vertices that lie in three nodes N_1, N_2, N_3, such that every arc $A = N_i N_j$ contains exactly all vertices in $N_i \cap N_j$. Moreover, if A_1, A_2, A_3 are the three arcs between these nodes, then for all $v_1 \in A_1$, $v_2 \in A_2$ and v_3, A_3 the graph induced by v_1, v_2, v_3 is a triangle graph.

An example of such a triad is seen in Figure 20. As argued above, in every tamed set all vertices either share one arc, one node or are a subset of a triad. We extend this definition to find $\mathbb{H}$-tamed sets of vertices outside the web.

Definition 4.7 (Tamed web)**.** A vertex or a set of vertices $Y \subseteq V \backslash X$ is called $\mathbb{H}$-*tamed* if the set of its neighbours in the web is tamed. Moreover, if in a graph every induced path of vertices outside the web is $\mathbb{H}$-tamed, we say that the underlying graph is $\mathbb{H}$-*tamed*. We refer to specifically an $\mathbb{H}$-untamed induced path of vertices $Y \in V \backslash X$ as $\mathbb{H}$-*wild*.

If Figure 18 and Figure 19 some examples of $\mathbb{H}$-tamed and $\mathbb{H}$-wild sets are shown. Importantly, when every connected component of vertices outside the web is $\mathbb{H}$-tamed, we can conclude something about the existence of trees connecting the terminals inside the web.

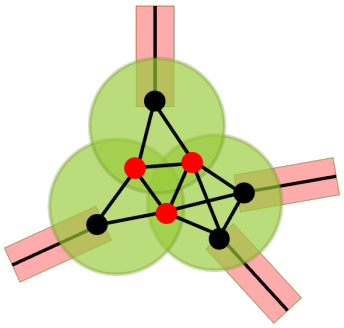

FIGURE 20: Part of a web containing three red vertices that form a triad

Theorem 4.3 (Tamed web - simple web version)**.** *Take a simple undirected graph $G = (V, E)$, three terminals and a web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ over G. If the graph G is $\mathbb{H}$-tamed, the graph G with the given terminals does not allow a solution to the three-in-a-tree problem.*

Proof. Remember that every valid vertex-minimal tree connecting the three terminals is cycle-free and contains a center-vertex according to Lemma (4.1). We will show that for a graph G that is $\mathbb{H}$-tamed, those statements can never be jointly true.

Assume that there exists a valid tree connecting the terminals and that center-vertex v_c is part of the vertices of our web, i.e. $v_c \in X$. The center-vertex must have at least one neighbour y outside of the web that is also part of this tree, as in a simple arc it can only have two neighbours, and by the definition of a node it can have only a single neighbour in a node without forming a triangle. We call the connected component that y lies in Y. We consider different cases for the neighbourhood $N_X(Y)$ of the set $Y \subseteq V \backslash X$. Note that by choice of Y, always $v_c \in N_X(Y)$. We consider three cases, where the tamed set $N_X(Y)$ consists of vertices ...

- ... with a common node N. Then within the tree v_c can have at most one neighbour in N, and must surely have a neighbour in Y. Because all non-terminal vertices have degree 2 in the tree, there is a finite number of vertices in Y before returning to a vertex in $N_X(Y)$. This vertex is either v_c or one of its neighbours and we found a cycle.

- ... with a common arc A. There are three disjoint paths from v_c to all terminals. Because the terminals lie neither in Y, and at most one is an end-vertex of A, those paths must all contain one of the end-vertices of the arc path. Because there are only two end-vertices of the arc, the tree contains a cycle.

- ... that form a triad. By assumptions, there exists a path from y to one of the terminals that does not contain v_c. This path must then contain another vertex from the triad, which is also a neighbour of v_c. We again find a cycle which yields a contradiction.

The proof that v_c can not lie in one of the sets $Y \in V \backslash X$ is very similar to the cases above and therefore omitted.

□

4.4 Aiding web

In the previous section it was shown that a graph G with three given terminals, and a web $\mathbb{H}$ such that G is $\mathbb{H}$-tamed, can never contain an induced tree connecting the terminals.

The converse is however not true. Sets outside the web that are wild might or might not allow three-in-a-tree. In this section we introduce the aiding web, which will allow us to find stronger wild sets.

Definition 4.8 (Split-component)**.** Given a graph G with a web $\mathbb{H}$ we call $\mathbb{G}$ a *split-component* of the web if $\mathbb{G}$ is either an arc (U, V), or $\mathbb{G}$ is the maximal subgraph such that the set $\{U, V\}$ forms a cut-set for the graph induced by X, but U and V are not adjacent in $\mathbb{G}$ and do not form a cut-set in the subgraph $\mathbb{G}$. We also refer to $\{U, V\}$ as the *split-pair* of $\mathbb{G}$ for $\mathbb{H}$.

Definition 4.9 (Chunks)**.** Given a graph $G = (V, E)$ with a web $\mathbb{H}$ we call $C \subseteq V$ a chunk of $\mathbb{H}$ if it contains the vertices of one or more split-components that share a common split-pair. We also write that C is the (U, V)-chunk of $\mathbb{H}$. A *maximal chunk* is a chunk that is not contained by any other chunk.

It should be noted that a chunk can contain at most one of the terminals of the three-in-a-tree problem, which then lies in one of the split-nodes of the chunk. We now observe that a wild set of a web $\mathbb{H}$, that is $Y \in V \backslash X$, which neighbours $N_X(Y)$ all lie in a single chunk behaves like a tamed set. In particular the following theorem holds:

Theorem 4.4. *Given a graph $G = (V, E)$ together with three terminals and a web $\mathbb{H}$. If all sets $Y \in V \backslash X$ are tamed or $N_X(Y) \subseteq C$ for some chunk C of $\mathbb{H}$, then there exists no induced tree connecting the three terminals.*

Proof. Assume to the contrary that this graph has an induced tree connecting the terminals. By Theorem 4.3, we only need to consider the sets $Y \in V \backslash X$ where $N_X(Y) \subseteq C$ for some chunk $C = (U, W)$ of $\mathbb{H}$ and $v_c \in C \cup Y$. Because v_c is the center-vertex, there exist three disjoint paths from v_c to the terminals. If the chunk contains no terminals, the three paths all contain a vertex in one of the split-nodes of the arc. At least two of those share a node and are either the same vertex, or neighbours from the definition of a node in Definition 4.2, meaning the tree contains a cycle. If on the other hand the chunk does contain a terminal, then this terminal lies in a split-node of the chunk, which is w.l.o.g. U. Moreover, there exist two disjoint paths P_1, P_2 from v_c to the other terminals, both containing a vertex in W. These two vertices are either the same vertex, or they lie in different arcs and are neighbours from the definition of a node. This means the tree contains a cycle, which is again a contradiction. We conclude that the graph contains no induced tree connecting the three terminals. □

We want to use the property observed above, to create a smaller web with fewer wild sets. Therefore, we use the operation MERGE.

Definition 4.10 (Merge)**.** Given a three-in-a-tree problem with graph $G = (V, E)$ and terminals $T \subseteq V$ and web $\mathbb{H}$, the operation MERGE(C) defined on a chunk $C = UV$ consisting of all vertices in arcs A_i, $i = 1, 2, ..K$ does the following:

1. Delete all nodes of vertices in the chunk that are not the split-nodes U and V of the chunk;
2. Delete all arcs A_i, $i = 1, 2, ..K$ in the chunk;
3. Create one new arc $A = UV$ with all vertices in C and end-nodes U and V.

We are now ready to define the aiding web.

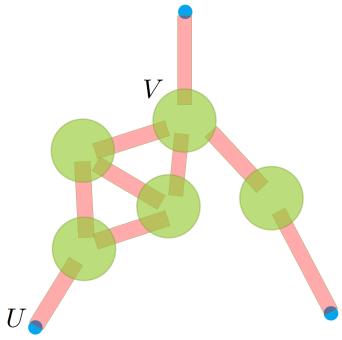

FIGURE 21: A web $\mathbb{H}$, consisting of green nodes and red arcs. Nodes U and V form a split-pair.

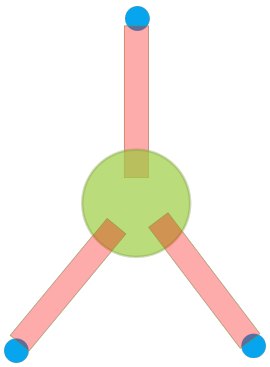

FIGURE 22: An aiding web $\mathbb{H}^\dagger$, for the web in Figure 21

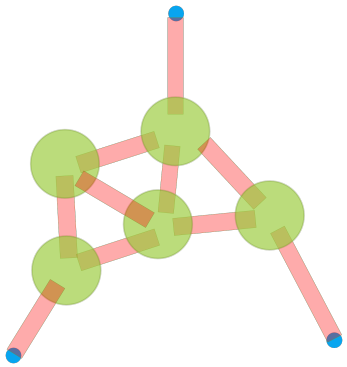

FIGURE 23: A web that is self-aiding, in other words $\mathbb{H} = \mathbb{H}^\dagger$

Definition 4.11 (Aiding web)**.** An *aiding web* $\mathbb{H}^\dagger$ for a graph G with web $\mathbb{H}$ is a web where every maximal chunk with split-nodes $\{U, W\}$ is merged to become an arc $A = (U, W)$ where the vertex set of the arc is $A = C$. We also refer to sets that are wild in the aiding web as $\mathbb{H}^\dagger$-wild.

In Figure 21 a web can be seen with its aiding web in Figure 22, the split-nodes U and V are merged into a single arc in the aiding web. The web shown in Figure 23 is self-aiding, meaning that $\mathbb{H} = \mathbb{H}^\dagger$.

4.5 Flexible arcs

We now introduce the concept of *flexible arcs*. Contrary to the simple arcs, these do not consist of an induced path of vertices, but they do still connect their two *end-nodes*. The flexible arcs can consist of vertices from multiple non-disjoint paths connecting the nodes. To introduce flexible arcs, we first need the concept of sprouts. These sprouts will then give us important properties of the arc, that will guarantee the existence of a three-in-a-tree in many cases. There are three types of sprouts defined as follows:

Definition 4.12 (Sprouts)**.** Given a graph G, its web $\mathbb{H}$ two of the nodes of the web N_1, N_2 and a set $S \subseteq V$. Then an (S, N_1, N_2)-*sprout* is an induced subgraph $\mathcal{S}$, such that one of three following conditions holds:

1. The induced subgraph $\mathcal{S}$ is a tree that intersects each of S, N_1 and N_2 at exactly one vertex;
2. The induced subgraph $\mathcal{S}$ consists of an induced path between S and N_1, and a disjoint induced path between S and N_2;
3. The induced subgraph $\mathcal{S}$ consist of an induced path between N_1 and N_2, and a disjoint induced path between S and $N_1 \cup N_2$.

We also refer to the three types of sprouts as $\mathcal{S}_i$, $i = 1, 2, 3$. These sprouts are an important tool in showing the existence of a three-in-a-tree. In particular, if sets outside of the web are connected to a sprout, this will always lead to the existence of a three-in-a-tree. Luckily, we can introduce a type of arc, called *flexible arc* that contains sprouts for all of

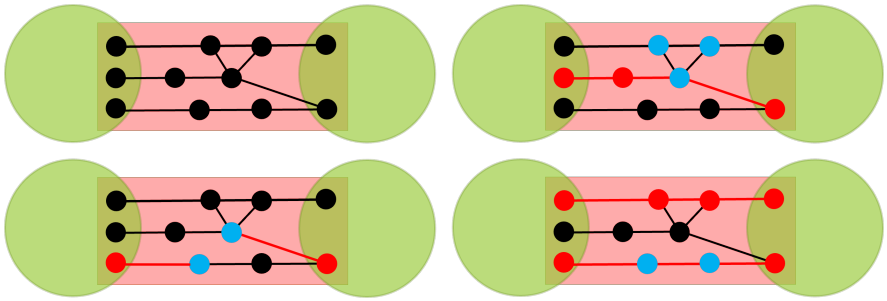

FIGURE 24: A flexible arc with all three types of sprouts. The set S in blue and an (S, N_1, N_2)-sprout in red.

its subsets, and grow our web in such a way that it will only contain simple and flexible arcs.

