Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Formal System Analysis Research Group

Optimising parity game solvers using dynamic SCC maintenance

Master Thesis

A.M. Huijsmans

Supervisors:
T.A.C. Willemse

Eindhoven, September 2021

Abstract

This thesis is about the optimization of a parity game solving algorithm. The parity game solving algorithm which will be optimized in this thesis is Zielonka's recursive algorithm with SCC decomposition. Two optimizations for this algorithm are investigated. The first optimization is partial re-decomposition, in which only the part of the graph containing vertices of SCCs which have 1 or more vertices removed will be re-decomposed. The second optimization is dynamic SCC maintenance, which builds an SCC tree for each SCC and then maintains those when vertices or edges are removed from the graph. An implementation in Java is made for 3 versions of Zielonka's algorithm: the first version is Zielonka's algorithm with Tarjan's algorithm as described in literature. The second version is Zielonka's algorithm with the use of partial re-decomposition. The third version is Zielonka's algorithm with dynamic SCC maintenance. The 3 versions are tested on various games. The conclusion from the tests is that Zielonka's algorithm with partial re-decomposition gives the best improvement.

Contents

Chapter 1

Introduction

In software engineering there is an increasing focus on the use of formal models for the specification and verification of software [3]. The promise that formal models brings to software engineering is that their use results in more reliable software. A technique is to model the behavior with a transition system like a Kripke structure. Figure 1.1 provides an example of such a model for the behavior of a double light switch system.

Such a transition system describes the different states of the system as nodes. Each of these nodes can have one or multiple labels such as a switch being on or off: the presence of S_1 when light 1 is on, or the absence when light 1 is off. Once such a model is made of the behavior, one can verify if all requirements on the behavior are fulfilled, using linear temporal logic (LTL). With LTL a requirement on the system can be transformed to a formula. For example, the requirement "The system shall not have the lights on if both switches are off", can be transformed to a formula `G(¬(L ∧ ¬`S_1` ∧ ¬`S_2`))`. In this formula `G`, which is a temporal operator, expresses that the formula must always hold, so in all states either the light is off or one of or both of the switches are on. LTL has many more of such temporal modal operators to be able to express a lot of different requirements, such as "nexttime": `X`, "sometimes": `F` etc[2]. An even more expressive formal language is μ-calculus, which adds a greatest fixed point operator (ν) and least fixed point operator (μ) [2].

In small systems such as the double light switch system it is easy to verify whether the requirements or formulas hold or not. We can check the states one by one or check all possible paths. But if there are a few million states and more complex requirements, then this is no longer possible. One way to check the correctness of the μ-calculus formulas is to transform the combination of Kripke structure and formula together into a parity game as shown in [16], which can be done in polynomial time. Such a parity game, an example is shown in Figure 3.1, is a 2 player turn based game. In such a game each vertex is won by exactly one player. Section 3.1 will elaborate more on how exactly parity games work. By solving the parity game it is possible to determine whether the μ-calculus formula holds for a state in the transition system.

As μ-calculus formulas can be transformed to parity games in polynomial time, it is beneficial to formal model checking to be able to solve parity games very fast. There are many algorithms to do this. One of these algorithms is Zielonka's algorithm [6], which will be used in this thesis to solve the parity games. This algorithm will be discussed in section 3.4. Some of the parity game solving algorithms, including Zielonka's Algorithm, can be sped up using a decomposition of the game into strongly connected components (SCCs), sets of vertices such that each vertex in the set can reach each other vertex. Chapter 3 will give an introduction to parity games and SCCs as well as the parity game solving algorithm Zielonka's algorithm and the SCC decomposition algorithm Tarjan's algorithm.

This thesis will focus on optimizing the SCC decomposition part of parity game solvers, with Zielonka's algorithm in particular, using dynamic SCC maintenance or partial re-decomposition. Chapter 4 will give a more detailed description of the research questions answered in this thesis. The main research questions are: what is the impact in practice of using an existing algorithm

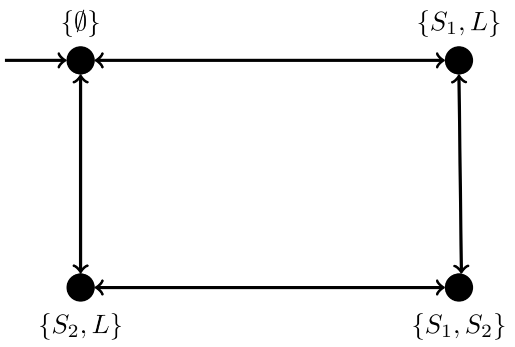


Figure 1.1: A Kripke structure of a light system with 2 switches S_1 and S_2. If S_1 or S_2 holds in a state, then that switch is on. If L holds in a state then the light is on.

for dynamic SCC maintenance with Zielonka's algorithm? and what is the impact in practice of using partial re-decomposition with Zielonka's algorithm.

Partial re-decomposition is a heuristic we develop to speed up the repeated SCC decomposition in Zielonka's algorithm and will be described in detail in chapter 5. In short the idea is that when at the end of an iteration of Zielonka's algorithm the whole leftover graph is decomposed into SCCs, it may be faster to only decompose the vertices of SCCs of which vertices have been removed into SCCs.

Dynamic SCC maintenance [9] is an algorithm that first makes a dynamic SCC decomposition which consists of an SCC tree for each SCC in the game. The maintenance of this dynamic SCC decomposition consists of an algorithm which can update these trees to delete edges or vertices. The dynamic SCC maintenance could replace the repeated SCC decomposition in Zielonka's algorithm and update the dynamic SCC decomposition instead. Chapter 6 explores all details about dynamic SCC maintenance that were investigated for this thesis in order to implement and use this mechanism in Zielonka's algorithm in practice.

The experiments executed to answer the research questions are described in chapter 7. They test for good ways to build the SCC trees, testing the decrease in the number of vertices decomposed by partial re-decomposition and testing the impact in practice of partial re-decomposition and dynamic SCC maintenance by measuring the runtime.

The last chapter, chapter 9, gives a summary of the answers to the research questions and discusses several threats to validity. It will show that using partial re-decomposition with Zielonka's algorithm has a positive impact. Especially when Zielonka's algorithm has many iterations. The number of vertices decomposed into SCCs is sometimes even decreased by more than a factor 100. Using dynamic SCC maintenance with Zielonka's algorithm most of the time had a negative impact. The dynamic decomposition of a lot of games contain very long SCC trees, and especially for these games using dynamic SCC maintenance has a big negative impact as more time is spend building the trees than solving the game. For games whose dynamic decomposition contains short trees, and Zielonka's algorithm has a lot of iterations, the use of dynamic SCC decomposition still had a positive impact, but only compared to running Zielonka's algorithm normally where the whole leftover graph is decomposed into SCCs at the end of each iteration of Zielonka's algorithm. When compared to partial re-decomposition, dynamic SCC maintenance is always slower.

Chapter 2

Related Work

There are many algorithms capable of solving parity games, each with their own tactics and complexity. The algorithm used in this thesis, Zielonka's algorithm, was tested to be the best parity game solving algorithm in practice by Friedman and Lange in [4]. In this paper they tested 3 algorithms, namely Zielonka's algorithm, the small progress measures algorithm of Jurdziński [7] and the strategy improvement algorithm [15]. This paper made a generic solver in which the parity game was decomposed into SCCs and each final SCC was solved with any of the algorithms. These were then implemented in a tool named PGSolver. With the PGSolver tool the runtime results were measured for several different sets of games. These results showed that Zielonka's recursive algorithm was much better than the other 2 algorithms in solving large parity games efficiently in practice. Another observation made was that SCC decomposition was highly profitable for any of the algorithms. Although this thesis only tests the effect of dynamic SCC maintenance on Zielonka's algorithm, it might be possible for these other algorithms which profit from SCC decomposition to also profit from dynamic SCC maintenance.

Van Dijk also did practical experiments with several algorithms in [14]. In this paper Zielonka's algorithm and the priority promotion algorithm were compared with several variations of the newly introduced Tangle learning algorithm. The Tangle learning algorithm as described in [14] makes use of the notion of tangles. A tangle is an SCC such that a player $\alpha \in \{0, 1\}$ has a strategy to win all cycles in the tangle and therefore the other player is forced to escape from this tangle if she is to win vertices in the tangle. For the comparison between the algorithms, the various algorithms were implemented in C++ in a parity game solver named Oink. Each algorithm was then tested against a set of Random games and a set of model checking and equivalence checking games. The results showed that Zielonka's algorithm and the priority promotion algorithm performed better for the model checking and equivalence checking games, but worse for large random games.

Di Stasio et al introduced several improvements to the implementation of Zielonka's algorithm in [12]. These improvements saved up to two orders of magnitude in running time compared to the formerly described PGSolver tool [4]. Their main improvement was the change of programming language. Instead of the OCaml language, they used the Scala programming language. This language allowed them to make a heavily optimized implementation. They tested several games produced by the PGSolver tool, such as random games, clique games, ladder games and Jurdzinksi's games, both with the PGSolver tool and the new implementation in Scala. The results of these tests showed the improvements in running time up to two orders of magnitude. The implemented version of Zielonka's algorithm in this Scala implementation does not make use of SCC decomposition however. They claim that "these improvements often do not show the desired performance".

Considering SCC related literature, Bernstein in [1] introduced an improved version of the in this thesis discussed dynamic SCC maintenance algorithm. Although the algorithm is more complicated, it has an update time of $\widetilde{O}(m)$, so near linear as $\widetilde{O}$ suppresses logarithmic factors. The algorithm makes use of so called GES-trees and uses these GES-trees to maintain SCCs. Although our findings point out that the dynamic SCC maintenance algorithm from [9] does not seem to have a positive influence on the performance of Zielonka's algorithm, perhaps this algorithm from [1] with improved update time could speed up the updates. However building the GES-trees still takes $O(m{*}\log^2 n)$ time on each level, so it not unlikely that still too much time is spent building the GES-trees at the start of Zielonka's algorithm.

Chapter 3

Preliminaries

The thesis focuses on using a dynamic strongly connected component maintenance algorithm to optimize parity game solving algorithms. This chapter starts with an explanation of what parity games are in section 3.1. Then in section 3.2 we explain what strongly connected components (SCCs) are and we discuss an algorithm, Tarjan's algorithm [13], which can decompose a graph into SCCs in section 3.3. The last section, section 3.4, will discuss Zielonka's algorithm, which is an algorithm which can solve parity games.

3.1 Parity Games

Definition 1 (Parity game)**.** A parity game is a structure $G = (V, E, pr, (V_0, V_1))$. In the structure, V is the set of vertices of the graph and E the set of edges such that $\forall_{v \in V} \exists_{w \in V} : (v, w) \in E$. In this graph every vertex $v \in V$ has a priority $pr(v) \in \{0, 1, ..., d\}$ and is owned by either player 0 or player 1. $V_0 \subseteq V$ contains the vertices owned by player 0, and $V_1 \subseteq V$ contains the vertices owned by player 1.

We visualize a parity game as a graph, where vertices owned by player 1 will be presented as a square and vertices owned by player 0 will be presented as a circle. The priority will be denoted inside the vertex and the name of the vertex just outside the vertex. See, e.g. Figure 3.1 where vertices v_1, v_3 and v_4 are owned by player 1 and vertices vertices v_2 and v_5 are owned by player 0. Vertex v_3 in this figure has priority 6.

The priority of a set of vertices $U \subseteq V$ is the highest priority of any vertex $v \in U$, so $pr(U) = max\{pr(v)|v \in U\}$.

In a game a token is placed on a starting vertex $v \in V$ after which the token is passed around indefinitely.

Definition 2 (Play)**.** A play is an infinite path of vertices $v_0, v_1, v_2...$ such in which the token is passed from vertex v_i to v_{i+1} for $i \in \mathbf{N}$, and for each v_i, v_{i+1} we have that $(v_i, v_{i+1}) \in E$.

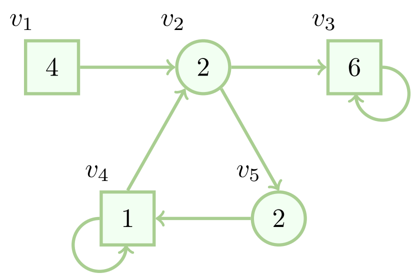


Figure 3.1: A parity game

Definition 3 (Winner of a play)**.** The winner of a play is determined by the highest priority that occurs infinitely often in the play. If this highest priority is even, the play is won by player 0. If this highest priority is odd, then the play is won by player 1.

An example of a play would be, $v_1, (v_2, v_5, v_4)^\omega$ which is a play over the game in Figure 3.1. For this play we have that priorities 1 and 2 occur infinitely often, 2 is the highest priority therefore the winner of this play is player 0. Priority 4 is not considered since it does not occur infinitely often. At each position in the play the next vertex is decided by the owner of the current vertex the token is at. So if the token is at vertex v_2 in Figure 3.1 then player 0 gets to decide whether to pass the token to v_3 or v_5. This choice is called a strategy.

Definition 4 (Strategy)**.** A memoryless strategy σ_α for player $\alpha \in \{0, 1\}$ is a partial function $\sigma_\alpha : V_\alpha \to V$.

A play conforms to a strategy if all choices in the play are made according to the strategy, so if for a player $\alpha \in \{0, 1\}$ we have that if $v_j \in V_\alpha$ is in the play and $\sigma_\alpha(v_j)$ is defined, then $v_{j+1} = \sigma_\alpha(v_j)$. If $\sigma_\alpha(v_j)$ is not defined then $(v_j, v_{j+1}) \in E$ is accepted. We denote all plays starting in a vertex $v \in U$, $U \subseteq V$ that conform to a strategy σ as $Plays(U, \sigma)$.

Definition 5 (Winning strategy)**.** A winning strategy σ_α for a node $v \in V$ and a player $\alpha \in \{0, 1\}$ is a strategy such that every play in $Plays(\{v\}, \sigma_\alpha)$ is won by player α.

Definition 6 (Winner)**.** Player α is the winner of a game G starting in vertex v if there exists a winning strategy σ_α for G starting at v.

The computation of a winner can be used to solve the parity game. Solving a parity game means computing the winner of the game for every vertex $v \in V$. To compute the winner of the game for every vertex the computation of the attractor set is needed. An attractor set of a set of vertices $U \subseteq V$ for a player $\alpha \in \{0, 1\}$ contains all vertices such that if the vertex belongs to player α then player α can move towards U and if the vertex belongs to player $1 - \alpha$ then that player has no choice but to move towards U. E.g. in Figure 3.1 the attractor set of $U = \{v_2, v_5\}$ for player 0 contains vertices v_1, v_2 and v_5. In vertex v_1 player 1 can only move towards $v_2 \in U$. Vertex v_4 for example is not part of the attractor set because player 1 can choose to keep moving to v_4 and stay away from U. The the attractor set can be defined formally as follows [4]:

Definition 7 ($Attr^G_\alpha(U)$)**.** The attractor $Attr^G_\alpha(U)$, of player α towards set $U \subseteq V$ is defined as $Attr^G_\alpha(U) := \bigcup\limits_{k \in N} Attr^k_\alpha(U)$, where:

$$\begin{aligned} Attr^0_\alpha(U) :=\ & U \\ Attr^{k+1}_\alpha(U) :=\ & Attr^k_\alpha(U) \cup \{v \in V_\alpha | \exists_{w \in Attr^k_\alpha(U)} : (v, w) \in E\} \cup \\ & \{v \in V_{1-\alpha} | \forall_{w \in V} : (v, w) \in E \implies w \in Attr^k_\alpha(U)\} \end{aligned}$$

Sometimes while calculating the attractor set, the algorithm which uses the attractor set is also needs to know in the strategy which α uses to get to or stay in U. In this case every time a new vertex $v \in V_\alpha$ is added to the attractor set, the $w \in U$ for which $(v, w) \in E$ is added to the strategy. In this document $Attr^0_\alpha(U)$ is used in 2 ways, either with return argument the set of vertices in the attractor set when the algorithm is not interested in the strategies, or with return argument the set of vertices in the attractor set and the strategy for player α.

Another property of the attractor set is that when an attractor set A is removed from a parity game G, then the remainder is still a valid parity game: $G \backslash A$ is a parity game [17], where $G' = G \backslash A = (V', E', pr, (V'_0, V'_1))$ is the game G' such that all vertices in A have been removed from G and all edges with an endpoint in A have been removed as well. So $V' = V \backslash A$, $E' = \{(v, w) \in E | v \notin A \wedge w \notin A\}$, $V'_0 = V_0 \backslash A$ and $V'_1 = V_1 \backslash A$.

Definition 8 (dominion[14])**.** A dominion $D \subseteq V$ is a set of vertices for a player $\alpha \in \{0, 1\}$ such that α has a winning strategy σ_α such that all plays consistent with σ_α stay in D .

A game is a paradise for player $\alpha \in \{0, 1\}$ if player α has a winning strategy for every vertex $v \in V$.

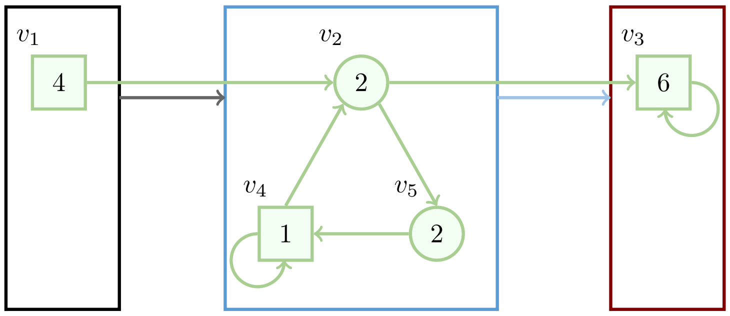


Figure 3.2: The topological order of the SCCs of a parity game

3.2 Strongly connected components

A graph can be partitioned in strongly connected components (SCCs). An SCC is a maximal non empty subset of vertices such that any vertex in the subset can reach any other vertex in the subset. Or formally: a maximal subset $C \subseteq V$ such that $\forall_{v,w \in C} : vE^*w$ where E^* is the reflexive transitive closure of E. In Figure 3.2, for example, we would have the SCCs: $C_1 = \{v_2, v_4, v_5\}$(black), $C_2 = \{v_1\}$ (blue) and $C_3 = \{v_3\}$ (red).

SCCs can be topologically ordered such that for two SCCs C_i and C_j, $i \neq j$ we have that $C_i \rightarrow C_j$ iff $\exists_{v \in C_i, w \in C_j} : (v, w) \in E$. A final SCC is an SCC in the topological order which does not have an outgoing edge. Every graph must have at least one final SCC, because if not then the graph would be one big SCC, which would make it a final SCC in itself.

Example 1. As an example of a topological order of SCCs take the game in Figure 3.2. As mentioned before it has SCCs: $C_1 = \{v_1\}$ (black) , $C_2 = \{v_2, v_4, v_5\}$ (blue), and $C_3 = \{v_3\}$ (red). The vertex v_1 in SCC C_1 (black) has an edge to vertex v_2 in SCC C_2 (blue). Therefore in the topological order C_1 (black) is placed before C_2 (blue). The vertex v_2 in SCC C_2 (blue) has an edge to vertex v_3 in SCC C_3 (red). Therefore in the topological order C_2 (blue) is placed before C_3 (red). So the game of Figure 3.2 would have the following topological order: $C_1 \rightarrow C_2$ (black $\rightarrow$ blue) and $C_2 \rightarrow C_3$ (blue $\rightarrow red$). The only SCC that does not have a vertex with an edge to another vertex outside the SCC is SCC C_3 (red), so this SCC is a final SCC.

SCCs are useful for parity games because it is possible to solve parity games SCC wise. An SCC C of a parity game G is also a parity game in itself: $G \cap C$ is a parity game. [6].

Definition 9 $(G \cap C)$**.** If $G = (V, E, pr, (V_0, V_1))$ and $C \subseteq V$ is an SCC, then $G' = G \cap C = (V', E', pr, (V_0', V_1'))$ with $V' = \{v \in V | v \in C\}$, $E' = \{(v, w) \in E | (v, w) \in C\}$, $V_0' = \{v \in V_0 | v \in C\}$ and $V_1' = \{v \in V_1 | v \in C\}$.

Every game will eventually get trapped in an SCC and the winner will also depend only on the priorities in the SCC. This information can be used to speed up algorithms as SCC decomposition uses only $O(|E|)$ time [4]. It is for example used to speed up Zielonka's Recursive algorithm in [6] or as an optimization in a generic solver in [4]. Zielonka's algorithm with the use of SCCs will be further discussed in section 3.4.

3.3 Tarjan's Algorithm

One of the simplest and most efficient algorithms to decompose a graph G in SCCs is Tarjan's algorithm [13]. The algorithm which works similarly to a depth first search (DFS). The algorithm will be discussed in more detail in appendix A which also includes the pseudocode.

Theorem 1. The recursive version of Tarjan's algorithm, `Tarjan`(G), returns the SCC decomposition of the graph G in $O(|V| + |E|)$ time.

This theorem has been proven by R.E. Tarjan in [13].

One of the possible problems with running Tarjan's algorithm on big graphs in practice is that there might be a lot of recursions of the algorithm stored on the program stack at the same time. To solve this W.R.M. Schols described an iterative version of Tarjan's algorithm in his master thesis [11] and showed its correctness. The pseudocode of the iterative version of Tarjan's algorithm is included in appendix A.

Theorem 2. The iterative version of Tarjan's algorithm, `TarjanItter(`G`)`, returns the SCC decomposition of the graph G in $O(|V| + |E|)$ time.

In [11] W.R.M. Schols has proven that the iterative version works correctly and just like the recursive version also has a complexity of $O(|V| + |E|)$.

3.4 Zielonka's Recursive Algorithm

There are a lot of different algorithms which solve parity games using different tactics. One of these parity game solvers is Zielonka's recursive algorithm [6] which will be described in this section. Zielonka's recursive algorithm is proven to have an exponential worst case complexity of $O(mn^d)$, where m is the number of edges, n the number of vertices and d the number of different priorities [6]. There are several other parity games with a better complexity than Zielonka's algorithm, such as for example the small progress measures algorithm by M. Jurdzinski with a complexity of $O(dm \cdot (\frac{n}{\lfloor d/2 \rfloor})^{\lfloor \frac{d}{2} \rfloor})$ [7]. However O. Friendman and M. Lange showed in [4] that in practice Zielonka's recursive algorithm is faster than other parity game solving algorithms. An improved version of the algorithm with SCC decomposition is shown to run in polynomial for several special classes of "simple" parity games for which the original algorithm would require exponentially many recursive calls [6].