Definition 4.13 (Flexible arcs)**.** A *flexible arc* of a web consist of vertices $A \subseteq V$ that connect two end-nodes N_1 and N_2, such that for every non-empty subset $S \subseteq A$, (S, N_1, N_2) is a sprout.

A flexible arc containing sprouts of all three types is shown in Figure 24. Here the vertices in S are shown in blue and an (S, N_1, N_2) sprout is shown in red, along with the edges it induces.

Remember in a web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ every pair of vertices $v_1, v_2 \in N$ in a node $N \in \mathcal{N}$ that lie in different arcs, have an edge between them. If two vertices share the same flexible arc, they do not necessarily need to be connected by an edge but might be. As shown later the operation MERGE(C) as introduced above is only used in such a way that our web $\mathbb{H}$ maintains the property that it only consists of simple and flexible arcs. In our aiding web, other more complex and non-flexible edges might arise as a result from merging arcs. The aiding web is, however, only used for obtaining fewer wild sets. Our web $\mathbb{H}$ will always consist of only simple and flexible arcs. Therefore, we only need to focus on a web with simple and flexible arcs and show that such a web still allows no trees. In order to do so, we now extend Lemmas 4.2 and 4.7, and show that they still hold for a web with simple and flexible arcs.

Lemma 4.5 (Web contains no tree - flexible web version)**.** *There exists no subset of vertices of a flexible web that induces a tree containing all three terminals.*

Proof. Firstly, following the reasoning of Lemma 4.2, we can immediately conclude that the vertex v_c cannot lie in a simple arc, or in a node such that the vertex is the intersection of a node and a simple arc. We consider two more cases, either the center vertex v_c is part of a flexible arc but not of a node, or v_c lies in the intersection of a flexible arc and a node.

1. First assume that v_c is part of a flexible arc A, but not of any node and assume that our graph G contains a subset of vertices $X \subseteq V$ that induces a tree between the terminals. Then the paths P_1, P_2, P_3 from v_c to the terminals $t_1, t_2, t_3 \in T$ respectively, must all consist of vertices from A, and vertices not from A. Possibly exactly one of the terminals lies in A, however that will not matter for the following argument. Because three paths 'leave' arc A, we know that at least two of the paths must do so via the same node N_1. Indeed, note that if one of the terminals is part of A, the other two paths still contain vertices from a mutual node N_1 and the former

still holds. Without loss of generality assume that the paths P_1 and P_2 contain vertices of N_1, and denote by P_1' and P_2' the sub-paths from v_c until the first vertices of each path that lies in node N_1. Clearly, not both of these paths can end in N_1 and therefore the set X contains at least one more vertex $v \in X$ to induce a tree connecting the terminals. However now $v_c \cup P_1' \cup v \cup P_2' \subseteq X$ and these vertices form a cycle. This is a contradiction and v_c is not part of the interior of a flexible arc.

2. Secondly, assume that v_c lies in the intersection of a flexible arc A and a node N and we have a set $X \subseteq V$ that induces a tree connecting the terminals. Then, there exists at most one other arc A' that contains vertices of $X \cap N$, because otherwise we find an induced triangle by the definition of a node. In particular, all vertices of X in N lie in either A or A'. Therefore, there must be a flexible arc that contains at least two neighbours of v_c. These two neighbours then lie on paths P_1 and P_2 to terminals t_1 and t_2. As stated before, X contains no other vertices of N that are not a part of A or A'. Clearly then, these paths P_1 and P_2 leave their common arc A (or A') through the same end-node N_2. Because at least one terminal can lie in N_2, we know that X also contains a vertex v in N_2 from a different third arc A''. Then taking again P_1' and P_2' as the sub-paths from v_c until the first vertex in N_2, we find that $v_c \cup P_1' \cup v \cup v \subseteq X$ forms a cycle.

We conclude that the vertex $v_c \in X$ cannot be part of any node, simple or flexible arc of a web where v_c contains no neighbours that are not present in the web. We will extend the proof for tamed sets, meaning that if our graph with a flexible web $\mathbb{H}$ is $\mathbb{H}$-tamed, there cannot be any $X \subseteq V$ that induces a tree containing the terminals. □

Lemma 4.6 (Tamed web - flexible web version)**.** *Take a simple undirected graph $G = (V, E)$, three terminals and a flexible web $\mathbb{H}$ over G. If the graph G is $\mathbb{H}$-tamed, the graph G with the given terminals does not allow three-in-a-tree.*

Proof. Assume to the contrary that there exists a set $X \subseteq V$ that induces a tree containing the three terminals. Following the proof of Lemma 4.5 we can see that the center-vertex v_c of our tree induced by X cannot be part of any node, simple or flexible arc. This can be seen because the argument that multiple paths must leave arcs through the same node still holds when our graph is $\mathbb{H}$-tamed, as such tamed sets are only paths that have end-vertices in the same solid set. There are however two more options. Either v_c is the end-vertex of some tamed path P, or v_c is part of a flexible arc A and has at least one neighbour in some tamed path P.

1. In the first case, where v_c is the end vertex of some tamed path P, note that at least two neighbours of v_c must all lie in either the same node or arc. Also the other end-vertex of the path P that is not v_c can have only neighbours in this node or arc, and share no neighbours with v_c as otherwise they form a cycle. Therefore, there are at least three vertices of X that share some common node or arc, and are part of three disjoint paths to the terminals of the three. If they share a common node, they must also share a common arc because otherwise they form a cycle together with v_c from the definition of a node. Thus there are three vertices on three disjoint paths P_1, P_2, P_3 that share an arc. Again at least two of these paths contain vertices from one end-node N_1 of the common arc. Moreover, because X must contain a vertex of N_1 that is not part of the common arc, we find a cycle.

2. Now assume v_c is part of a flexible arc A and has at least one neighbour in some tamed path P. Because v_c cannot be part of a node, we know that v_c lies in the

interior of arc A, and has exactly one neighbour in P, as well as two neighbours in A. The end-vertex of P that is not a neighbour of v_c then also only contains neighbours in A, by definition of a tamed web. Again we have three vertices that share an arc A, but lie on three disjoint paths to the terminals and the conclusion of the proof is the same as above.

In conclusion the vertex v_c does not exist, and therefore there is no $X \subseteq V$ that induces a tree connecting the terminals. □

4.6 Solid sets

There are two different types of $\mathbb{H}^\dagger$-wild sets that will not allow us to immediately conclude the existence of a tree connecting the terminals. We will show how to find these sets, and consequently how to grow the size of our web. The first type are solid sets.

Definition 4.14 (Solid sets)**.** Given a graph $G = (V, E)$ with web $\mathbb{H}$ consisting of the vertices $X \subseteq V$, a set $S \subset X$ is $\mathbb{H}$-*solid* if S is either equal to all vertices in a single node, or S is a subset of an arc $A = N_1N_2$ such that there is no (S, N_1, N_2)-sprout. Moreover, a set $Y \in V\backslash X$ is also referred to as $\mathbb{H}$-solid if Y is an induced path $y_1y_2 \dots y_K$ such that both $N_X(y_1)$ and $N_X(y_K)$ are solid sets, and $N_X(y_j) = \emptyset$ for $j = 2, \dots, K-1$.

We already know that by definition flexible arcs contain sprouts for every subset S of their vertices. Therefore we only need to consider the vertices in simple arcs. It turns out that sprouts are present for most subsets S of a simple arc, except for one specific case.

Lemma 4.7. *Given a graph $G = (V, E)$ with a web $\mathbb{H}$ and a simple arc $A = N_1N_2$ of the web, consisting of the vertices in an induced path P. A set $S \subseteq P$ is $\mathbb{H}$-solid if and only if $|S| = 2$ and the vertices $s_1, s_2 \in S$ are neighbours.*

Proof.
(Sufficiency)
Let $s_1, s_2 \in S$ be closest to N_1 and N_2 respectively. If $s_1 = s_2$ then clearly there exists a sprout of type $\mathcal{S}_1$. Otherwise, if $s_1 \neq s_2$ and s_1, s_2 are not neighbours, there exists a sprout of type $\mathcal{S}_2$. Therefore, s_1 and s_2 must be two separate vertices that are neighbours. Because s_1 and s_2 were chosen closest to N_1 and N_2, it follows that $|S| = 2$.

(Necessity)
Assume we have a set $S \subseteq P$ with $|S| = 2$ and $s_1 \in N(s_2)$ that is a subset of a simple arc $A = N_1N_2$. We need to show that there exists no induced subgraph $\mathcal{S}$ that is a (S, N_1, N_2)-sprout. Clearly the only tree that intersects both N_1 and N_2 is equal to P, so no sprout of type $\mathcal{S}_1$ exists.

Assume s_1 is the vertex closest to N_1 on P. Then, every induced path between S and N_1 contains at least s_1, and similarly every induced path between S and N_2 contains s_2. Because s_1 and s_2 are neighbours, these paths are never disjoint and there exists no sprout of type $\mathcal{S}_2$.

Lastly, every induced path between N_1 and N_2 contains S, so there exists no sprout of type $\mathcal{S}_3$.

Therefore no sprout exists and S is $\mathbb{H}$-solid. □

All that is left to show is that a wild $\mathbb{H}$-solid set $Y \in V\backslash X$ can be used to grow the size of our web.

Theorem 4.8. *If for a web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ we have a $Y \in V\backslash X$ that is $\mathbb{H}$-solid, we can strictly increase the size of our web maintaining its properties, such that the new web contains only simple and flexible arcs.*

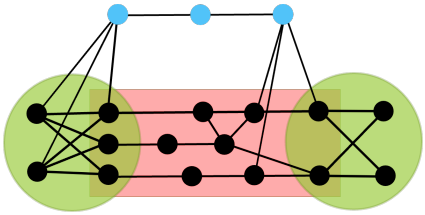

FIGURE 25: A flexible arc with an $\mathbb{H}$-podded wild set in blue.

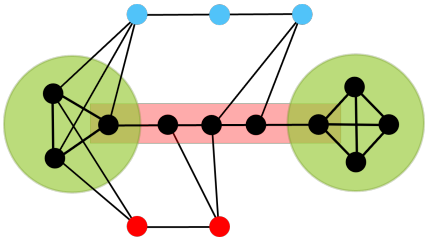

FIGURE 26: A flexible arc with two wild sets. The blue set is both $\mathbb{H}$-podded and $\mathbb{H}$-solid. The red set is $\mathbb{H}$-podded but not $\mathbb{H}$-solid.

Proof. From the definition of an $\mathbb{H}$-solid set, together with Lemma 4.7 we know that Y is an induced path and the neighbours of its end-points in X are either a node or two neighbouring vertices in a simple arc. We denote by $S_j = N_X(y_j)$ the endpoints of the induced path Y.

If S_j is a node, we simply add vertex y_j to the node. If S_j is equal to two vertices s_1, s_2 in an arc $A = N_1N_2$ such that s_1, s_2 is an edge and s_1 is the vertex in S closest to N_1 and the arc A consists of vertices on the induced path P, we do three things:

1. Create a node $N = \{s_1, s_2, y_j\}$;
2. Create two new arcs: $A_1 = N_1N$ consisting of the vertices of path P from $N_1 \cap A$ to s_1 and $A_2 = N_2N$ consisting of the vertices on P from s_2 to $N_2 \cap A$;
3. Remove the arc A from the web.

Lastly, we create a new arc A_Y with vertices Y and its induced path $y_1y_2 \dots y_K$. It is easy to directly confirm that $\mathbb{H}_{new}$ is still a web of the graph, while its size has strictly increased by adding the vertices of Y. Moreover, the newly added arc is a simple arc. If one more or more arcs of the web are split up when a new node is created, this arc was simple by definition of a solid set, and the resulting two arcs are also clearly simple. Therefore, we maintain the fact that our web consists of only simple and flexible arcs. □

4.7 Podded sets

The second type of wild sets that do not allow us to directly conclude that a graph and its terminals allow three-in-a-tree are *podded sets*. First we need to define pods.

Definition 4.15 (Pod)**.** Given a graph $G = (V, E)$ with web $\mathbb{H}$ consisting of the vertices $X \subseteq V$, a pod of $Y \subseteq V \backslash X$, such that Y is an induced path $Y = y_1y_2 \dots y_K$ is a chunk $C = N_1N_2$ such that $N_X(Y) \subseteq N_1 \cup C \cup N_2$ and for both end-nodes of the chunk, i.e. for $i = 1, 2$ either

1. $N_{N_i}(y) \subseteq C$, or
2. $N_i \subseteq C \cup N(y)$

holds for an end-vertex $y \in \{y_1, y_K\}$. Moreover, we say that Y allows a pod in the web $\mathbb{H}$ if such a pod exists. A *podded set* is defined as an induced path $Y \subseteq V \backslash X$ such that Y allows a pod in $\mathbb{H}$. We might then also say that the set Y is $\mathbb{H}$-podded.

Examples of a wild sets with a pod are shown in Figures 25 and 26. Note that the wild set of blue vertices in Figure 26 is both solid and podded. Similarly to Theorem 4.8 we want to show that a podded wild set allows us to increase the size of our web. Note that if a set Y is $\mathbb{H}$-podded we can always find such a unique smallest pod of this set, i.e. a pod $C = N_1N_2$ that is a minimal chunk and there is no other chunk consisting of only a subset of the arcs of chunk C. Our algorithm to grow the web will only work for such a smallest pod. Luckily such a smallest pod is easy to find.

Lemma 4.9. *For a web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$, let $Y \subseteq V \backslash X$ be $\mathbb{H}$-podded and not $\mathbb{H}$-solid. We can construct a new web $\mathbb{H}'$ where Y has a minimal pod $C' = N_1N_2$ such that for each end-node N_j either*

1. *there is more than one arc of C' incident with N_j,*
2. *there is a single flexible arc of C' incident with N_j, or*
3. *there is a single simple arc of C' incident with N_j and $N_X(Y)$ intersects $N_2 \backslash C$.*

Moreover, this new web will only consist of simple and flexible arcs.

Proof. Because the set Y is $\mathbb{H}^\dagger$-wild, we know that for at least one of the end-nodes of C, which we assume w.l.o.g. is N_1, it holds that $N_X(Y) \subseteq N_1 \backslash C$. Now we look at the other end-node N_2. If there is a flexible arc of C incident with N_2 or there is more than one arc of C incident with N_j, we are immediately done. Assume there is a single simple arc $A = MN_2$ incident with N_j and $N_X(Y)$ does not intersect $N_2 \backslash C$. From the minimality of the pod we must have $|N_X(Y) \cap (C \backslash M)| \geq 1$. Now let v_2 be the vertex on the path of arc A closest to N_2 (possibly in N_2), and v_3 the neighbour of v_2 on the same path and further away from N_2. Then, create a new node N_3 consisting of the vertices v_2 and v_3, and update the arcs accordingly. Clearly, the result is still a web $\mathbb{H}'$ of the graph G consisting of simple and flexible arcs. However, there exists a new smallest chunk $C' = N_1N_3$ and $N_X(Y)$ intersects $N_3 \backslash C$. □

Now we are ready to add our $\mathbb{H}^\dagger$-wild set to our newly found web $\mathbb{H}'$.

Theorem 4.10. *If for a web $\mathbb{H}$ containing vertices $X \in V$ we have an $Y \in V \backslash X$ that is $\mathbb{H}$-podded and not $\mathbb{H}$-solid, we can strictly increase the size of our web maintaining its properties, such that the new web contains only simple and flexible arcs.*

Proof. We can first use Lemma 4.9 to obtain a new web which we will still denote $\mathbb{H}$ and a minimal pod $C = N_1N_2$ obeying the conditions of the lemma. We now simply merge all arcs of C into one flexible arc and add all vertices of Y to that arc. Moreover, each end-vertex y_j of Y must have that for one of the nodes N_i, $N_i \subseteq C \cup N_X(y)$, by definition of a pod and Lemma 4.9. Therefore we can add the vertex y_j to such a node N_i and preserve the web condition for nodes.
The only thing that we still need to show is that the resulting merged arc A is in fact flexible and we are done. This proof is very tedious and it is needed to identify one of three sprout types in many different cases. We refer to the proof of Lemma 3.2(2) in [18]. □

4.8 Three-in-a-tree algorithm

In the sections above we have shown two things:

1. For a graph G and terminals T with a taming web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$, there exists no induced tree connecting the terminals.

2. Let G be a graph with terminals T, a taming web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ and a wild set $Y \subseteq V \backslash X$ such that Y is either $\mathbb{H}$-solid or $\mathbb{H}$-podded. We can then increase the size of our web.

If we can show that all other wild sets Y that are neither podded nor solid will allow us to directly decide something about the existence of trees for the three-in-a-tree problem, we have a clear path to an algorithm. This is possible and in particular we find that all such wild sets allow us to directly find a tree connecting the terminals. For this we first need an important lemma. Let us call the graph where nodes and arcs of the aiding web $\mathbb{H}^\dagger$ are edges and vertices $H^\dagger$.

Lemma 4.11 (Center vertex in an aiding web)**.** *Given a graph $H^\dagger$ that is the graph representation of an aiding web $\mathbb{H}^\dagger$, then from every vertex of $H^\dagger$ that has degree at least 3, there exist disjoint paths to the three leaf vertices of $H^\dagger$.*

Proof. To the contrary, assume there exists a vertex v with degree at least 3, but there are no disjoint paths to the three leafs of the graph. First note that the graph only contains exactly 3 leafs, and therefore there exist paths from v through three of its neighbours v_1, v_2, v_3 to the leafs that thus do not share the first vertex on their path. Now, because we assumed the contrary of the lemma, there must be a vertex on every pair of paths of these of these vertices, without loss of generality v_1 and v_2. Now, from a special case of Menger's theorem [19] these paths must then all intersect in a single vertex w. Therefore, all the arcs between the node variants of v and w share these nodes as their split-nodes and form a non-trivial chunk. Thus these arcs should have been merged and cannot be present in the aiding web $\mathbb{H}^\dagger$. This is a contradiction and thus the lemma is true. □

The lemma above shows us that if we can find, with the help of a wild set, a vertex v_c that contains 3 disjoint paths to three different aiding arcs, then we will find a tree that connects the terminals. Clearly, in such a graph where a vertex is connected to three different aiding arcs, there are no additional chunk-like structures. That we can always find such a vertex if a wild set is neither solid or podded is shown in the theorem below.

Theorem 4.12 (Wild set theorem)**.** *Given a graph $G = (V, E)$ with a web $\mathbb{H}$ consisting of vertices X and terminals T, and a non-solid non-podded vertex-minimal $\mathbb{H}^\dagger$-wild path Y, there is a subset of the vertices $X \cup Y$ that induce a tree connecting the terminals.*

Proof. There are two cases to consider. Either the neighbours of the end-vertices of the path Y are all part of one aiding arc A and its end-nodes N_1 and N_2, or the end-vertices of the path lie in at least two such arcs and their end-nodes.

In the first case, because the $\mathbb{H}^\dagger$-wild set is neither podded or solid, there must be at least one end-node, w.l.o.g. N_1, and a vertex v in this node, but not in A such that $N_X(Y)$ intersects the node minus the arc $N_1 \backslash A$, but $v \notin N_X(Y)$, as other-wise Y is either solid or podded. First, observe that the node N_1 has vertices from at least three different arcs, as otherwise we obtain a non-trivial chunk in our aiding web.

Therefore, there must exist two vertices $u, w \in X$ such that u and w do not share an aiding arc, and u is a neighbour of Y in the web, but w is not. Now let $v_c = u$. This vertex has a neighbour in its own aiding arc A', it has as a neighbour the vertex w from a different aiding arc A'', and it neighbours the path Y which connects to arc A. Thus this vertex v_c has three disjoint paths to three different aiding arcs, and it follows that there is a tree connecting the terminals by Lemma 4.11.

In the second case, the set Y is clearly not podded, as it is not contained by a single aiding arc and its end-nodes. Because the wild set is also not solid, there must be an end-vertex y_1 of the path Y which neighbours $N_X(y_1)$ do not form a solid set. If its neighbours are part of two different nodes or arcs, but do not consist of an entire triad, finding a tree is trivial from Lemma 4.11. If however, its neighbours $S = N_X(y_1)$ are a subset of a single simple arc, flexible arc or a node a bit more care is needed. We look at all cases for this set S.

1. If the set of neighbours S of y_1 in the web is a subset of a simple arc A, then we know that either $|S| \neq 2$ or the two vertices in S are not neighbours. If $|S| = 1$, the neighbour of y_1 is the center vertex of our web. In both cases $|S| > 2$ and $|S| = 2$ the vertex y_1 has two non-neighbours in the web because S is the subset of a simple arc. Therefore, y_1 is the center-vertex of our web, because it can induce paths to two different nodes and a different arc or node of our graph.

2. If the set of neighbours S of y_1 in the web is any subset of a flexible arc, then by definition of a flexible arc we know that there must exist a sprout for this set S in the arc. For each sprout type $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ it is not hard to see that this leads to finding a center-vertex in the aiding arc, or in case of a sprout of type $\mathcal{S}_3$ in its end-node.

3. If the set of neighbours S of y_1 in the web is a subset of a node, then any neighbour of y_1 in this node can be the center-vertex, as there exists another vertex in this node that is a non-neighbour of y_1, it has a neighbour in its own arc, and it has y_1 has neighbour which via path Y connects to a different aiding arc, and we again find a vertex that has disjoint paths to three different aiding arcs.

All in all, in all cases that we have a wild set Y that is not solid or podded, we can construct a tree connecting the terminals from the vertices of $X \cup Y$. □

The only thing that is left for us to do, is to show how to obtain an initial web $\mathbb{H}$ for a given three-in-a-tree problem. This is easily done as shown in the following section.

4.8.1 Initialisation and complete three-in-a-tree algorithm

We turn to the step of finding an initial web for a given three-in-a-tree problem. If we could conclude immediately that for our given graph there exists a solution there is no need to find an initial web. For all graphs of which we cannot make an immediately conclusion however we do need to find some set of vertices that form a web according to definition. The following theorem shows us a very simple way to achieve this.

Theorem 4.13. *Given a three-in-a-tree problem with graph G and terminals $T = \{t_1, t_2, t_3\}$ we are able to obtain an initial web or decide there exists a solution to the three-in-a-tree problem in time $O(n^2)$.*

Proof. We calculate a shortest path P_1 between t_1 and t_2, and a shortest path P_2 between t_3 and all vertices of P_1. We look at the graph H that is induced from G by the vertices in P_1 and P_2. First of all note that the graph induced only by the vertices in P_1 or only by the vertices in P_2 contains no cycles, as those are shortest paths. We will refer to the second to last vertex in path P_2 as a_3, i.e. $P_2 = t_3 p_1, p_2, ..., a_3, p_K$. clearly p_K is also a vertex of the path P_1 and a_3 is not. Moreover a_3 is the only vertex of P_2 with neighbours in P_1, as otherwise P_2 would not be a shortest path.