The Zielonka's recursive algorithm without SCC decomposition is shown in Algorithm 1. It is a divide and conquer algorithm that makes use of the fact that higher priorities dominate lower priorities and that forced revisit is beneficial to the player with the same parity as the priority. It constructs winning regions for both players out of the solution of subgames with fewer different priorities and fewer vertices and removes the vertices with the highest priority from the game and its attractor set.

Example 1. When Zielonka's Algorithm is run on the parity game shown as the left-most graph in Figure 3.3 the following happens. The highest priority is 3, so vertices with this priority are beneficial for player 1. The vertices with this priority are v_3 and v_4. The vertices that are attracted to these 2 vertices by player 1 are $\{v_0, v_1, v_3, v_4\}$. So the algorithm recurses on the parity game from which these vertices have been removed. This leftover parity game is shown in the middle graph of Figure 3.3 as $G\backslash A$. In this game $G\backslash A$ the highest priority is 2, and since all vertices belong to player 0 all vertices are added to the attractor set. After recursing on an empty parity game the algorithm will return that $\{v_2, v_5, v_6, v_7\}$ are all won by player 0 for game $G\backslash A$. Then back in the algorithm for game G the attractor set of $\{v_2, v_5, v_6, v_7\}$ shows that player 0 can have a strategy such that for all vertices except v_1 it can send the game to $\{v_2, v_5, v_6, v_7\}$ and thereby win the play. The algorithm therefore recurses on the subgame of just v_1 which is shown as the rightmost graph in Figure 3.3 as $G\backslash B$. This game has highest priority 1 and is the only vertex so the algorithm returns that this vertex is won by player 1. The algorithm on graph G now knows this vertex v_1 is won by player 1 and already knew all other vertices are won by player 0 so it reports this as the answer. So the vertices in the set $\{v_0, v_2, v_3, v_4, v_5, v_6, v_7\}$ are won by player 0 and the vertices in the set $\{v_1\}$ are won by player 1.

Zielonka's recursive algorithm with SCC decomposition, described by Gazda and Willemse in [6], is shown in Algorithm 2. It first decomposes the game into SCCs and then uses the steps Zielonka's algorithm performs on the final SCCs. Afterwards the final SCCs and their attractor sets are removed, breaking the SCCs down in smaller SCCs. Because each infinite play will eventually stay in an SCC, the winner of such play is determined by only vertices of that SCC. So

Algorithm 1 Zielonka's recursive algorithm

```
function ZIELONKA($G = (V, E, pr, (V_0, V_1))$)
    if $V = \emptyset$ then
        $(W_0, W_1) \leftarrow (\emptyset, \emptyset)$
    else
        $m \leftarrow max\{pr(v) | v \in V\}$
        $\alpha \leftarrow m \bmod 2$
        $\overline{\alpha} \leftarrow 1 - \alpha$
        $U \leftarrow \{v \in V | pr(v) = m\}$
        $A \leftarrow Attr^G_\alpha(U)$
        $(W'_0, W'_1) \leftarrow$ ZIELONKA($G \backslash A$)
        $B \leftarrow Attr^G_{\overline{\alpha}}(W'_{\overline{\alpha}})$
        if $B = W'_{\overline{\alpha}}$ then
            $(W_\alpha, W_{\overline{\alpha}}) \leftarrow (A \cup W'_\alpha, B)$
        else
            $(W'_0, W'_1) \leftarrow$ZIELONKA($G \backslash B$)
            $(W_\alpha, W_{\overline{\alpha}}) \leftarrow (W'_\alpha, W'_{\overline{\alpha}} \cup B)$
        end if
    end if
    return $(W_0, W_1)$
end function
```

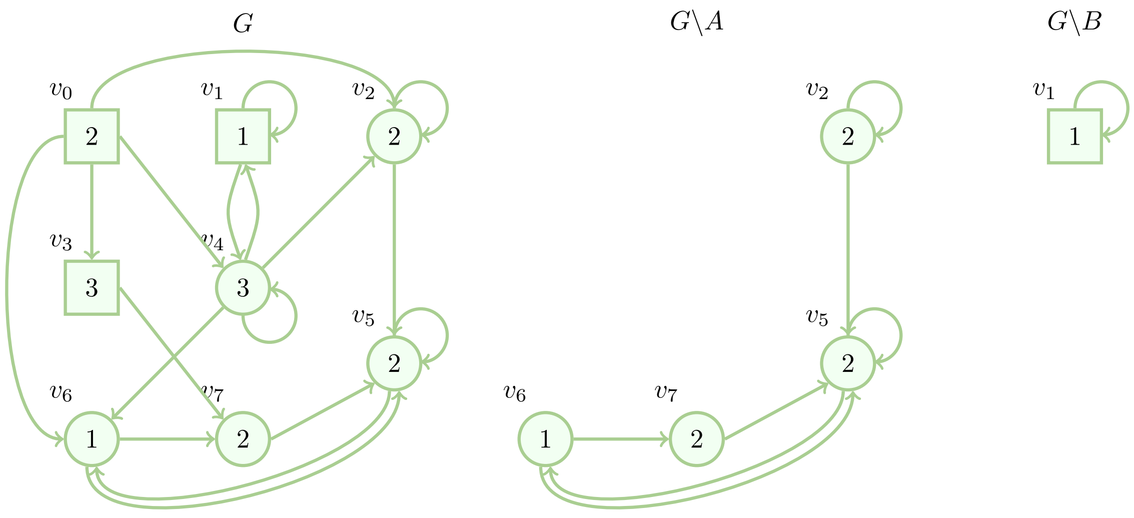


Figure 3.3: Different stages of running Zielonka's recursive algorithm on parity game G

first looking at the winning regions of final SCCs from which plays cannot leave and their attractor sets should give a much more efficient algorithm. An attractor set for a player α that stays within an SCC H and thus can only include vertices that are present in H is denoted by $Attr^H_\alpha$.

Algorithm 2 Optimized Zielonka's recursive algorithm[1]

```
1: function ZIELONKA_SCC(G)
2:     $(W_0^G, W_1^G) \leftarrow (\emptyset, \emptyset)$
3:     while $G \backslash (W_0^G \cup W_1^G) \neq \emptyset$ do
4:         $S \leftarrow$ SCC_GRAPH_DECOMPOSITION($G \backslash (W_0^G \cup W_1^G)$)
5:         for all final SCC $C \in S$ do
6:             $H \leftarrow G \cap C$
7:             $m \leftarrow max\{pr(v) | v \in C\}$
8:             $\alpha \leftarrow m \bmod 2$
9:             $\overline{\alpha} \leftarrow 1 - \alpha$
10:            $U \leftarrow \{v \in C | pr(v) = m\}$
11:            $A \leftarrow Attr^H_\alpha(U)$
12:            $(W'_0, W'_1) \leftarrow$ ZIELONKA_SCC($H \backslash A$)
13:            $B \leftarrow Attr^H_{\overline{\alpha}}(W'_{\overline{\alpha}})$
14:            if $B = W'_{\overline{\alpha}}$ then
15:                $(W_\alpha, W_{\overline{\alpha}}) \leftarrow (A \cup W'_\alpha, B)$
16:            else
17:                $(W'_0, W'_1) \leftarrow$ZIELONKA_SCC($H \backslash B$)
18:                $(W_\alpha, W_{\overline{\alpha}}) \leftarrow (W'_\alpha, W'_{\overline{\alpha}} \cup B)$
19:            end if
20:            $(W_0^G, W_1^G) \leftarrow (W_0^G \cup W'_0, W_1^G \cup W'_1)$
21:        end for
22:        $(W_0^G, W_1^G) \leftarrow (Attr^G_0(W_0^G), Attr^G_1(W_1^G))$
23:    end while
24:    return $(W_0^G, W_1^G)$
25: end function
```

Example 2. As an example where using SCC decomposition might increase the effectiveness of Zielonka's recursive algorithm, the flow of the algorithm with SCC decomposition is shown in Figure 3.3. The first SCC decomposition when running Zielonka's recursive algorithm with SCC decomposition on parity game G of Figure 3.3 is shown in the leftmost game in Figure 3.4. The algorithm starts with SCC decomposition. Each SCC in Figure 3.4 is given its own color. The only final SCC is the one with the purple color, so this one is picked for the first iteration and shown in isolation as H_1. The highest priority is 2 and since all vertices of H_1 belong to player 0 all vertices are attracted. The algorithm then checks for all of G which vertices are attracted to the vertices won by either player in H_1. Since player 1 did not win any vertices none are attracted, but for player 0 vertices $v_0, v_2, v_3, v_4, v_5, v_6$ and v_7 are. So these vertices are all added as winning vertices for player 0. After all these vertices have been removed from G this leaves only v_1 which is just a single SCC called H_2. This vertex is won by player 1 and attracts only itself. After removal of this vertex the graph will be empty so the algorithm is done.

Even in this small example first looking at vertices with the highest priority in the final SCC instead of the overall graph, already saved some computations. The algorithm without SCC decomposition was executed 3 times, not counting the recursions on empty games. The algorithm with SCC decomposition executed the same calculations only twice, once for the final SCC and once for that last vertex.

[1]Some changes have been made to the algorithm starting from this line which speeds up the algorithm with several magnitudes. These changes are trivial and most likely intended by the author of the paper containing the algorithm.

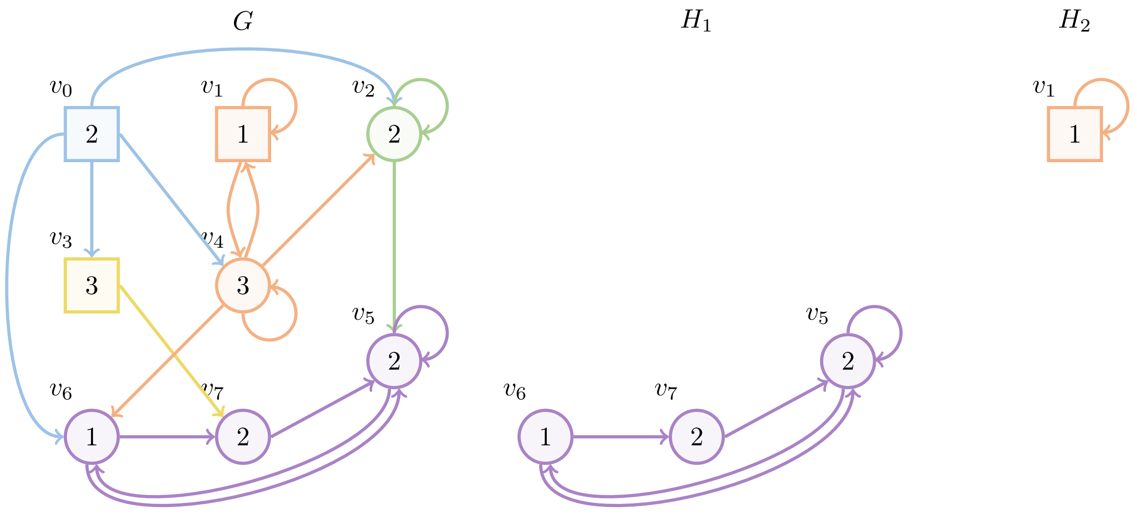


Figure 3.4: Different stages of running Zielonka's recursive algorithm with SCC decomposition on parity game G. Each SCC is shown in a different color.

Chapter 4

Problem statement

In Zielonka's algorithm optimized with the SCC decomposition, in each iteration and recursion the graph is decomposed into SCCs. Can Zielonka's algorithm be improved by decomposing in a smarter way? This idea is the basis for the research questions answered in this thesis. This chapter defines those research questions.

In Zielonka's recursive algorithm the addition of SCC decomposition is useful for speeding up the algorithm. The graph is decomposed in SCCs at 2 points in the algorithm. First at the start of the algorithm to find the SCCs. Secondly, at the end of each iteration of the loop when the winning region of both players is removed from the graph. This is needed since the winning regions not only contain vertices of the final SCC considered in this iteration, but also vertices of the attractor set over the whole graph. When those vertices are removed, the SCC they were part of might possibly split into multiple new SCCs.

In Zielonka's recursive algorithm optimized with SCC decomposition it is always the case that at the end of the iteration, the whole graph $G\backslash(W_0^G \cup W_1^G)$ is decomposed into SCCs. This might however not always be necessary. There might be SCCs in the old decomposition from which no vertex or edge was removed. These SCCs will not change even if the graph is decomposed into SCCs again. The Zielonka's algorithm does not use this property. The following example illustrates this idea:

Example 1. Take as an example the game depicted in Figure 4.1. In this game every set of 3 vertices with the same priority form an SCC. When taking the final SCC, no other vertices are attracted to it as it is more beneficial for the player of the preceding SCC not to enter this final SCC. So only vertices of the final SCC are removed from the game in every iteration of the loop. Zielonka's recursive algorithm will again decompose the graph into SCCs, resulting in basically the same SCC decomposition, just without that final SCC which was processed that last iteration.

We observe that a lot of computations could be saved when only the graph containing vertices of the SCCs from which some vertices have been removed has to be decomposed into SCCs. Not

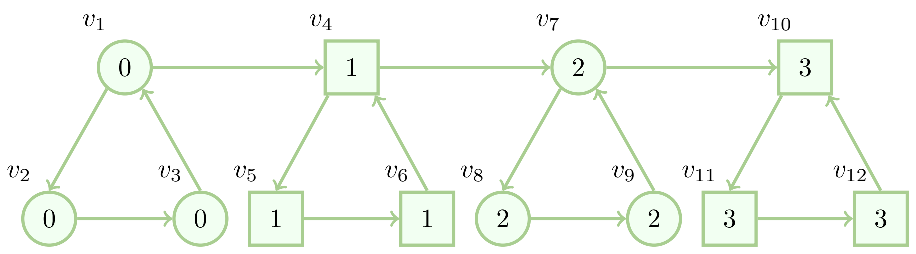


Figure 4.1: A triangle parity game

having to decompose the unchanged SCCs again might save a lot of time as Tarjan's algorithm can run on a much smaller graph. Or Tarjan's algorithm might not have to be called at all after some iterations if all SCCs either have no or all vertices removed. This idea leads to the following 2 research questions:

1. What is the impact on the runtime of Zielonka's recursive algorithm of only recalculating SCCs whose vertices have been removed from the graph instead of recalculating all SCCs?
2. What is the difference in the total number of vertices decomposed into SCCs when recalculating SCCs whose vertices have been removed from the graph compared of recalculating all SCCs?

These questions can be answered theoretically or practically. In this thesis, we focus on answering the questions in the practical sense by implementing both ways and assessing their runtime on the benchmarks described in [10]. The process of how to decompose only the graph containing vertices of SCCs of which some vertices have been removed (partial decomposition) is described in chapter 5.

Another approach would be to, instead of decomposing the graph into SCCs over and over again, to dynamically keep track of the SCC decomposition under edge or vertex deletion. A way of dynamically keeping track of SCCs under edge deletion is described by Łącki in [9]. He proposes to store each SCC in an SCC tree, which can be updated under edge deletion. Chapter 6 will describe how Łącki proposes to store the SCC decomposition in *SCC trees* and the process of updating these SCC trees. Dynamically keeping track of the SCC decomposition might in practice improve the runtime of Parity game solving algorithms. This also can be seen from the former example. If the game of Figure 4.1 consisted of n of these games connected by a single edge to continue the same pattern, then every iteration only 3 vertices are removed while for example after the first iteration $n - 1$ SCCs that did not change have to be recalculated. It might save a lot of time to dynamically remove these vertices. This leads to the next research question:

3. What is the impact on the runtime of Zielonka's algorithm of using dynamic SCC maintenance instead of recalculating the SCCs after each iteration?
4. Does the size of the SCC tree matter for the runtime of Zielonka's algorithm with dynamic SCC maintenance? And if so, what is the relation between size and runtime?
5. How many nodes in the SCC tree are updated during the entire run of Zielonka's recursive algorithm in comparison the number of vertices decomposed by partial decomposition or decomposition of the whole graph.

In chapters 5 and 6 we describe both approaches in more detail, and in chapter 7, we report on the experiments we conducted with implementations of both approaches and answer the research questions listed above.

Chapter 5

Partial re-decomposition

At the end of the Optimized Zielonka's recursive algorithm (Algorithm 2) in line 21, vertices, whose winner is known, are removed from the graph and the remainder of the graph is decomposed into SCCs. However in the old SCC decomposition there might be SCCs from which none of the vertices are removed. When the graph is decomposed again, these SCCs will stay exactly the same. In this chapter, we will develop a novel algorithm to decompose the part of the graph of the SCCs that did have vertices removed.

Example 1. Take as an example the graph on the left in Figure 5.1. When removing v_5 from this graph, it suffices to re-decompose the sub graph of the blue SCC that v_5 is part of. The SCC in the orange square will not change. When decomposing the sub graph of vertices v_1, v_2, v_3 and v_6, 2 new SCCs are found. These new SCCs are shown in the graph on the right in the purple and green square. This graph on the right shows the resulting SCC decomposition after removing v_5

Algorithm 3 Partial re-decomposition

```
1: function PARTIAL(Graph $G = (V, E)$, SCC decomposition $S$, $X \subseteq V$)
2:     $V' \leftarrow \emptyset$
3:     for all vertex $v \in X$ do
4:         $C \leftarrow SCC(v)$
5:         if $C \in S$ then
6:             Remove $C$ from $S$
7:             $V' \leftarrow V' \cup \{C \backslash X\}$
8:         end if
9:     end for
10:    $S' \leftarrow$ SCC_GRAPH_DECOMPOSITION($(V', \{(v, w) \in E | v, w \in V'\})$)
11:    return $S \cup S'$
12: end function
```

In Algorithm 3 the function `Partial(`G, S, X`)` takes a graph G, its SCC decomposition S and a set of vertices X that have to be removed from G. For any vertex $x \in X$ it removes $SCC(x)$ from S. The vertices in $SCC(x)$ that are not in X will have to be assigned to a new SCC, so they are added to the set V'. So after construction of V', the sub graph made from the vertices in V' is decomposed into SCCs. This new collection of SCCs is then merged with the SCCs in S that did not have a vertex in X and returned. So the returned SCC decomposition is the SCC decomposition of the graph $G \backslash X$.

Theorem 3. The function `Partial(`G, S, X`)` returns the SCC decomposition of the graph $G \backslash X$, given a graph $G = (V, E)$, the SCC decomposition S of G and a set of vertices $X \subseteq V$.

Proof. To prove that `Partial(`G, S, X`)` returns the SCC decomposition of $G \backslash X$ assuming that S is a correct SCC decomposition of G we need to prove the following:

1. Each vertex $v \in V \backslash X$ is part of exactly 1 SCC
2. Any vertex $x \in X$ is not a member of any SCC in the returned SCC decomposition
3. For each SCC: any vertex in the SCC can reach every other vertex in the SCC
4. For each SCC: the SCC is maximal

We first show property 1. To prove that each vertex $v \in V \backslash X$ is part of exactly 1 SCC we need to distinguish 2 cases:

- **In S, the SCC v is part of does not contain any vertex in the set X.** In this case $SCC(v)$ will not be removed from S and any SCC not removed from S is present in the returned SCC decomposition. So v is present in at least 1 SCC. Because the SCC does not contain any vertex in X, no vertex of $SCC(v)$ will be added to V'. So since v will not be present in V' it will not be in an SCC in S' and therefore v is present in at most 1 SCC.

- **In S, the SCC v is part of does contain a vertex in the set X.** Because v is part of an SCC that does contain a vertex in the set X, $SCC(v)$ is removed from S and v is added to V'. So S no longer contains an SCC which has v as member. Because $v \in V'$ it will be a member of 1 SCC in S' since the decomposition algorithm returns a correct decomposition. So v is a member of exactly 1 SCC in the returned SCC decomposition.

We next show the second property. To prove that any vertex $x \in X$ is not a member of any SCC in the returned SCC decomposition we assume that x is a member of an SCC in the returned SCC decomposition and prove a contradiction. If x is a member of an SCC in the returned SCC decomposition then either $SCC(x) \in S$ or $SCC(x) \in S'$. If, upon termination, $SCC(x) \in S$, then $SCC(x)$ should not be removed from S in the loop in lines 3 to 11. But this loop removes the SCC of any vertex in X and $x \in X$. So this path reaches a contradiction. If $SCC(x) \in S'$ then $x \in V'$ since S' contains the decomposition of the graph $(V', \{(v,w) \in E | v, w \in V'\})$. However lines 7 to 9 only adds vertices in the set $C \backslash X$ to V', and since $x \in X$, it cannot be the case that $x \in C \backslash X$, so a contradiction is reached on this path as well. So since $SCC(x) \notin S$ or $SCC(x) \notin S'$ we have proven that x is not a member of any SCC in the returned SCC decomposition.

For property 3, to prove that for each SCC C, any vertex in C can reach every other vertex in C, we again use a case distinction:

- **At line 13 $C \in S$.** This case is trivial since the assumption was that S was a correct SCC decomposition of G and since C is still in S at line 13 it means that no vertices or edges have been removed and thus all paths still exist.

- **At line 13 $C \in S'$.** This case is trivial as well since the call to the decomposition algorithm returns a correct SCC decomposition.

So no matter if $C \in S$ or $C \in S'$ we have that any vertex in C can reach every other vertex in C.

Finally for property 4, to prove that for each SCC C: C is maximal, we again use a case distinction:

- **At line 13 $C \in S$.** Assume that C is not maximal. Then there exists a vertex $w \in V \backslash X$ such that all vertices in C can reach w and can be reached from w. But since $w \in V \backslash X$ we also have that $w \in V$, so this would mean that $C \in S$ was already not maximal as input argument. Since S is a correct SCC decomposition of G, and $C \in S$ implies $C \cap X = \emptyset$, C is maximal.

- **At line 13 $C \in S'$.** This case is trivial since the call to the decomposition algorithm returns a correct SCC decomposition and S is a correct SCC decomposition of G at time of input.

So no matter if $C \in S$ or $C \in S'$ we have that C is maximal. □

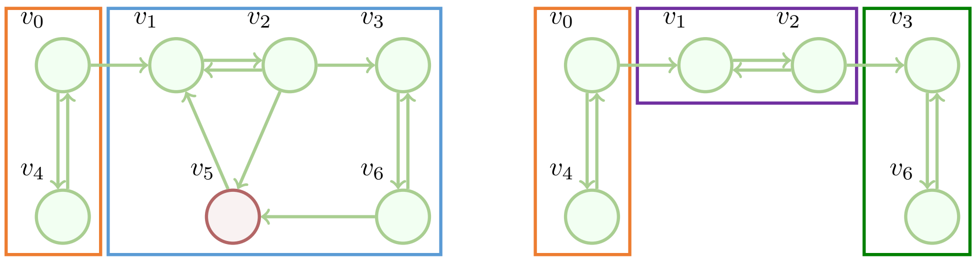


Figure 5.1: Example of a partial re-decomposition.