We look at the number of neighbours of a_3 that lie on the path P_1. If $|N_{P_1}(a_3)| = 1$

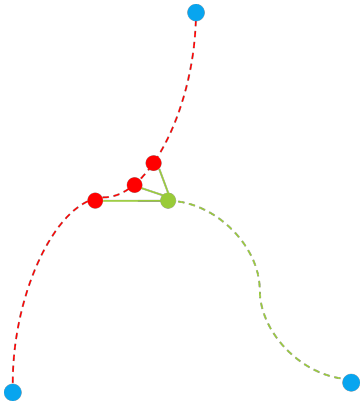

FIGURE 27: Shortest paths connecting the blue terminal vertices T. In red dashes path P_1 in green dashes path P_2. The green vertex is vertex a_3. The vertex a_3 has 3 neighbours on the path P_1, allowing for a tree.

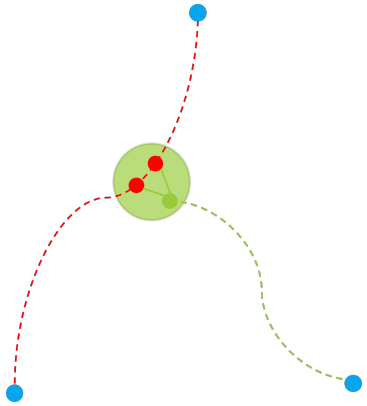

FIGURE 28: Shortest paths connecting the blue terminal vertices T. In red dashes path P_1 in green dashes path P_2. The green vertex is vertex a_3. The neighbours of a_3 on P_1 together with a_3 form a triangle.

we have found a tree consisting of all vertices in P_1 and P_2 with center-vertex $v_c = p_K$. If $|N_{P_1}(a_3)| \geq 2$ we define $a_1 \in N_{P_1}(a_3)$ as the vertex closest to t_1 on P_1 and a_2 closest to t_2 on P_1, see Figure 27. Let P_3 be the sub-path connecting a_1 and a_2 on P_1. If a_1 and a_2 are not neighbours, it becomes immediately clear that the vertices $(P_1 \cup P_2) \backslash P_3$ induce a tree with as center vertex $v_c = a_3$.

Otherwise, if the vertices a_1 and a_2 are neighbours as seen in Figure 28, then no subset S of the vertices in $P_1 \cup P_2$ induces a tree connecting the terminals. This is clear from the fact that a_1, a_2 and a_3 all must be elements of S to connect the terminals, but they induce a triangle. In this case consider the set $X = P_1 \cup P_2$, create arcs A_i from the unique paths in X connecting t_i and a_i for $i = 1, 2, 3$ and create a node $N = \{a_1, a_2, a_3\}$. It can be directly verified from Definitions 4.2, 4.3 and 4.4 that the vertices in X partitioned into these arcs and node form a web $\mathbb{H}$ over our graph.

It is easy to see that the steps above can decide in time $O(n^2)$ for every graph G whether there exists a trivial solution to the three-in-a-tree problem, or obtain an initial web. □

Concluding from all theorems and proofs of the sections above, the algorithm to decide on three-in-a-tree problems has the following form for a given three-in-a-tree problem with graph $G = (V, E)$ and terminals T. Running time and implementation strategies will be discussed in the next section, therefore we omit good strategies here to obtain for example wild sets, which is clearly possible but the asymptotic running time is unclear for now.

Theorem 4.14 (Three-in-a-tree correctness)**.** *The three-in-a-tree algorithm described in Algorithm 10 and concluding all the sections above determines correctly whether a given graph $G = (V, E)$ contains a three-in-a-tree for given terminals T.*

Proof. The proof is based on the fact that we know that during our entire algorithm, our web $\mathbb{H}$ consists of only simple and flexible arcs based on its initialization and the proofs of Theorems 4.8 and 4.10. If there is a tamed set there is no tree from Lemma 4.6. Moreover, if there is a wild set connecting to a web of simple and flexible arcs that is neither solid or

Algorithm 10 Three-in-a-tree algorithm

```
1: Obtain an initial web ℍ.
2: while true do
3:     if ℍ† is taming then
4:         return false.
5:     else
6:         Obtain a vertex-minimal induced path Y ⊆ V\X that is ℍ†-wild.
7:         if Y is neither ℍ-solid or ℍ-podded then
8:             return true.
9:         else if Y is ℍ-solid then
10:            Add Y to the web.
11:            Update the aiding web accordingly
12:        else if Y is ℍ-podded then
13:            Add Y to the web.
14:        end if
15:    end if
16: end while
```

podded, there always exists a solutions to the three-in-a-tree problem according to Theorem 4.12. If the set is solid or podded we know for sure that the vertices of our web together with that solid or podded set do not contain a solution to the three-in-a-tree problem, because they can be joined in a web with only simple and flexible arcs. Moreover, our web strictly increases each iteration because each wild set consists of at least one vertex and is always completely added to the web. □

4.9 Improved pyramid detection

In the sections above we described how we can find out for a graph and three terminals whether this graph contains a set of vertices that induces a tree connecting these terminals. It was already hinted that this is used to now improve upon the algorithm for pyramid detection in Algorithm 4. We now show how we can use Algorithm 10 to detect pyramids. When discussing all the implementation strategies in the Section below, we will find that we can implement the three-in-a-tree algorithm in time $O(n^3)$. This algorithm also clearly needs to be run at most $O(n^3)$ times, as its input consists of 3 terminals. But, how can we decide for three terminals of our graph if there is a three-in-a-tree whether a pyramid also exists? This is shown below.

Theorem 4.15 (Improved pyramid detection)**.** *Given a graph $G = (V, E)$ and a three-in-a-tree algorithm that runs in time $O(n^3)$, we can determine whether G contains a pyramid using Algorithm 11 in time $O(n^6)$.*

Proof. Assume graph G contains a pyramid induced by a set $H \subseteq V$. Then the base b_1, b_2, b_3 is generated as a set T of terminals. Clearly in the graph without the edges between each b_i and b_j the vertices of H induce a tree between the base vertices. No vertices of $N(t_i) \cap N(t_j) \setminus \{t_k\}$ are in H, because the top vertex of a pyramid is adjacent to at most one of the base vertices. Moreover, exactly one neighbour of each base vertex is present in H. Therefore the vertices of H still induce a tree between the base vertices in graph G', and this tree is found by our three-in-a-tree algorithm.

Now assume our algorithm determines that there exists a three-in-a-tree for some set of terminals T and graph G'. Then, this tree contains exactly one vertex $N(t_i)$, $i = 1, 2, 3$

Algorithm 11 Pyramids from three-in-a-tree

```
Generate all sets T = {t1, t2, t3} ⊆ V that induce a triangle
for Each set T do
    Create a copy graph G' = (V, E)
    Remove all vertices N(ti) ∩ N(tj) \ {tk}, with i, j, k permutation of {1, 2, 3}
    Remove edges between all ti and tj
    Add edges to each set of neighbours N(ti), i = 1, 2, 3
    if G' contains an induced tree for terminals T then
        return True
    end if
end for
return False
```

thus none of the edges added to the graph are present in the three. Clearly, there are no edges or overlapping vertices between the paths of the base vertices to the center-vertex v_c of the three, and this center vertex is the top-vertex of the pyramid. Consequently, by adding the edges between the base vertices we find a pyramid. Adding the deleted vertices $N(t_i) \cap N(t_j) \backslash t_k$ does of course not change the graph induced by the vertices of the three-in-a-tree that was found.

For the running time, note that we loop over $O(n^3)$ sets of terminals. Changes made to the graph can be done in $O(n^2)$ per triangle, running the three-in-a-tree algorithm takes $O(n^3)$ per triangle by assumption, which is shown to be true in the next section. □

5 Three-in-a-tree implementation

In the sections above an algorithmic idea to solve the three-in-a-tree problem was introduced. The asymptotic running time for the algorithm however is not immediately clear, for example considering adding solid and podded sets to our web. In the next sections we will introduce implementation strategies and algorithmic ideas to obtain a running time of $O(n^3)$ to solve the three-in-a-tree problem for a given graph G and terminals T. The implementation strategy for our three-in-a-tree algorithm follows a similar idea to Lai et al. [18]. They provide implementation ideas for an algorithm with a running time of $O(n^2 \log n)$. This algorithm, however, relies heavily on highly theoretical graph theory results from other papers, such as dynamic and incremental SPQR-trees [13], dynamic spanning forests [17] and top space forests [1]. These results are nearly impossible to implement and it is even unclear if their implementation would pay off in practice.

Therefore, we present a more 'implementable' algorithm that does not rely on other results. This algorithm has a running time of $O(n^3)$ instead of $O(n^2 \log n)$ [18], but the trade-off is enough as we will review later. Specifically the resulting algorithm for pyramid detection in time $O(n^6)$ will have the same running time as many of the other subalgorithms for Berge graph detection. We start by introducing some general implementation ideas, after which we will explain how we obtain wild sets and grow the web over our graph. Lastly, the maintenance of the aiding web is discussed.

To start our implementation we are given some graph $G = (V, E)$ with three terminals T. Moreover we assume we have an adjacency matrix containing the incidence relation between our vertices. Remember that by $N(v)$ and $N_S(v)$ we denote the neighbours of a vertex v and the neighbours of v in the set S respectively.

5.1 Colouring vertices

In order to maintain our web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ we will associate colours with every arc, aiding arc and node of a web. Every vertex that is present in the web is then coloured using a maximum of three colours: one arc colour, and at most two node colours. Because the vertices in an arc or a node might be recoloured during our algorithm we refer to a node or arc of a given colour as *dummy* when it contains no vertices. Moreover, for each node we maintain information of whether its present in the aiding web. Lastly, for the arcs we store whether it is a *simple* or a *flexible* arc.

Note that we in particular do not store for every component which vertices it contains as this will lead to slower running times, however we do maintain the size of every component as well as the size of the intersection between every arc and node. For simple arcs we do store the vertices in its induced path as a linked list. To speed up the recolouring of our vertices we additionally save a colour inheritance map that will map an arc colour to a different one. Below we summarize all the information that we store during our algorithm that was introduced above.

Web information

List of Arcs, $\mathcal{A}$;
List of Aiding arcs, $\mathcal{A}^\dagger$;
List of Nodes, $\mathcal{N}$;
Map containing colour inheritance, $\mathcal{M}(A) : \mathcal{A} \to \mathcal{A}$;
Map containing intersection size of arcs and nodes $\mathcal{I}(A, N) : (\mathcal{A}, \mathcal{N}) \to \mathbb{N}$;
Map that indicates for every arc A part of which aiding arc $A^\dagger$ it is.

Arc information

Arc colour, A_c;
Arc size, $|A|$;
Arc type, $A_{\text{type}} \in \{\text{simple, flexible, dummy}\}$;
Arc path, A_P, (only for simple arcs).

Node information

Node colour, N_c;
Node size, $|N|$;
Node type, $N_{\text{type}} \in \{\text{aiding, non-aiding, dummy}\}$.

Vertex information

Arc colour $v_A \in \mathcal{A}$;
Node colours $v_{N_1}, v_{N_2} \in \mathcal{N}$.

Note that each vertex does not store itself directly which aiding arc colour it has, but

it knows its arc color v_A, and the arc A knows in which aiding arc $A^\dagger$ this arc A lies. Therefore, we do write directly $v_{A^\dagger} \in \mathcal{A}^\dagger$, to indicate that every vertex keeps track of its aiding arc colour.

5.2 Obtaining wild sets

In order to decide if a solution exists to our given three-in-a-tree problem, an important sub-part of Algorithm 10 suggests that we must obtain $\mathbb{H}^\dagger$-wild sets of our web or decide that none exist. Clearly, taking all possible subsets of vertices outside the web and checking whether they are a vertex-minimal wild induced path will never lead to a running time of $O(n^2)$. A more intricate idea is introduced in this section to handle this problem. This idea consists of two steps, where we first want to create a database which allows us to check in constant time if a vertex is $\mathbb{H}^\dagger$-wild and secondly use this database twice to obtain a vertex-minimal $\mathbb{H}^\dagger$-wild induced path in time $O(n^2)$.

5.2.1 Representative sets

First we will introduce the concept of *representative sets*. Our goal is to be able to find sets containing of at most three vertices that will represent the behaviour of the neighbours of sets outside the web.

Definition 5.1 (Representative set)**.** Given a three-in-a-tree problem with a graph G, terminals T and a web $\mathbb{H} = (X, \mathcal{N}, \mathcal{A})$ over the graph. A set $R \subseteq X$ is a *representative set* of a set $S \subseteq X$, if

1. R contains at most three vertices,
2. R is $\mathbb{H}^\dagger$-tamed if an only if S is $\mathbb{H}^\dagger$-tamed and
3. for every other set $Z \subseteq X$ we have that $R \cup Z$ is $\mathbb{H}^\dagger$-wild if and only if $S \cup Z$ is $\mathbb{H}^\dagger$-wild.

We will show that such representative sets for a set $S \subseteq X$ can be generated in time $O(|S|)$. Clearly if the set S is wild we can simply choose two vertices that do not share a node or an arc and R is also wild. If two sets are tamed however, we must be careful to ensure that we correctly determine whether they are wild together. Moreover, it then follows that if we have two representative sets R_1, R_2 for Y_1 and Y_2 respectively, we will be able to find the representative set R for $Y_1 \cup Y_2$ in constant time, as $|R_1 \cup R_2| \leq 6$. The theorem below shows that this is indeed the case.

Theorem 5.1 (Representative set generation)**.** *Given a three-in-a-tree problem with graph G, terminals T and a web $\mathbb{H}$ we are able to find a representative set of a set $S \subseteq X$ in time $O(|S|)$ using Algorithm 12.*

Algorithm 12 Representative sets

1: Collect the set of aiding arcs $\mathcal{A}_S \subseteq \mathcal{A}^\dagger$ of which a vertex is in S.
2: **if** $|\mathcal{A}_S| = 1$ **then**
3: **return** a vertex from both end-nodes and an internal vertex of the arc if they exist.
4: **else**
5: Determine a vertex $s_1 \in S$ with a minimal amount of Node colours.
6: **if** vertex v_1 has no node colours **then**
7: **return** a vertex v_1 together with a vertex v_2 such that $(v_2)_{A^\dagger} \neq (v_1)_{A^\dagger}$.
8: **else if** vertex v_1 has one node colour N **then**
9: **if** all vertices in S also have node colour N **then**
10: **return** vertex v_1, and at most 2 other vertices that do not share arcs with v_1 or each other.
11: **else**
12: **return** two vertices that do not share any node and arc colours.
13: **end if**
14: **else**
15: **if** the vertices lie in a triad **then**
16: **return** a vertex that lies in every pair of nodes N_i, N_j $i \neq j$.
17: **else**
18: **return** two vertices that do not share any node and arc colours.
19: **end if**
20: **end if**
21: **end if**

Proof. We will first show correctness. Clearly, the algorithm considers all possible cases of sets in X and always return a set of at most three vertices. For our proof we will distinct the cases where S is a wild or a tamed set. We consider the return statements in lines 3, 7, 10, 12, 16 and 18 and argue that they all return a correct representative set.

(7,12,18) First observe that the set S is clearly $\mathbb{H}^\dagger$-wild if the algorithm reaches the return statements in lines 7, 12 and 18, and a wild set is indeed returned containing two vertices that share no arc and node colours. Therefore, we can immediately conclude that such a wild set consisting of two vertices is indeed by definition a correct representative set of S.

(3) Clearly if all vertices of S lie in a single arc, the set is tamed. Because there are only three different types of vertices, one in both end-nodes and one internal vertex of the arc outside of the nodes, it becomes immediately clear that picking one from every one of these cases if it exist suffices to represent S. Therefore line 3 returns a correct representative set.

(10) Consider the set R returned in line 10, where all vertices have node colour N. Clearly S and R are both tamed sets. If a set $Z \subseteq X$ is wild, then also both $S \cup Z$ and $R \cup Z$ are wild. If Z is tamed and $S \cup Z$ is tamed as well, $Z \cup R$ is by definition tamed too. Now assume that Z is a tamed set, but $S \cup Z$ is wild. This case requires a little more care. First, we can without loss of generality assume that $|Z| = 1$, and denote $Z = \{z\}$. Then, because $S \cup z$ is wild z cannot have N as one of its node colours.

The set R returned either contains three or two vertices. We look at the case where $|R| = 2$ first. Both vertices in R have a different arc colour, and share node colour N. Note that in our aiding web, there exist no parallel arcs between two nodes, as those

surely share their split pair. Therefore, the arcs that r_1, r_2 lie in have a different end-node N_1, N_2. Assume to the contrary now that $R \cup z$ is tamed, and therefore we may assume that z also has node colour N_1 as it needs to share at least a node with r_1 or r_2. If z and r_2 share an arc, then that arc must be $A = NN_1$, which is a contradiction. On the other hand if z and r_2 share a node N_2 then they lie in a triangle zr_1r_2. Such a triangle is only tamed if it is a subset of a triad induced by the nodes N, N_1 and N_2. Therefore, $S = \{r_1, r_2\}$ and $S \cup z$ is tamed, another contradiction. The case $R = 3$ is trivial, because z has to share an arc colour with a vertex in r and a node colour with another, which leads to parallel arcs.

(16) The vertices returned here are exactly equal to the set S, and therefore this case is trivial.

We conclude that for every set S a correct representative set is returned. All lines except (12) and (18) can be trivially implemented in time $O(|S|)$ by looping through the vertices once. For line (12) we know that there exists a vertex v_2 that does not have N as a node colour. This vertex can be found in $O(|S|)$, either v_1, v_2 share an aiding arc colour or they can be returned. If they share an arc, then because $|\mathcal{A}_S| \geq 2$ we loop once more through the vertices and find another vertex v_3 that does not share a node and arc colour with either v_1 or v_2. In line (18), because all vertices lie in two nodes, but not a triad wild set is trivially found by returning two vertices that do not share any nodes by bookkeeping every combination of nodes when looping through S. □

5.2.2 Wild path generation

The next step is to use our representative sets and find a set Y that is $\mathbb{H}^\dagger$-wild. The idea is to start by calculating the representative sets for all single vertices in $Y = V \backslash X$. Denote with R_y the representative set of a vertex y, and by $\mathcal{R}_1 = \{R_y | y \in Y\}$. All these sets can be calculated in time $O(|Y||X|)$ from Theorem 5.1, and if one of them is wild we are immediately done. If none of them are wild, we need to find a vertex-minimal induced wild path. Clearly, looping through all sets of vertices in Y is too slow. We will show a different approach with a suitable running time. Note that the alternative approach for this sub-algorithm in [18] runs in time $O(n \log n)$, but relies heavily on results from [13]. The algorithm below is as all results in this paper self-contained and can be directly implemented with a running time of $O(n^2)$.

Theorem 5.2 (Wild path generation)**.** *Given a three-in-a-tree problem with graph G, terminals T, a web $\mathbb{H}$ and corresponding representative sets $R_y \in \mathcal{R}_1$ for all $y \in V \backslash X$, we can find a vertex minimal induced wild path using Algorithm 13 in time $O(n^2)$.*

Algorithm 13 Wild path generation

1: Obtain the graph induced by the vertices $Y = V \backslash X$.
2: Find a connected component induced by $Y_c \subseteq Y$, such that $N_X(Y_c)$ is wild.
3: Pick a random vertex $y_1 \in Y_c$, and let $Y_T = \{y_1\}$.
4: **while** the set Y_T is tamed **do**
5: Add a vertex to Y_T using breadth-first search and remember its parent vertex.
6: Calculate the new representative set for Y_T.
7: **end while**
8: Obtain the path $P_Y = y_1, y_2 ..., y_K$ from y_1 to the last vertex added to Y_T.
9: Let $R_T = R_{y_K}$ and $y = y_K$.
10: **while** the set R_T is tamed **do**
11: Let y be the next vertex in P_Y in reverse order
12: Update the representative set R_T as $R_T \cup N_X(y)$
13: **end while**
14: Let m be the index of the last y in the loop above, such that $y_m = y$.
15: **return** The sub-path of P_Y, $Y_{\text{wild}} = y_m, y_{m+1}, ...y_K$.

Proof. Again we will first discuss the correctness of the algorithm. We need to show three things. Namely, that the set Y returned by the algorithm is:

1. wild;
2. an induced path;
3. vertex minimal.

Because the algorithm finds a wild connected component Y_c, there clearly exists a subset of Y_c that is wild. Clearly, the last vertex added to Y_T can form a wild set together with one of the other vertices. Therefore a shortest path between two such vertices will form a wild set. A sub-path of this path then also exists that is wild.

We know that the path that is returned is a shortest path from y to y_K. Assume this is not an induced path. Then there is some edge connecting two vertices in the path, and the path is not a shortest path. Therefore an induced path is returned.

Lastly, we need to show vertex-minimality. Assume to the contrary that the returned path $y_m, y_{m+1}, ..., y_K$ is not vertex-minimal. This means that one of the induced paths $y_m, y_{m+1}, ..., y_{K-1}$ or $y_{m+1}, y_{m+2}, ..., y_K$ is also a wild path. If $y_m, y_{m+1}, ..., y_{K-1}$ is wild, then the set $Y_T \backslash y_K$ in the algorithm is also wild. Because the vertex y_K is the last vertex added to Y_T this is a contradiction. Clearly, from lines 10-12 of the algorithm it immediately follows that the path $y_{m+1}, y_{m+2}, ..., y_K$ is tamed. Therefore, we can conclude that the path is indeed vertex-minimal. Using some standard algorithms for finding connected components and breadth-first search the algorithm can clearly be implemented in time $O(n^2)$.

□

Using Algorithms 12 and 13 we are able to obtain a vertex minimal $\mathbb{H}^\dagger$-wild induced path in time $O(n^2)$. In the following sections we will check whether this set is podded or solid and show how to increase the size of our web if that is the case. Remember from Algorithm 10 that in all other cases we are immediately done.

5.3 Solid sets

Let Y_{wild} be the $\mathbb{H}^\dagger$-wild induced path obtain from Algorithm 13. We will first show how to check whether this set is solid. If $|Y_{\text{wild}}| \geq 2$ any straightforward implementation of the definition is sufficient, however we advise to take care when checking this for a single vertex. We simply propose the following algorithm that will work for both cases.

Theorem 5.3. *Given a three-in-a-tree problem with graph G, terminals T, a web $\mathbb{H}$ and a vertex minimal $\mathbb{H}^\dagger$-wild induced path Y_{wild}, we can determine in time $O(n^2)$ whether this set is solid using algorithm 14.*

Algorithm 14 Solid set check

if $|Y_{\text{wild}}| > 2$ **then**
  for every internal vertex y of the path Y_{wild} **do**
    if $N_X(y) \neq \emptyset$ **then**
      return false
    end if
  end for
end if
Let $S = N_X(Y)$.
Obtain all arc and node colors from the vertices in S, $\mathcal{N}_S$ and $\mathcal{A}_S$ respectively.
for every node $N \in \mathcal{N}_S$ **do**
  if the amount of vertices in S with node color N_c is equal to $|N|$ **then**
    Save node N as a solid set.
  end if
end for
for every arc $A \in \mathcal{A}_S$ **do**
  if there are exactly two vertices $v_1, v_2 \in S$ with arc colour A_c, A is a simple arc and v_1 is a neighbour of v_2 **then**
    Save arc A as a solid set
  end if
end for
if there are not exactly two saved solid sets **then**
  return false
end if
for each vertex $v \in Y$ **do**
  if v does not have a node or arc colour corresponding to one of the two solid sets **then return false**
  end if
end for
return true

Proof. Correctness of the algorithm should be clear as it is almost a direct implementation of the definition of a solid set. Regarding the running time, there are $O(n)$ possible arcs and nodes to check, and every check is just looping through $O(n^2)$ vertices and the running time is evidently as claimed. □

If Algorithm 14 determines that our wild set Y is in fact a solid set, we will increase the size of our web as proposed in Theorem 4.8. In order to do this, note that from Algorithm 14 we could also obtain the two solid sets S_1, S_2, such that S_j is the set of neighbours of

y_j for each end-vertex of Y. In the following algorithm the reason for maintaining paths A_p for all our simple arcs will also become clear.