Theorem 4. The function `Partial`(G, S, X) returns the SCC decomposition of the graph $G\backslash X$, given a graph $G = (V, E)$, the SCC decomposition S of G and a set of vertices $X \subseteq V$, in $O(|V| + |E|)$ time.

Proof. The complexity of $O(|V| + |E|)$ can be seen from the steps of algorithm 3. First the SCCs of the vertices in X are removed. Considering S to be a set of SCCs, removing an SCC will take $O(1)$ time. There are at most $|V|$ vertices in X, so removing the SCCs of the vertices in X takes $O(|V|)$ time. Then the vertices in the removed SCCs that are not in X are added to the set of vertices V'. Since each vertex in V is in exactly 1 SCC, there are at most $|V|$ vertices added to V', taking $O(|V|)$ time. Then the sub graph $(V', \{(v, w) \in E | v, w \in V'\})$ is decomposed in SCCs, which takes $O(|V'| + |\{(v, w) \in E | v, w \in V'\}|) = O(|V| + |E|)$ time. All these steps together take $O(|V| + |V| + |V| + |E|) = O(|V| + |E|)$ time. □

Chapter 6

Dynamic SCC Maintenance

This chapter describes a combination of algorithms and a data structure which make it possible to dynamically keep track of the SCCs of a graph under edge deletion. The input for this chapter is a paper by Łącki [9]. The algorithms from [9], can process a sequence of edge deletions in the data structure in $O(mn)$ time and can check if there is a path between 2 vertices in $O(1)$ time, where m is the number of edges and n the number of vertices.

The data structure which can dynamically keep track of the SCCs of a graph under edge deletion is called an SCC tree. The SCC tree will be described in section 6.1 together with some functions that are needed to describe this structure as well as how to build it.

How this data structure keeps track of the SCCs under edge deletion is described in section 6.2. Next to the function for edge deletion which is described in the paper by Łącki, section 6.2 also describes 2 functions which together can maintain the SCC trees while removing vertices from the graph.

6.1 SCC tree initialization

Each SCC of a given input graph is stored in an *SCC tree*, a data structure in which each leaf node represents a vertex of the SCC and each non leaf node represents a sub graph of the SCC such that this sub graph on its own is also an SCC of the subgraph and contains all vertices of the leaf nodes in the sub tree rooted at this non leaf node.

Definition 10 (SCC tree)**.** Formally an SCC tree is a tuple $(TN, (I, L, S), E, r)$, where TN is a set of nodes, (I, L, S) is a partition of TN into inner nodes (I), leaf nodes (L) and split nodes (S), $E \subseteq I \times TN$ is a set of edges and $r \in TN$ the root of the tree, where we require that $\neg\exists_{w \in TN} : (w, r) \in E$. E

To to build an SCC tree the functions `split(`G, d`)` and `condense(`G`)` are needed. These 2 functions are described in subsections 6.1.1 and 6.1.2. Then subsection 6.1.3 will follow with a detailed explanation of the *SCC tree* data structure and how to build an *SCC tree*. It will also give the complexity for building this *SCC tree* and the space complexity to store the *SCC tree* in memory.

6.1.1 Split

The function `split(`G, d`)` takes a vertex $d \in V$ in a graph $G = (V, E)$ and splits it into 2 fresh vertices, a vertex that inherits the incoming edges $(v, d) \in E$ (*inedges*) of d and a vertex that gets the outgoing edges $(d, v) \in E$ (*outedges*) of d.

The splitting of d into 2 vertices follows the following steps. First it adds 2 new vertices to V which are called d_{in} and d_{out}. The vertex d itself is removed from V. Then all *inedges* of d are removed and each edge (v, d) is replaced by the corresponding edge (v, d_{in}). All *outedges* of d are

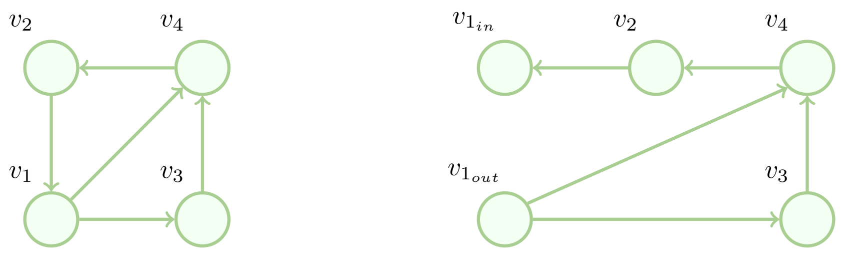


Figure 6.1: The graph on the right shows visualisation of the effect of the function `split(G, v`$_1$`)`, on the graph on the left

also removed and this time each edge (d, v) is replaced by the corresponding (d_{out}, v) edge. The function `split`(G, d) returns the modified version of the graph.

Definition 11 (Split)**.** Formally the graph returned by `split(G, d)` is the graph $G' = (V', E')$ with $V' = (V\backslash\{d\}) \cup \{d_{in}, d_{out}\}$ and $E' = (E\backslash\{(v, w)|v = d \lor w = d\}) \cup \{(v, d_{in})|(v, d) \in E\backslash\{(d, d)\}\} \cup \{(d_{out}, w)|(d, w) \in E\backslash\{(d, d)\}\} \cup \{(d_{out}, d_{in})|(d, d) \in E\}$.

Figure 6.1 visualizes the effect of the `split`(G, d) function. In the graph on the left the vertex v_1 is split into $v_{1_{in}}$ and $v_{1_{out}}$. Vertex v_1 itself is removed. The *inedge* (v_2, v_1) of v_1 is replaced by $(v_2, v_{1_{in}})$, and the *outedges* (v_1, v_3) and (v_1, v_4) are replaced by $(v_{1_{out}}, v_3)$ and $(1_{out}, v_4)$ respectively.

6.1.2 Condense

The graph $G' = (V', E')$, with $V' \subseteq 2^V$ and $E' \subseteq V' \times V' \times V \times V$, returned by the function `Condense`(G) described in [9] is a condensation of $G = (V, E)$. In a condensation the SCCs of the graph are replaced by a single vertex which gets all edges going in and out of the SCC. The edges in a condensed graph are tuples which contain both the condensed vertices and the original endpoints of the edge. In this chapter the graph G' returned by `Condense`(G) will be called a condensed graph and its vertices condensed vertices.

In the set of condensed vertices V' returned by `Condense`(G) each condensed vertex $v' \in V'$ represents a unique SCC in G. Every edge $(a, b, y, v) \in E'$, $a \neq b$, is an edge between condensed vertices $a, b \in V'$ annotated by a pair (y, v), representing an edge in G. In these edges a is the condensed vertex representing the SCC containing y and b is the condensed vertex representing the SCC containing v. There can be multiple edges between the same condensed vertices in V' each labeled with a different edge from G. When removing an edge (a, b, y, v) from E', this will sometimes be shortened to removing (y, v) from E', which is a unique label.

Definition 12 (condense)**.** Let $SCC(x)$ denote the SCC $x \in V$ is part of. Then the condensed graph $G' = (V', E')$, with $V' \subseteq 2^V$ and $E' \subseteq V' \times V' \times V \times V$, returned by the function `Condense`(G) can be described by $V' = \{SCC(v)|v \in V\}$ and $E' = \{(a, b, c, d)|a, b \in V' \land a \neq b \land c \in a \land d \in b \land (c, d) \in E\}$.

Example 1. Figure 6.2 shows the effect of `Condense`(G) on a graph. The graph on the left of Figure 6.2 has 3 SCCs: $\{v_2\}$, $\{v_1, v_3, v_4\}$ and $\{v_5, v_6\}$. So for these SCCs the condensed vertices $\{v_2\}$, $\{v_1, v_3, v_4\}$ and $\{v_5, v_6\}$ are added to V'. The edges which have endpoints in different SCCs are $(v_2, v_1), (v_3, v_5), (v_4, v_6)$, so for each of these edges an edge between the condensed vertices is added, resulting in $E' = \{(\{v_2\}, \{v_1, v_3, v_4\}, v_2, v_1), (\{v_1, v_3, v_4\}, \{v_5, v_6\}, v_3, v_5), (\{v_1, v_3, v_4\}, \{v_5, v_6\}, v_4, v_6)\}$. There are 2 edges between condensed vertices $\{v_1, v_3, v_4\}$ and $\{v_5, v_6\}$ in E' because there were 2 edges between vertices in the SCC $\{v_1, v_3, v_4\}$ and $\{v_5, v_6\}$, but both are annotated with different edges from G. The condensed graph is shown in the graph on the right in Figure 6.2. In a condensed graph, an edge $(a, b, c, d) \in E'$ is shown as an arrow from a to b with a label (c, d).

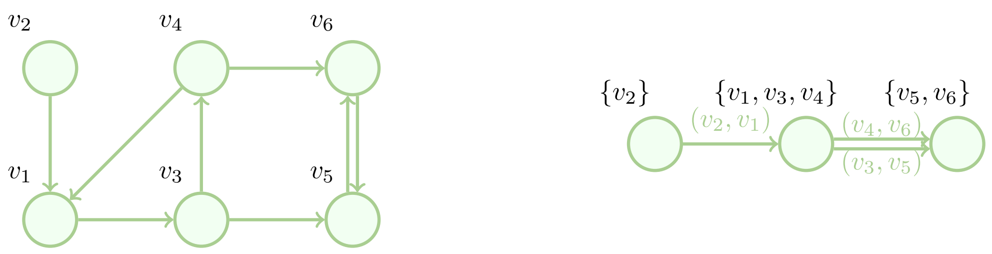


Figure 6.2: The graph on the right shows visualisation of the effect of the function `condense(G)`, on the graph on the left

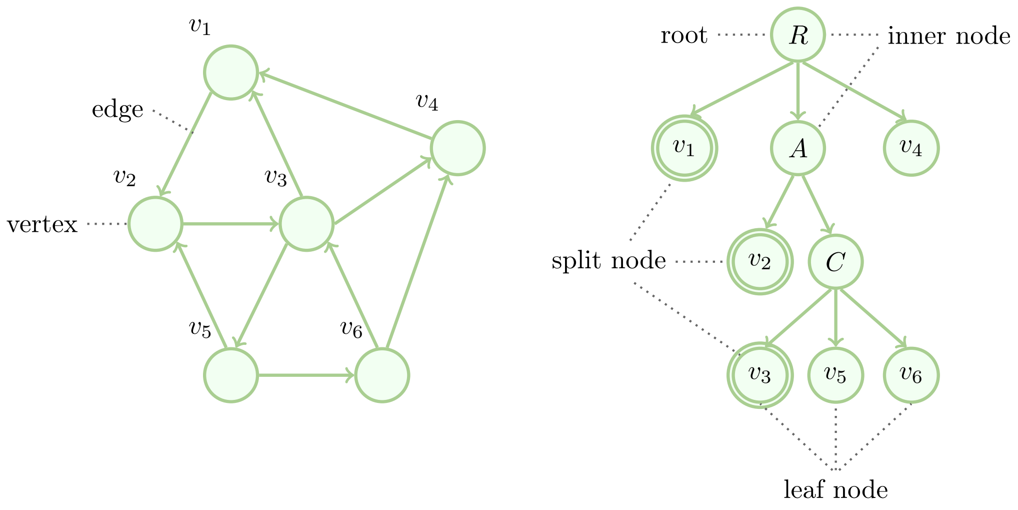


Figure 6.3: legend

Next to `condense`, [9] also describes the function `splitAndCondense(`G, d`)`, which applies the condense function after the split function, i.e. `splitAndCondense(`G, d`)` $=$ `condense(split(`G, d`))`. An application of this function results in a condensed directed acyclic graph as if there had been cycles these would be in the same SCC and thus condensed.