Theorem 5.4. *Given a three-in-a-tree problem with graph G, terminals T, a web $\mathbb{H}$, a vertex minimal $\mathbb{H}^\dagger$-wild induced path Y_{wild} and two solid sets S_j, such that S_j are the neighbours of y_j for each end-vertex of Y, we can strictly increase the size by implementing Theorem 4.8 in time $O(n)$ using Algorithm 15.*

Algorithm 15 Solid set increase

1: **for** $j \in \{1, |Y|\}$ **do**
2: **if** S_j is a node of the web $\mathbb{H}$ **then**
3: Move vertex y_j into the web and give it the node colour of S_j.
4: **else**
5: Let $A = N_1N_2$ be the arc of which the vertices in S_j are a subset.
6: Create a new simple arc A'.
7: Let v be the vertex in A_p that lies in N_1, give it the new arc colour A'_c.
8: Remove v from A_p and add it to A'_p.
9: **while** v is not in S_j **do**
10: Let v be the next vertex in path A_p.
11: Recolour the arc colour of vertex v to A'_c.
12: Remove v from A_p and add it to A'_p.
13: **end while**
14: Create a new node N' with vertices $S_j \cup y_j$.
15: Set the Node-arc-intersection of N' with A_p and A' to 1.
16: Set the Node-arc-intersection of N_1 with A_p to 0, and with A' to 1.
17: **end if**
18: **end for**
19: Create another new simple arc A''.
20: Give all vertices in Y arc colour A''_c, and set its path as the induced path of Y.
21: Set the Node-arc-intersection of A'' with the nodes of the end-vertices of Y to 1.

Proof. Again, correctness of the Algorithm is immediately clear from Theorem 4.8 and its proof. For the running time we only need to mention that in line 12 removing the vertex v from A_p and adding it to A'_p takes time $O(1)$ because v is removed from and added to the end of the respective paths. This means that our while loop takes $O(n)$ time. Obviously all other steps can be implemented in time $O(n)$ as well. □

5.4 Podded sets

We will now move on to podded sets. While the algorithms below can determine for all wild sets whether they are podded, note that it is important to always first determine whether the wild set is solid. In particular, examples can be constructed where adding a set that is both solid and podded to the web using the algorithm below will lead to incorrect taming webs and mislabelling of three-in-a-tree problems. The algorithms here are based on the proof of Theorem 4.10 and its corresponding Lemmas.

Note that first of all we need to be able to obtain a smallest chunk $C = N_1N_2$ such that it is a pod of our wild set Y. In particular in contrast with the results in [18], we cannot rely on an incremental dynamic SPQR tree [13] over our graph which immediately gives such smallest chunks, but use a new and different approach to find a smallest chunk C containing a given set of arcs end nodes. An *articulation vertex* of a connected graph G is a vertex v such that $G - v$ is a disconnected graph.

We write H for the graph of the web $\mathbb{H}$, where its arcs are edges and its nodes are vertices. It is clear that the end-nodes of a chunk are articulation vertices n_1, n_2 in H, and there is a component H' of $H\backslash\{n_1, n_2\}$ for which the graph induced by $V_{H'} \cup \{n_1, n_2\}$ contains all nodes and arcs of $N_X(Y)$. In particular we need to find n_1 and n_2 such that there are no other vertices in $V_{H'} \cup\{n_1, n_2\}$ for which this holds. First we will show how to find articulation points of a graph, then we will obtain a minimum chunk and lastly we will test whether it is a pod of Y.

Theorem 5.5. *Given a three-in-a-tree problem with graph G, terminals T, a web $\mathbb{H}$ and a vertex minimal $\mathbb{H}^\dagger$-wild induced path $Y_{wild} = y_1y_2...y_K$, we can determine in time $O(n)$ whether this set has a $\mathbb{H}^\dagger$-pod $C = N_1N_2$ using Algorithm 16.*

Algorithm 16 Podded set check

```
1: if The neighbours of Y_wild do not all lie in C and its end-nodes N_1, N_2. then
2:     return false
3: end if
4: for end-nodes N ∈ {N_1, N_2} do:
5:     for end-vertex y ∈ {y_1, y_K} do
6:         Count the number μ(y) of neighbours of y that lie in node N, but not have
   aiding arc color C. I.e. determine |N_X(Y) ∩ N\C|.
7:     end for
8:     if μ(y) = 0 for both y or μ(y) = |N\C| for one y and 0 for the other then
9:         continue
10:    else
11:        return false
12:    end if
13: end for
14: return true
```

Proof. The correctness of the algorithm is evident by looking at the definition of a pod. For each end vertex of the wild path we count all neighbours in the nodes $N_1\backslash C$ and $N_2\backslash C$. Because we know the size of each node, and the size of the intersection of a node and an arc, we can calculate the size of each $N_i\backslash C$ $i = 1, 2$ and check whether the neighbours of Y either cover this set, or do not intersect with it at all.

Checking the running time, all loops just have two iterations. Counting the neighbours

of Y and checking their node and arc colours takes $O(n)$. Concluding for each end-vertex is done in time $O(1)$ and the running time is as claimed. □

Now that we can determine for a given chunk $C = N_1N_2$ whether this set is a pod of a wild set Y_{wild}, we want to generate minimal chunks that are potential pods in the aiding web. We do this in two steps, firstly we check whether an entire aiding arc is a pod, after which we generate a smallest chunk that still is a pod of Y. Note that first of all, every pod always contains one of the end-nodes of an aiding arc. And secondly, if the aiding arc is a pod of a set Y, there could exist a smaller pod, however surely there exists a smallest pod as the aiding arc itself is one. Using Algorithm 16 we can determine for every wild non-solid set whether it allows a pod. We simply determine if all neighbours lie in a single aiding arc and its end-nodes, after which we test whether those sets together are a pod of Y. We now turn to obtaining a smallest pod for our wild set. In order to find such smallest sets, we will rely on finding articulation points of our web.

Lemma 5.6. *For a given graph G we can obtain all articulation points in time $O(n^2)$.*

Proof. Running Tarjan's algorithm [23] will yield all articulation points in time $O(|n + m|) = O(n^2)$. □

Lemma 5.7. *Given a three-in-a-tree problem with graph G, terminals T, a web $\mathbb{H}$ and a set of nodes $\mathcal{N}_Y$ that all appear in a chunk $C = N_1N_2$ and either N_1 or N_2 appears in $\mathcal{N}_Y$, we can find the smallest chunk $C' = N_1'N_2'$, and all nodes between the end-nodes using Algorithm 17.*

Algorithm 17 Smallest chunk

1: Determine the end-nodes of C that are in $\mathcal{N}_Y$.
2: **if** both end-nodes are in $\mathcal{N}_Y$ **then**
3: **return** The entire chunk C.
4: **end if**
5: Let N_1 be the end-node of C that is in $\mathcal{N}_Y$, and N_2 the other end-node of the chunk C.
6: Construct the graph $H_{\mathcal{C}}$ where the nodes, arcs and node-arc intersections of the chunk C form the vertices and edges of the graph $H_{\mathcal{C}}$.
7: Determine the shortest paths from N_2 to all other nodes in the graph $H_{\mathcal{C}}$.
8: Let the node $N_Y^* \in \mathcal{N}_Y$ be closest to N_2, with distance d^*.
9: Determine the articulation point N_2' such that the distance $d(N_2, N_2')$ is the maximum distance that is still smaller than $d*$.
10: Remove the node N_2' from the graph $H_{\mathcal{C}}$.
11: Determine the set C' of all arcs that are still reachable from N_2'.
12: **return** chunk $C' = N_1N_2'$.

It is only for these smallest pods that we can guarantee correctness of the total three-in-a-tree algorithm. We turn to show how such wild sets with a minimal pod will increase the size of our web. The correctness of the algorithm already follows from 4.10, and therefore we only discuss the algorithm and its running time.

Theorem 5.8. *Given a three-in-a-tree problem with graph G, a web $\mathbb{H}$ and a vertex minimal $\mathbb{H}^\dagger$-wild induced path $Y_{wild} = y_1y_2...y_K$ with corresponding minimal pod C, we can strictly increase the size of our web while maintaining its properties by implementing Theorem 4.10 and Lemma 4.9 in time $O(n)$ as in Algorithm 18.*

Algorithm 18 Podded increase

1: Determine the end-node N of $C = N_1N_2$ that has no vertex neighbouring Y.
2: **if** N exists and is the end-node of more than one arc or a flexible arc of C **then**
3: Let A be the simple arc in C that has end-node N.
4: Find vertex $v_2 \in N_A(Y)$ closest to N_2 on A, and v_3 its neighbour on A further away from N_2.
5: Create a node N_3 with v_2 and v_3 and add the end vertex of Y which is their neighbour to it.
6: Update the arcs accordingly.
7: Let C' be the new minimal pod of Y .
8: **end if**
9: Merge the arcs of C' and add all vertices of Y to the new arc.

Proof. The algorithm is a direct implementation of Lemma 4.9 and Theorem 4.10 and their proofs show correctness of the algorithm. In order to see that the running time is correct, see that from Lemma 5.7 we know all arcs and nodes of our smallest chunk. The first few checks are therefore easily done in $O(n)$. Finding the vertex furthest away from N on A that is also a neighbour of Y takes $O(n)$ too. As last we need to update our arcs. However, this is easily done by counting the number of vertices on the resulting simple arc from N_3 to N. From this number we can update the arc sizes, and this takes only $O(n)$ time. Merging takes $O(n)$ time as well. □

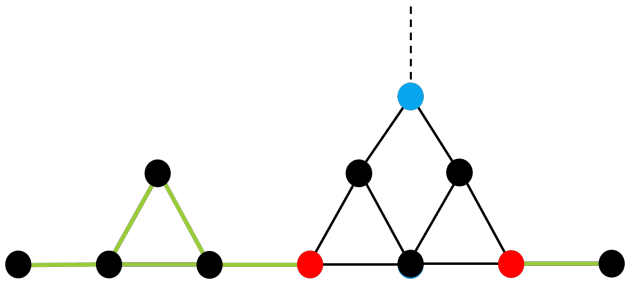

FIGURE 29: Graph representation of arcs and nodes inside a single arc of the aiding web. A new arc from a solid wild set has been added to the blue node. Red nodes indicate articulation points closest to the blue node in both directions. Green arcs will merged into new aiding arcs.

5.5 Maintaining the aiding web

During our algorithm when searching for wild sets, we always consider our aiding web $\mathbb{H}^\dagger$. This web is maintained during the entire algorithm. Firstly, note that during the first step of the Algorithm 10 the aiding web is always equal to the web of the graph. Then during the remainder of the algorithm, there are two parts where the aiding web could possibly change: when adding podded sets, or when adding solid sets. For our desired running time it is too slow to construct a new aiding web every time. Remember again that the aiding web in [18] is maintained as an incremental dynamic SPQR-tree during the algorithm, based on the result in [13], which was omitted to make the algorithm implementable. Therefore, we will consider a different way to maintain our aiding web. Again we will rely on the articulation points of subgraphs, using Tarjan's algorithm [23].

We first consider when a $\mathbb{H}^\dagger$-podded set is added to our web. By definition, this set only has neighbours in one aiding arc and its end-nodes. Clearly, after these vertices are added to an arc, they will always have the end-nodes of this aiding arc as split-nodes. Therefore, the aiding web never changes when an $\mathbb{H}^\dagger$-podded set is added to our web.

The second case is, however, a bit more difficult. Note for example the web and its aiding web in Figures 21 and 22. When an arc is added and we obtain the new web as in Figure 23. The aiding web has to change now because many arcs now have new split-nodes and new aiding arcs appear. In this case the aiding web becomes equal to the web, but this does not need to be the case in general.

We will now describe how to update our aiding web during the algorithm. In the following we will look at our updated web $\mathbb{H}_{\text{new}}$ after adding a solid set, and refer to our aiding web that still needs to be updated as web ${\mathbb{H}^\dagger}_{\text{old}}$. As seen in Algorithm 15, the solid $\mathbb{H}^\dagger$-wild set Y will form a new arc A. The newly added arc containing the vertices from the wild-set has two end-nodes N_1 and N_2 in the web $\mathbb{H}_{\text{new}}$. If both of these end-nodes are also nodes in the aiding web ${\mathbb{H}^\dagger}_{\text{old}}$, it is obvious to see that the aiding web does not change at all. On the other hand, if these end-nodes are internal nodes of an aiding arc, the web will always change. We consider the aiding arcs $A_1^\dagger$ and $A_2^\dagger$ of the internal nodes N_1 and N_2 separately, if they exist. In particular, all other aiding arcs will stay the same, because the split-nodes of the internal web components can not change. We can update the two aiding arcs separately, and the resulting aiding web will be the aiding web of $\mathbb{H}_{\text{new}}$.

Theorem 5.9. *Given an aiding arc $A^\dagger$ with end-nodes N_1, N_2, the set of all its arcs $\mathcal{A}'$ and one of its internal nodes N_{new} to which an arc A_{new}, consisting of the vertices of a solid $\mathbb{H}^\dagger$-wild set Y, has been added in Algorithm 15, we can update the part of the aiding web inside this aiding arc in time $O(n^2)$ using Algorithm 19.*

Algorithm 19 Aiding web maintenance

Let H' be the graph were all arcs in $\mathcal{A}'$ are edges and their end-nodes are vertices.
Find a shortest path from N_{new} to every vertex in H'.
Let P_1 be the path from N_{new} to N_1 and P_2 similarly to N_2.
Determine the set S of all articulation points in the graph.
if $N_{\text{new}} \in S$ **then**
    Let M_1 and M_2 be equal to N_{new}.
else
    Let M_1 be the articulation point on P_1 closest to N_{new}.
    Let M_2 be similar to M_1 on P_2.
end if
Create a new aiding arc $A^\dagger_{\text{new}}$.
for each arc $A \in \mathcal{A}'$ **do**
    if A lies the component between N_1 and M_1 **then**
        Remove A from $A^\dagger$ and add it to aiding arc $A^\dagger_{\text{new}}$.
        Update sizes of A, $A^\dagger$ and $A^\dagger_{\text{new}}$ and the aiding arc of A accordingly.
    else if A lies in the component between M_1 and M_2 **then**
        Create a new aiding arc that is just a copy of A.
        Update arc sizes and such accordingly.
    else if A lies in the component between M_2 and N_2 **then**
        Do nothing.
    end if
end for

Proof. An example of an aiding arc to which a new solid set has been added can be seen in Figure 29, while the structure cannot appear in this exact way in the web it will allow us to proof the theorem for almost all kinds of aiding arcs. The only thing that should be noted is that there are no non-trivial split-nodes in between the articulation points closest to the vertex A_{new}, because when podded sets are added to our web, chunks are also merged in the web $\mathbb{H}$.

First note that because M_1 is a articulation point, all arcs between M_1 and N_1 share the split-pair (M_1, N_1). This split-pair is also maximal, as otherwise another articulation point closer to N_{new} would exist. The same holds for M_2 and N_2, and therefore these two sets of arcs are correctly merged into chunks of the aiding web in the algorithm.

All other arcs in the web between the articulation points M_1 and M_2 do not share any split-nodes anymore after the arc A_{new} has been added to the web. Therefore all these arcs become arcs in the aiding web, and the algorithm correctly handles this.

Because we maintain the graph $H^\dagger$ as well as our actual aiding web $\mathbb{H}^\dagger$during our three-in-a-tree algorithm, the algorithm just works on a simple graph. Finding all shortest paths and determining articulation points is thus done in time $O(n^2)$. The components of the arcs can easily be found by removing articulation points of the graph and looping through the components. All updates to the arcs and aiding arcs are done in $O(1)$. In total our algorithm therefore take $O(n^2)$ time. □

Now that all algorithms are introduced, and our implementation strategies are discussed we turn to the main results about the running time of our complete algorithm.

Lemma 5.10 (Running time three-in-a-tree)**.** *We can implement Algorithm 10 in time $O(n^3)$ to find a three-in-a-tree for a given graph G and terminals T.*

Proof. First we obtain an initial web of our graph in time $O(n^2)$ as seen in Theorem 4.13. Now we turn to the main loop of our algorithm. Because the size of our web strictly increases, as seen in Theorems 4.8 and 4.10, we know that at least one vertex is added to the web in each iteration and the algorithm is done in $O(n)$ iterations.

During each iteration we first generate all representative sets of the vertices of the web, and find a vertex-minimal wild induced path as seen in Theorem 5.2 in time $O(n^2)$. Checking whether this set is $\mathbb{H}$-solid takes $O(n^2)$ time as seen in Theorem 5.3 and checking whether it is $\mathbb{H}$-podded takes $O(n)$ as in Theorem 5.5. Adding podded and wild sets can also be done in time $O(n^2)$ from Theorems 5.4 and 5.8.

After a wild set is added to the web we update our aiding web in time $O(n)^2$ as stated in Theorem 5.9. In conclusion, everything during one iteration is of time $O(n^2)$, and in the total $O(n)$ iterations we obtain an algorithm to find the three-in-a-tree in time $O(n^3)$. □

Theorem 5.11 (Running time improved pyramid detection)**.** *Given a graph $G = (V, E)$ we can determine in time $O(n^6)$ whether there is a subset $S \subseteq V$ that induces a pyramid.*

Proof. Follows from Lemma 5.10 and Theorem 4.15. □

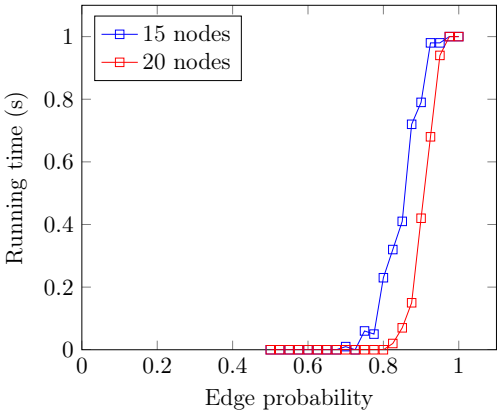

FIGURE 30: Fraction of graphs that is perfect for varying edge probabilities

6 Computational results

Both the implementation for the perfect graph detection and the three-in-a-tree algorithms that were discussed in this thesis have been implemented in Python 3.8. Because we believe the Berge graph detection from [6] has not been implemented before, we first discuss the results of that implementation. Next, we compare the original pyramid detection sub-algorithm to our new and improved implementation using a three-in-a-tree algorithm. Lastly, we discuss some interesting findings about the three-in-a-tree implementation in general.

In this section we again make use of Erdös-Rènyi random graphs, generated from a given edge-probability.

6.1 Perfect graph detection

Some of the running times for sub-components of the Berge graph detection algorithm were already shown in Section 3.4. These plots showed that the running times for several sub-algorithms are slower for graphs that have a slightly lower edge probability. Because we know that fully connected graphs are always perfect, while sparse graphs are mostly not perfect we check for which edge density our graphs tend to be perfect. Because graphs with multiple components are not of interest, we generate these random graphs such that they are always connected and consist of a single component.

In Figure 30 we see for graphs with a given number of nodes, but a varying edge-probability the amount of graphs that are perfect. For each edge-probability and number of nodes we ran 200 randomly generated test cases and plotted the fraction of those that are perfect. It turns out that there is a sharp and not too large region which is critical for the algorithm. In this region the graphs are either perfect or non-perfect, while graphs with low edge-probability are never perfect and (almost) complete graphs are of course always perfect.

6.2 Pyramid detection algorithms comparison

In order to check the performance of the three-in-a-tree algorithm for finding pyramids in comparison with the original sub-algorithm to detect pyramids we constructed more random graphs. We want to determine for which edge-probability both algorithm perform worst. For this, we generate random graphs with 15 and 20 nodes and varying edge-probability. The results are seen in Figures 31 and 32. Interestingly, the worst running

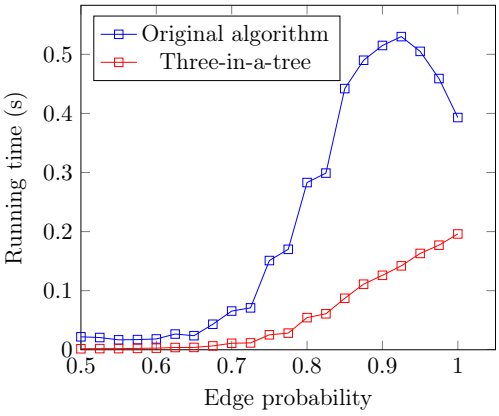

FIGURE 31: Average running times for pyramid detection in randomly generated graphs with 15 nodes.

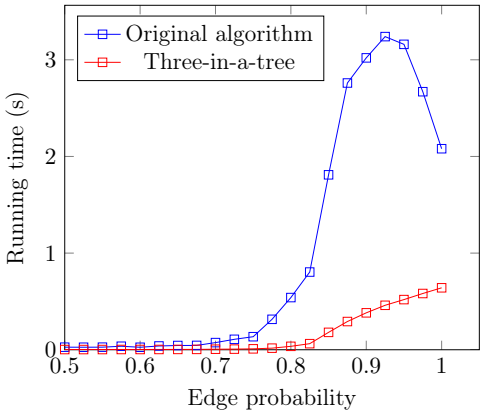

FIGURE 32: Average running times for pyramid detection in randomly generated graphs with 20 nodes.

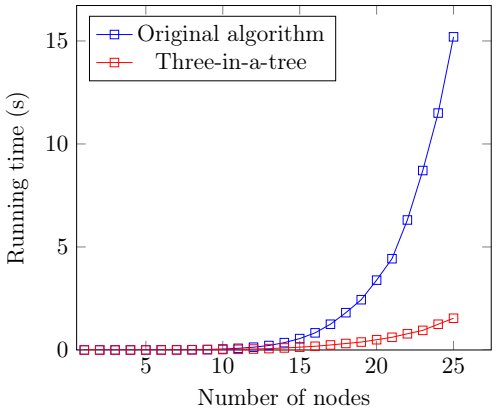

FIGURE 33: Average running times for pyramid detection in randomly generated graphs with edge-probability 0.9 and an increasing number of nodes.

times of the original algorithm correspond to the region of the edge-probabilities in which some but not all graphs are perfect as seen in Figure 30. The new algorithm based on three-in-a-tree performs increasingly worse with growing edge probability.

Of course, the edge-probabilities around which the graphs are sometimes but not always perfect are the most important for our pyramids detection. Therefore, we now fix our edge-probability on 0.9 and compare both algorithms for an increasing number of nodes as seen in Figure 33. In line with our expectations the new pyramid algorithm based on three-in-a-tree clearly outperforms the old algorithm.

As mentioned before, the running time for the algorithm that uses three-in-a-tree gets increasingly higher with increasing edge probability for a fixed number of nodes. Because the pyramid algorithms both rely on the generation of triangles in the graph, which are higher for denser graphs, it seems that the three-in-a-tree running time mostly depends on this factor. Therefore, we plot the number of triangles that are generated in random graphs in Figure 34 and the running time for both algorithms per triangle in a graph with 20 nodes in Figure 35. The running time of our new algorithm still grows with increasing edge-probability even per triangle.