Example 2. As an example, condensed graph G_{R_b} in Figure 6.4 on page 25 shows the result of applying `splitAndCondense` on the graph depicted in the top left of Figure 6.4.

6.1.3 SCC tree

Using the `splitAndCondense` function an SCC tree, which is the data structure used in [9], can be built. An SCC tree represents exactly one SCC. If the input graph consists of multiple SCCs, each of these SCCs will get their own SCC tree. To explain the structure of an SCC tree the graph G in the left top corner of Figure 6.4 will be used, which will result in the SCC tree in the right top corner. Algorithm 4 shows the pseudocode outlining how to build an SCC tree.

For an overview of a graphical representation of the terms regarding SCC trees see Figure 6.3. We refer to the input graph as the original graph. The vertices of the original graph are called vertices, the vertices of the SCC tree are called nodes.

Recall that an SCC tree is a tuple $(TN, (I, L, S), E, r)$, Where each node $N \in TN$ represents an SCC. We associate a unique name N_C for a node representing SCC $C \subseteq V$; as convention we write N_v when $C = \{v\}$. The SCCs represented by the nodes in TN do either not overlap, or one is a subset of the other. So $\forall_{N_{C_1} \neq N_{C_2} \in TN} : (C_1 \cap C_2 = \emptyset) \vee (C_1 \subset C_2) \vee (C_2 \subset C1)$. It also must be the case that for each inner node $N_C \in I$ there is a path from N_C to $N_v \in L$ for each vertex $v \in C$ and this leaf node must exist. So $\forall_{N_C \in I} \forall_{v \in C} : N_v \in L$ and $\forall_{N_C \in I} \forall_{N_v \in L} : v \in C \implies (N_C, N_v) \in E^*$ where E^* is the transitive closure of E.

Definition 13 ($G(N_C)$)**.** For a node N_C, $G(N_C)$ denotes the subgraph of G restricted to C, i.e. $G(N_C) = G \cap C$.

For each inner node $N_C \in I$, there are at least 2 edges in E. Of all *outedges* of N_C exactly 1 edge is to a split node $N_d \in S$. The targets of all other *outedges* of N_C have to be inner or leaf nodes, so are part of the set $TN \backslash S$. For each edge $(N_1, N_2) \in E$ the node N_2 is called the child of node N_1 and N_1 is the parent of node N_2.

Definition 14 ($p(N)$)**.** Given an SCC tree and node $N \in TN$ with $N \neq r$, $p(N)$ stands for the parent node of N. So $p(N)$ is the node $N_P \in TN$ such that $(N_P, N) \in E$ and there is only 1 such node. For root node r, $p(r)$ is not defined.

Definition 15 ($D(N_C)$)**.** Given an SCC tree and an inner node $N_C \in TN$, $D(N_C)$ is the condensed graph generated by `SplitAndCondense(`$G(N_C), d$`)`, with $d \in C$, $N_d \in S$ and $(N_C, N_d) \in E$.

The condensed graph $D(N_C) = (V', E')$ consists of several condensed vertices each of which represent an SCC. The set TN contains a vertex $N_C V$ for each condensed vertex $CV \in V' \backslash \{\{d_{in}\}, \{d_{out}\}$. Additionally there is an edge $(N_C, N_C V) \in E$. In future algorithms $V(D(N_C)) = V'$ is the set of condensed vertices in $D(N_C)$ and $E(D(N_C)) = E'$ is the set of edges in $D(N_C)$.

The building of the SCC tree $(TN, (I, L, S), E, r)$ for an SCC C with more than 1 vertex as described in algorithm 4 approximately follows the following process:

1. A node N_C represents the whole SCC C. This is the root node r, and since C has more than 1 vertex, N_C is included in both TN and I.
2. A function call is made to `SplitAndCondense(`$G(N_C), d$`)` for some randomly selected vertex $d \in C$.
3. For the condensed vertices $\{d_{in}\}$ and $\{d_{out}\}$ a single node N_d is added to TN and S. Since N_d is a child of N_C, an edge (N_C, N_d) is added to E.
4. For each condensed vertex CV:
    a) If the SCC represented by that vertex has only 1 vertex, then the node N_{CV} is a leaf node and added to TN and L. Since N_{CV} is a child of N_C, an edge (N_C, N_{CV}) is added to E.
    b) If the condensed vertex CV represents an SCC with multiple vertices, then the node N_{CV} is an inner node and added to TN and I. Since N_{CV} is a child of N_C, an edge (N_C, N_{CV}) is added to E. Since N_{CV} is an inner node, it will have children of its own, so the process returns to step 2 where now N_C is set to N_{CV}.

The loop in steps 2 to 4 continues until the whole tree is built. The tree is built in such way because, as stated by [9], by tracing the connectivity of `splitAndCondense(`$G(N_C), d$`)` one can retrieve information about the connectivity of $G(N_C)$. This means that as long as every condensed vertex in `splitAndCondense(`$G(N_C), d$`)` can reach $\{d_{out}\}$ and can be reached from $\{d_{in}\}$, it also is the case that any vertex in $G(N_C)$ can reach any other vertex in (N_C) making $G(N_C)$ a strongly connected component. Once a condensed vertex in `splitAndCondense(`$G(N_C), d$`)` cannot reach $\{d_{out}\}$ or cannot be reached from $\{d_{in}\}$, it means that all vertices in the SCC represented by that condensed vertex cannot reach d or cannot be reached from d, which means that $G(N_C)$ is no longer an SCC.

Example 3. An example of how an SCC tree is formed using the `splitAndCondense` function is shown in Figure 6.4. To save space in the figure, the inner nodes N_C have been shortened to a single letter. The root $R = N_{\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}}$ is added to the set of nodes and inner nodes. Since the SCC contains multiple vertices, `splitAndCondense(`$G(R), v_1$`)` is called. It has 2 split

Algorithm 4 Building an SCC tree

```
function DYNAMIC_SCC_DECOMPOSITION(Graph $G$)
    $DSD \leftarrow \{:\}$ empty Map, SCC tree root node to SCC tree
    $EBT \leftarrow \emptyset$ empty Set of edges between SCC trees
    $VTR \leftarrow \{:\}$ empty Map of vertex to SCC tree root node
    $D \leftarrow$ SCC_GRAPH_DECOMPOSITION($G$)
    for all SCC $C \in D$ do
        if $|C| = 1$ then
            pick $v \in C$
            $VTR[v] \leftarrow N_C$
            $DSD[N_C] \leftarrow \{(\{N_C\}, (\emptyset, \{N_C\}, \emptyset), \emptyset, N_C)\}$
        else
            $(TN, (I, L, S), E, r) \leftarrow$ BUILDTREE($G, C$)
            for all vertex $v \in C$ do
                $VTR[v] \leftarrow r$
            end for
            $DSD[r] \leftarrow (TN, (I, L, S), E, r)$
        end if
        $EBT \leftarrow EBT \cup \{(v, w) \in E | v \in C \wedge w \notin C\}$
    end for
    return $(DSD, EBT, VTR)$
end function

function BUILDTREE(Graph $G$, SCC $C$)
    $TN \leftarrow \{N_C\}$, $I \leftarrow \{N_C\}$, $L \leftarrow \emptyset$, $S \leftarrow \emptyset$, $E \leftarrow \emptyset$, $r \leftarrow N_C$
    $queue \leftarrow [N_C]$
    while $queue$ is not empty do
        $N \leftarrow queue.dequeue$
        $(V, E) \leftarrow G(N)$
        pick $d \in V$
        $(V', E') \leftarrow$ SPLITANDCONDENSE($G(N), d$)
        for all condensed vertices $CV \in V' \backslash \{\{d_{in}\}, \{d_{out}\}\}$ do
            $TN \leftarrow TN \cup \{N_{CV}\}$
            $E \leftarrow E \cup \{(N, N_{CV})\}$
            if $|CV| = 1$ then
                $L \leftarrow L \cup \{N_{CV}\}$
            else
                $I \leftarrow I \cup \{N_{CV}\}$
                $queue.enqueue(N_{CV})$
            end if
        end for
        $TN \leftarrow TN \cup \{N_d\}$
        $E \leftarrow E \cup \{(N, N_d)\}$
        $S \leftarrow S \cup \{N_d\}$
    end while
    return $(TN, (I, L, S), E, r)$
end function
```

nodes and 2 condensed vertices representing SCCs containing more than 1 vertex. Thus in the first layer of the SCC tree, the root R has 3 children: split node $N_{\{v_1\}}$, and 2 children that are both inner nodes: $A = N_{\{v_2,v_3,v_5,v_7\}}$ and $B = N_{\{v_4,v_6,v_8\}}$. The split node $N_{\{v_1\}}$ is added to S and inner nodes A and B are added to I. For each child an edge is introduced from R to the child. The steps are repeated for both inner nodes: `splitAndCondense(`$G(A), v_2$`)` is called for node A. The condensed graph $D(A)$ has 2 split nodes and a condensed vertex representing more than 1 vertex. So A has 2 children: split node $N_{\{v_2\}}$ and inner node $C = N_{\{v_3,v_5,v_7\}}$. The split node $N_{\{v_2\}}$ is added to S, inner nodes c is added to I and an edge is introduced from A to both children. Since C is an inner node, the steps are repeated: `splitAndCondense(`$G(C), v_3$`)` is called and now only contains 2 split nodes and 2 condensed vertices representing SCCs with only a single vertex. Now C has 3 children: split node $N_{\{v_3\}}$ and leaf nodes $N_{\{v_5\}}$ and $N_{\{v_7\}}$. The split node $N_{\{v_3\}}$ is added to S and leaf nodes $N_{\{v_5\}}$ and $N_{\{v_7\}}$ are added to L. An edge is introduced from C to each child. Since C has no inner node children the process stops here. The process for inner node B is similar to the process of inner node C.

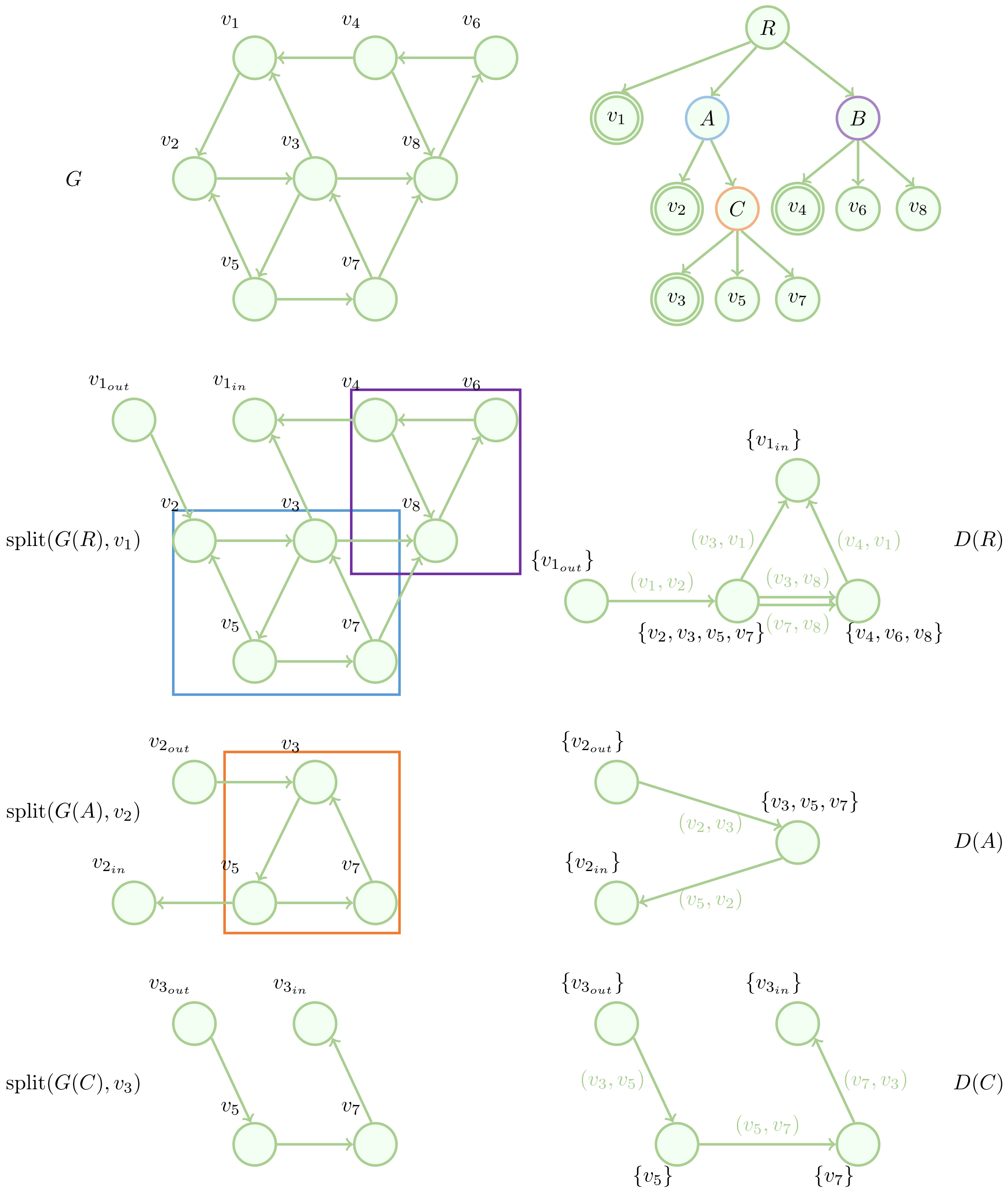
The dynamic SCC decomposition of graph $G = (V, E)$, which contains an SCC tree for each SCC of G also stores some auxiliary variables needed for ease of access by the maintenance algorithms. For each vertex $v \in V$ it stores the root of the SCC tree which contains leaf node N_v. Each edge $(v, w) \in E$ is stored in $D(N)$ of the least common ancestor N of N_v and N_w. The dynamic SCC decomposition stores this least common ancestor for each edge, so it can be found in constant time when this edge might need to be deleted. Lastly the dynamic SCC decomposition stores the edges between SCC trees, so the edges in E whose endpoints are part of different SCCs of G.

Theorem 5. According to [9], an SCC tree can be constructed for an SCC C of a graph G in $O(m\delta)$ time, where δ is the height of the tree and m the number of edges in $G \cap C$, and requires $O(n + m)$ space, with n the number of vertices in $G \cap C$.

Below, a short proof that these complexities can be met will be given. In appendix D.1 the implementation of the process of building SCC trees for a given input graph is given with a more detailed proof that constructing an SCC tree in such way can be done in $O(m\delta)$ time in practice. Appendix F shows how the implementation stores the SCC tree with a more detailed proof that this implementation meets the $O(n+m)$ space complexity requirement. It also makes some details which were omitted in [9] more explicit.

Proof. We first address the space complexity. That an SCC tree can store an SCC C in $O(n + m)$ space can be reasoned as follows: each inner node has at least 2 children, 1 of which is a leaf (split) node. Therefore there can be at most $O(|C|)$ inner nodes and at most $O(2 * |C|) = O(|C|)$ nodes in an SCC tree in total. Each node, except the root node, stores its parent, so the parent is stored at most $O(|C|)$ times. A node is a child of at most 1 other node. By storing the children of a node in a list in the node, storing the children will take $O(|C|)$ space over all nodes in total. Each edge in $G \cap C = (C, E)$ is stored in 1 inner node. By storing the edges in lists the total storage space needed to store the edges is $O(|E|)$. Lastly the auxiliary variables. The root has to be stored for each vertex. Storing this information in a hashmap takes at most $O(|C|)$ space, considering a constant load factor. The same goes for storing in which inner node each edge is stored. Storing this information for each edge in the SCC in a hashmap takes at most $O(|E|)$ space, considering a constant load factor. So in total to store the SCC tree takes $O(|C| + |E|) = O(m + n)$ space.

Next, we consider the runtime complexity. To construct an SCC tree for SCC C, with $G \cap C = (V, E)$, in $O(m\delta)$ time, each layer cannot take more than $O(m)$ time. For each inner node N_C, with $G(N_C) = (V', E')$, `splitAndCondense` needs to find the SCCs after splitting. This would take $O(|C| + |E'|)$ time with an algorithm like Tarjan's algorithm per inner node. For all inner nodes in a layer of the tree together this would take at most $O(|V| + |E|)$ time as the sub graphs of these SCCs do not overlap. Aside from these for every inner, leaf or split node a constant number of items, such as corresponding edges to children, need to be added to a set. Adding items to a set takes constant time, and since there are at most $O(|C|)$ inner nodes, this will


G
v_1 v_4 v_6 v_2 v_3 v_8 v_5 v_7
R v_1 A B v_2 C v_4 v_6 v_8 v_3 v_5 v_7
split$(G(R), v_1)$
$v_{1_{out}}$ $v_{1_{in}}$ v_4 v_6 v_2 v_3 v_8 v_5 v_7
$\{v_{1_{in}}\}$ $\{v_{1_{out}}\}$ (v_3, v_1) (v_4, v_1) (v_1, v_2) (v_3, v_8) (v_7, v_8) $\{v_2, v_3, v_5, v_7\}$ $\{v_4, v_6, v_8\}$
$D(R)$
split$(G(A), v_2)$
$v_{2_{out}}$ v_3 $v_{2_{in}}$ v_5 v_7
$\{v_{2_{out}}\}$ $\{v_3, v_5, v_7\}$ (v_2, v_3) $\{v_{2_{in}}\}$ (v_5, v_2)
$D(A)$
split$(G(C), v_3)$
$v_{3_{out}}$ $v_{3_{in}}$ v_5 v_7
$\{v_{3_{out}}\}$ $\{v_{3_{in}}\}$ (v_3, v_5) (v_7, v_3) (v_5, v_7) $\{v_5\}$ $\{v_7\}$
$D(C)$
Figure 6.4: Example of storing the graph of an SCC as an SCC tree. The graph G on the left top corner is stored in the SCC tree on the right top corner.

take $O(|C|)$ for all inner nodes together. So all together to create a layer of the SCC tree takes $O(|C| + |E|) = O(|E|) = O(m)$ time. □

6.2 SCC tree maintenance

To delete an edge from the *SCC tree* the functions called `merge(`G, d_1, d_2, d`)` and `findUnreachable(`G, S, w`)` are needed to determine the impact of the edge deletion on the SCC. Subsection 6.2.1 will first describe the functionality of the `merge(`G, d_1, d_2, d`)` function. Subsection 6.2.2 will then describe how the `findUnreachable(`G, S, w`)` function works and its complexity. Then subsection 6.2.3 will show how to delete an edge from the *SCC tree* and the complexity for this action.

The information about the functions `split(`G, d`)`, `condense(`G`)`, `merge(`G, d_1, d_2, d`)`, `findUnreachable(`G, S, w`)` and `deleteEdge(`T, e`)` is taken from [9]. Next to these functions subsection 6.2.4 describes how to remove vertices from an *SCC tree* which we created based on how `deleteEdge(`T, e`)` updates an *SCC tree* after edge deletion. Deleting vertices will be necessary for Zielonka's algorithm as after every iteration the vertices in the attractor sets have to be removed from the decomposition and to remove them from the decomposition they have to be removed from the SCC tree they are in first. Lastly section 6.2.5 shows which variables have to be updated when a full *SCC tree* is deleted from a dynamic SCC decomposition. These 2 new functions together allow the action of removing vertices from a graph.

6.2.1 Merge

The `merge(`G, d_1, d_2, d`)` function is basically a function which reverses the effect of `split(`G, d`)`. It removes the vertices d_1 and d_2 from the vertices of G and replaces these with vertex d. The edges of G remain the same except for those with source d_1, those will emerge from d, and hiwch with target d_2, which will be directed to d.

When `merge(`G`)` is used instead of `merge(`G, d_1, d_2, d`)`, on a graph which was split by `split(`G, d`)` before, then d_{in} will be used for d_1, d_{out} will be used for d_2 and the d from `split` is used as input for the d of `merge`. The input graph G for `merge(`G, d_1, d_2, d`)` or `merge(`G`)` can either be a normal graph, in which case d_1, d_2 and d are vertices, or a condensed graph where d_1, d_2 and d are condensed vertices.

Definition 16 (Merge)**.** Formally the graph $G' = (V', E')$ returned by `merge(`G, d_1, d_2, d`)` can be described as $V' = (V \backslash \{d_1, d_2\} \ \cup \ \{d\})$ and $E' = (E \backslash \{(v, w) | v = d_1 \ \vee \ w = d_2\}) \ \cup \{(v, d) | (v, d_2) \in E\} \cup \{(d, v) | (d_1, v) \in E\}$. In case G was a condensed graph, the condensed graph $G' = (V', E')$ returned by `merge(`G, d_1, d_2, d`)` can be described as $V' = (V \backslash \{d_1, d_2) \ \cup \ \{d\}\}$ and $E' = (E \backslash \{(v, w, a, b) | v = d_1 \vee w = d_2\}) \cup \{(v, d, a, b) | (v, d_2, a, b) \in E\} \cup \{(d, v, a, b) | (d_1, v, a, b) \in E\}$.

Example 1. As an illustration using Figure 6.1, a call to `merge(G)` on the graph on the right would take $v_{1_{in}}$ for d_1, $v_{1_{out}}$ for d_2 and v_1 for d as these are the vertices used and produced by calling the function `split(G,` v_1`)` on the graph on the left. So a call to `merge(G)` on the graph on the right would result in the graph on the left.

6.2.2 Find unreachable

When in an SCC an edge of the SCC is deleted, it might have an effect on the inner node which stored this edge and possibly on this inner node's ancestors as well. The reason is that an inner node represents an SCC and deleting an edge might cause this SCC to no longer be strongly connected. In this case an update of the SCC tree would be required to make sure the SCC tree is again really an SCC tree of the graph.

The function `findUnreachable(`G, S`)` receives as input a condensed graph $G = D(N_C)$ of the SCC C represented by the inner node N_C from which an edge was deleted. It also receives a list S which contains the condensed vertices in G that might have become a sink or a source as a consequence of the edge deletion in an inner node. A source is a (condensed) vertex with no *inedges* and a sink a (condensed) vertex with no *outedges*.

The function `findUnreachable(`G, S`)` will find all condensed vertices and their in and *outedges* (incident edges) in G that cannot reach the split node of G or cannot be reached from the split node. As long as all condensed vertices in G can reach $\{d_{in}\}$ and can be reached from $\{d_{out}\}$, with $d \in C$, $N_d \in S$ and $(N_C, N_d) \in E$, then the SCC represented by the inner node is still an SCC. The output of `findUnreachable(`G, S`)` is a minimal set of condensed vertices and their incident edges such that if you remove these condensed vertices, G will again be an SCC. According to [9] `findUnreachable(`G, S`)` will find this set of condensed vertices and edges in $O(|S| + |V| + |E|)$ time, with $G = (V, E)$ the input condensed graph.