The new algorithm to detect pyramids does not only rely on the three-in-a-tree algo-

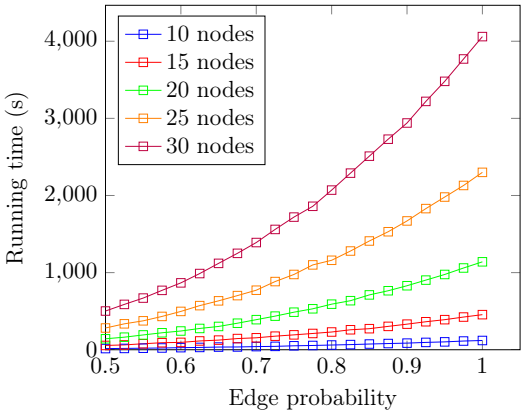

FIGURE 34: Average number of triangles in Erdös-Rènyi random with varying edge probabilities.

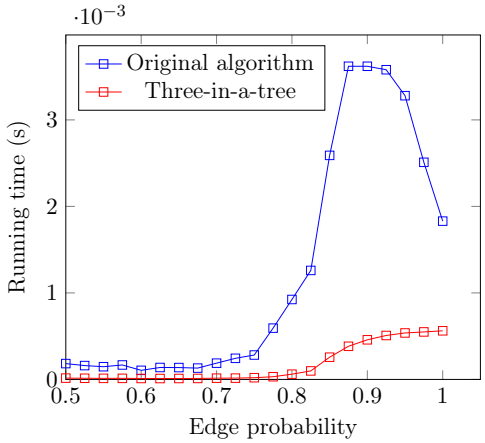

FIGURE 35: Average running time of two pyramid detection algorithms per triangle in a graph with 20 nodes

rithm. We also need to generate triangles in $O(n^3)$ time, and for each triangle change the graph such that we can run the three-in-a-tree algorithm in $O(n^2)$ time per triangle, as seen in Algorithm 11. We check for the effect on the running time of each of these parts. In Figures 36 and 37 we compare for graphs with 20 and 25 nodes the running time of these three different sub-algorithms.

Unexpectedly, the preprocessing algorithm is in average responsible for the majority of the running time for detecting pyramids. The keyword here is *average* because it turns out that the three-in-a-tree algorithm is very quick in almost all cases, which we will show in the next section.

6.3 Three-in-a-tree algorithm

The three-in-a-tree algorithm consist of one initial step, and is followed by multiple iterative steps. We want to determine in how many steps the algorithm is done in general, and how often it solves the problem in the initialization step alone. Finding solutions already in the initialization step only takes $O(n^2)$, and has a huge speed benefit over the original pyramid algorithm. While it seems that the algorithm can in practice indeed be often done in a small number of steps, theoretically the algorithm can of course take $O(n)$ steps. Therefore,

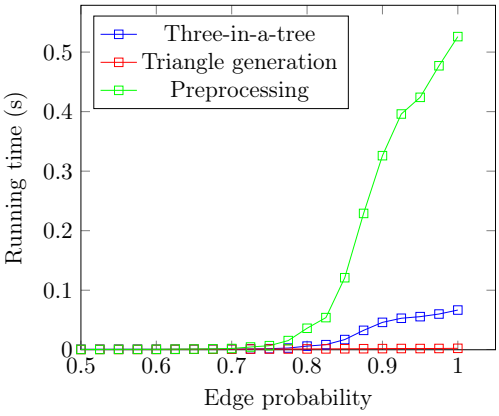

FIGURE 36: Running time of sub-algorithms for pyramid detection using three-in-a-tree in Erdös-Rènyi random with 20 nodes and varying edge probabilities.

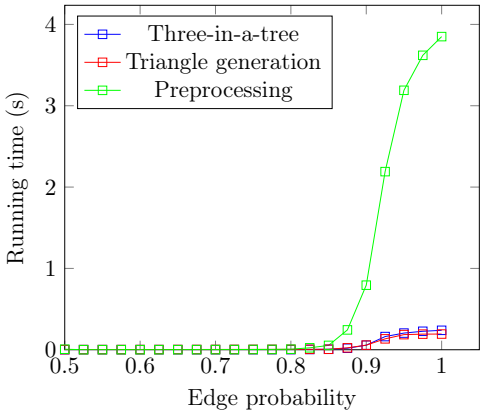

FIGURE 37: Running time of sub-algorithms for pyramid detection using three-in-a-tree in Erdös-Rènyi random with 30 nodes and varying edge probabilities.

TABLE 1: Number of iterations of the three-in-a-tree algorithm in 100000 random graphs

Number of nodes	init. step	1 it.	2 it.	3 it.	4 it.	5+ it.	max it. (#times)
20	19569	79185	204	158	136	748	17 (29)
25	19548	79272	186	131	134	729	22 (3)
30	19457	79294	204	165	134	746	27 (2)
35	19515	79311	208	138	114	715	31 (2)
40	19457	79401	169	142	121	710	34 (1)
50	19482	79350	171	153	120	754	38 (1)

we look at the average number of iterations for different amount of nodes. We generate 100000 random graphs with varying number of vertices. We choose an edge-probability of 0.925, as at this probability for all amounts of nodes the three-in-a-tree problem is not trivial, and the decision problems answer both yes and no in many cases. In each graph we pick 3 random vertices and run the three-in-a-tree algorithm. The number of steps that the three-in-a-tree algorithm takes to decide is seen in Table 1.

It becomes immediately clear why the algorithm seems to perform so well on average. In approximately 98% of the cases, the three-in-a-tree algorithm is done after a single iteration of searching for a wild set, which then either does not exist meaning the algorithm decides no, or is neither solid nor podded thus the algorithm directly decides yes.

7 Concluding remarks

In this thesis we introduced an implementable three-in-a-tree algorithm to accelerate perfect graph detection. The original perfect graph detection algorithm by Chudnovsky et al. as well as a new pyramid detection algorithm based on the three-in-a-tree algorithm by Lai et al. have been implemented. Because the new pyramid detection in time $O(n^6)$ ties the asymptotic running time with jewel, type T_2 and type T_3 detection, the forbidden substructure part of the algorithm has a running time of $O(n^6)$ now. The clear sole bottle-

neck is now the cleaning and detection of amenable holes, on which future research should be focused. Because this sub-algorithm seems nearly impossible to accelerate, a better approach would be to introduce new additional forbidden substructures, or generalise the current ones, leading to odd-holes that are easier to detect.

Our implementation to detect perfect graphs has an important consequence in the field of Mixed Integer Programming (MIP). For a perfect graph G, the LP relaxation of the set packing problem that corresponds to the stable set problem of the graph G has only integer solutions. In particular, when researching a set packing problem that seems to always yield integral solutions for its relaxation, we can test the graph of the corresponding stable set problem for perfection. If the graph is perfect and the set packing problem contains all cliques of the corresponding graph, this is enough to show integrality. This way, we can get an idea why a problem is integral and if the graphs are perfect try to prove that this is the case.

The results of our implementable three-in-a-tree algorithm are here used to accelerate perfect graph detection. The three-in-a-tree implementation can however be used for many other related induced subgraph problems. For example beetles can be found using a three-in-a-tree algorithm, which is part of the algorithm to detect even holes [4].

References

[1] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, oct 2005. ISSN 1549-6325. doi: 10.1145/1103963.1103966. URL https://doi.org/10.1145/1103963.1103966.

[2] C. Berge. Farbung von graphen, deren samtliche bzw. deren ungerade kreise starr sind. *Wissenschaftliche Zeitschrift*, 1961. URL https://cir.nii.ac.jp/crid/1573387450390873216.

[3] Dan Bienstock. On the complexity of testing for odd holes and induced odd paths. *Discrete Mathematics*, 90(1):85–92, 1991. ISSN 0012-365X. doi: https://doi.org/10.1016/0012-365X(91)90098-M. URL https://www.sciencedirect.com/science/article/pii/0012365X9190098M.

[4] Hsien-Chih Chang and Hsueh-I Lu. A faster algorithm to recognize even-hole-free graphs. *Journal of Combinatorial Theory, Series B*, 113:141–161, 2015. ISSN 0095-8956. doi: https://doi.org/10.1016/j.jctb.2015.02.001. URL https://www.sciencedirect.com/science/article/pii/S0095895615000155.

[5] Maria Chudnovsky and Paul Seymour. The three-in-a-tree problem. *Combinatorica*, 30(4):387–417, Jul 2010. ISSN 1439-6912. doi: 10.1007/s00493-010-2334-4. URL https://doi.org/10.1007/s00493-010-2334-4.

[6] Maria Chudnovsky, Gérard Cornuéjols, Xinming Liu†, Paul Seymour†, and Kristina Vušković‡. Recognizing berge graphs. *Combinatorica*, 25(2):143–186, Mar 2005. ISSN 1439-6912. doi: 10.1007/s00493-005-0012-8. URL https://doi.org/10.1007/s00493-005-0012-8.

[7] Maria Chudnovsky, Ken-ichi Kawarabayashi, and Paul Seymour. Detecting even holes. *Journal of Graph Theory*, 48(2):85–111, 2005. doi: https://doi.org/10.1002/jgt.20040. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jgt.20040.

[8] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. The strong perfect graph theorem. *Annals of Mathematics*, 164(1):51–229, 2006. URL http://www.jstor.org/stable/20159988.

[9] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. K4-free graphs with no odd holes. *Journal of Combinatorial Theory, Series B*, 100(3):313–331, 2010. ISSN 0095-8956. doi: https://doi.org/10.1016/j.jctb.2009.10.001. URL https://www.sciencedirect.com/science/article/pii/S009589560900080X.

[10] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. Detecting an odd hole, 2019.

[11] Maria Chudnovsky, Alex Scott, and Paul Seymour. Finding a shortest odd hole, 2020.

[12] Michele Conforti, Gérard Cornuéjols, Ajai Kapoor, and Kristina Vušković. Even-hole-free graphs part ii: Recognition algorithm. *Journal of Graph Theory*, 40(4):238–266, 2002. doi: https://doi.org/10.1002/jgt.10045. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jgt.10045.

[13] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components

with spqr-trees. *Algorithmica*, 15(4):302–318, Apr 1996. ISSN 1432-0541. doi: 10.1007/BF01961541. URL https://doi.org/10.1007/BF01961541.

[14] Reinhard Diestel. *Graph Theory*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[15] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, jun 1962. ISSN 0001-0782. doi: 10.1145/367766.368168. URL https://doi.org/10.1145/367766.368168.

[16] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, Jun 1981. ISSN 1439-6912. doi: 10.1007/BF02579273. URL https://doi.org/10.1007/BF02579273.

[17] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, jul 2001. ISSN 0004-5411. doi: 10.1145/502090.502095. URL https://doi.org/10.1145/502090.502095.

[18] Kai-Yuan Lai, Hsueh-I Lu, and Mikkel Thorup. Three-in-a-tree in near linear time. page 1279–1292, 2020. doi: 10.1145/3357713.3384235. URL https://doi.org/10.1145/3357713.3384235.

[19] Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[20] F. Roussel and P. Rubio. About skew partitions in minimal imperfect graphs. *Journal of Combinatorial Theory, Series B*, 83(2):171–190, 2001. ISSN 0095-8956. doi: https://doi.org/10.1006/jctb.2001.2044. URL https://www.sciencedirect.com/science/article/pii/S0095895601920441.

[21] Alex Scott and Paul Seymour. Colouring graphs with no odd holes. 10 2014.

[22] Jialei Song and Baogang Xu. On the chromatic number of a family of odd hole free graphs, 2021.

[23] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010. URL https://doi.org/10.1137/0201010.