The `findUnreachable(`G, S`)` function, of which the pseudocode is described in algorithm 5, taken from [9], makes use of 2 helper functions:

- `findUnreachableDown(`G, S, w`)` returns all condensed vertices and their incident edges in G which cannot be reached from source $w = \{d_{in}\}$. The input graph G is the condensed graph $D(N_C)$ of the SCC C represented by inner node N_C. The list S contains all condensed vertices in G that could possibly be a source. Any endpoint of any edge which was deleted from G has the possibility to become a source and will therefore be present in S. The condensed vertex w is the source split node of G.

- `findUnreachableUp(`G, S, w`)` returns all condensed vertices and their incident edges in G which cannot reach sink $w = \{d_{out}\}$. The input graph G is the condensed graph $D(N_C)$ of the SCC C represented by inner node N_C. The list S contains all condensed vertices in G that could possibly be a sink. Any endpoint of any edge which was deleted from G has the possibility to become a sink and will therefore be present in S. The condensed vertex w is the source split node of G.

To find all condensed vertices which cannot be reached from w, the `findUnreachableDown(`G, S, w`)` function first checks all condensed vertices in S. If the vertex is w, or if it still has *inedges* then it is still possible that the condensed vertex might be reachable from w. If neither is the case then the condensed vertex really is a source and thus cannot be reached from w and is added to the queue. The function uses this queue to process the vertices that cannot be reached from w in order, it contains those found by the function and that are not yet processed. Each vertex in the queue is added to the set of unreachable vertices, and its incident edges are added to the set of edges. Then it removes all its outgoing edges from G and checks if the target condensed vertex of any of these edges now has become a source as well. If so it is not reachable from w. If the target condensed vertex has become a source then it is added to the queue as well. This continues until the queue contains no more sources. All vertices that are not in the set of unreachable vertices can be reached from w.

Finding all vertices which cannot reach w with the `findUnreachableUp(`G, S, w`)` function is similar to the process of `findUnreachableDown(`G, S, w`)`. Instead of looking for sources, it now checks for sinks and instead of removing outgoing edges, the incoming edges are removed. When the source condensed vertex of such edge has become a sink it is added to the queue, since a sink cannot reach any other vertex and thus cannot reach w.

Example 2. As an example, consider running `findUnreachableDown(`$G, \{\{v_{1_{in}}\}, \{v_3\}, \{v_4\}\}$`)` on condensed graph G in Figure 6.5. This could be a condensed graph of an SCC of which edges $(\{v_{1_{in}}\}, \{v_3\}, v_1, v_3)$ and $(\{v_3\}, \{v_4\}, v_3, v_4)$ have been deleted. The list $S = \{\{v_{1_{in}}\}, \{v_3\}, \{v_4\}\}$ with the affected condensed vertices contains an actual source $\{v_3\}$, the split node and a condensed vertex which is not a source. The algorithm would first sift through these. Condensed vertex $\{v_{1_{in}}\}$ is not added to the queue since it is a split node, $\{v_3\}$ has no *inedges* so it is added to the queue, while $\{v_4\}$ is not because it has an *inedge*. It takes a condensed vertex from the queue, $\{v_3\}$,

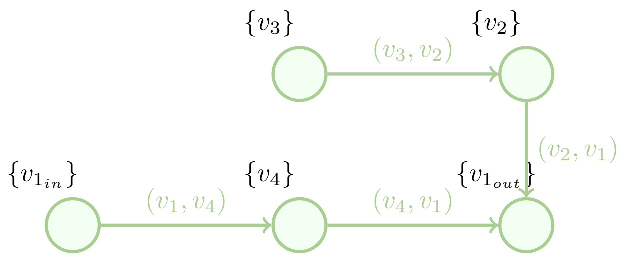


Figure 6.5: Running `findUnreachableDown(`$G, \{\{v_{1_{in}}\}, \{v_3\}, \{v_4\}\}$`)` on this condensed graph G will yield unreachable vertices v_3 and v_2 with their incident edges $(\{v_3\}, \{v_2\}, v_3, v_2)$ and $(\{v_2\}, \{v_{1_{out}}\}, v_2, v_1)$.

adds it to the list of unreachable vertices, adds its outgoing edge $(\{v_3\}, \{v_2\}, v_3, v_2)$ to the list of incident edges and removes the outgoing edge from G. It checks the target of this outgoing edge and finds that now $\{v_2\}$ no longer has any *inedges*, thus $\{v_2\}$ is added to the queue. The next condensed vertex that is taken from the queue is $\{v_2\}$, again it is added to the list of unreachable vertices, and its outgoing edge $(\{v_2\}, \{v_{1_{out}}\}, v_2, v_1)$ to the list of incident edge. Its outgoing edge is removed from G. The target condensed vertex of this edge still has *inedges* so the algorithm is done. The unreachable condensed vertices found are $\{v_3\}$ and $\{v_2\}$ which can indeed not be reached from $\{v_{1_{in}}\}$.

6.2.3 Deleting an edge from an SCC tree

After deleting an edge from an inner node of an SCC tree it might be the case that the SCC represented by the inner node no longer is an SCC tree. The function `deleteEdge(`T, e`)` is responsible for deleting an edge $e \in E$ from SCC tree T and updating the SCC tree such that after the function is done every inner node of T again represents an SCC.

Example 3. As an example of the 3 terms, take Figure 6.6. In the graph in this figure $p(Z) = T$. In the left bottom rectangle $G(Z)$ of node Z is shown and the condensed graph $D(T)$ of node T is shown in the middle black square.

An edge that is deleted from the graph is either part of an SCC, in which case it is stored in one of the inner nodes of the corresponding SCC tree, or an edge between SCCs. When the edge is an edge between SCCs it can just be deleted without affecting any SCC trees. This is not the case for an edge that is part of an SCC. Algorithm 6 describes how to delete an edge from an SCC tree. According to [9], this algorithm can delete an edge and update the SCC tree in $O(mn)$ time.

Deleting an edge from an SCC tree can be divided in 2 cases: deleting an edge from the root node of the SCC tree or deleting an edge from a non root inner node of the SCC tree. In case of deleting an edge from a non root inner node, possible children that have to be migrated upwards in the tree are added to the parent of the node. In case of deleting an edge from a root inner node, those children are not added to the parent but have to be transformed into new separate SCC trees.

After deleting an edge from a root node B, `findUnreachable` is called on $D(B)$ to check which children and edges have to be removed from B to make B a valid SCC tree. If none then nothing has to be done. If any, the nodes of the condensed vertices returned by `findUnreachable` have to be removed to make sure $G(B)$ again is an SCC. These nodes will be made into independent SCC trees since they still represent an SCC. The incident edges of vertices in the SCC represented by this node have been returned by `findUnreachable`, these are removed from B as well and now have become vertices between SCCs. If B now only has the split node left as its child, then B is replaced by a single leaf node representing the vertex that was stored in the split node. Either way B again is an SCC and so are the children that have been turned into independent SCC trees based on how the SCC tree is built.

Algorithm 5 Helper function of UpdateSCC-Tree

```
function FindUnreachable($(V, E), s$)
    $(U_{down}, I_{down}) \leftarrow$ FindUnreachableDown($(V, E), S, d_{out}$)
    $(U_{up}, I_{up}) \leftarrow$ findunreachableUp($(V, E), S, d_{in}$)
    return $(U_{down} \cup U_{up}, I_{down} \cup I_{up})$
end function

function FindUnreachableDown($(V, E), S, w$)
    $U \leftarrow \{:\}$ empty set
    $I \leftarrow \{:\}$ empty set
    $Q \leftarrow []$ empty queue
    for all $x \in S \backslash \{w\}$ do
        if $x$ has no in edges then
            Enqueue($Q, x$)
        end if
    end for
    while $Q \neq$ empty queue do
        $v \leftarrow$ Dequeue($Q$)
        $U \leftarrow U \cup \{v\}$
        $I \leftarrow I \cup$IncidentEdges($v$)
        for all $(v, x) \in E$ do
            $E \leftarrow E \backslash \{(v, x)\}$
            if $x$ has no in edges then
                Enqueue($Q, x$)
            end if
        end for
    end while
    return $(U, I)$
end function

function FindUnreachableUP($(V, E), S, w$)
    $U \leftarrow \{:\}$ empty set
    $I \leftarrow \{:\}$ empty set
    $Q \leftarrow []$ empty queue
    for all $x \in S \backslash \{w\}$ do
        if $x$ has no out edges then
            Enqueue($Q, x$)
        end if
    end for
    while $Q \neq$ empty queue do
        $v \leftarrow$ Dequeue($Q$)
        $U \leftarrow U \cup \{v\}$
        $I \leftarrow I \cup$IncidentEdges($v$)
        for all $(x, v) \in E$ do
            $E \leftarrow E \backslash \{(x, v)\}$
            if $x$ has no out edges then
                Enqueue($Q, x$)
            end if
        end for
    end while
    return $(U, I)$
end function
```

Deleting an edge from a non root inner node of the tree is similar but takes a few more steps, but still after deleting the edge from an inner node I, `findUnreachable` is called on $D(I)$ to check which children have to be removed such that the sub-tree rooted at I is again a valid SCC tree. If none, nothing has to be done, since the whole tree is fine. If any, the nodes of the condensed vertices returned by `findUnreachable` have to be moved elsewhere, outside of I, to make sure I again is an SCC. So those nodes are added to the parent of I: $p(I)$. The incident edges of vertices in the SCC represented by any of the moved nodes have to be moved to $p(I)$ as well. If I now only has the split node left as its child, then I is replaced by a single leaf node representing the vertex that was stored in the split node. So now I is fine and again an SCC, but this might not be the case for $p(I)$. So now `findUnreachable` is called on $D(P(I))$ to find if any nodes have to be removed from $p(I)$ to ensure $p(I)$ stays an SCC, and in this case move those nodes and incident edges up to $p(p(I))$ and then do the same, until either the ancestor is found to still be an SCC or the root node is reached.

Example 4. As an example of deleting an edge between vertices in an SCC, edge (v_6, v_7) is deleted from the graph G_{old} in the top left corner of Figure 6.7. Edge (v_6, v_7) is stored in node Z as this is the lowest common ancestor between leaf nodes v_6 and v_7. In the red rectangle in Figure 6.7 $D(Z)$ is shown with the edge to be deleted in red. After (v_6, v_7) has been deleted, `findUnreachable` will find that condensed vertex $\{v_7\}$, with incident edge $(\{v_7\}, \{v_{4_{in}}\}, v_7, v_4)$ now has become unreachable. So yielding corresponding leaf node v_7 is moved to $P(Z) = T$, as well as the incident edge, giving the new version of the SCC tree T and condensed graph $D(T)$ shown in the black rectangle in Figure 6.7. The node Z now is fine as without v_7, $G(Z)$ again is an SCC. Now, it has to be checked for T if this is the case as well. The function `findUnreachable` will find that condensed vertex $\{v_7\}$, with incident edge $(\{v_7\}, \{v_4, v_5, v_6\}, v_7, v_4)$ is still unreachable in $D(T)$ as well. This time T is a root node, so instead of moving leaf node v_7 to $P(T)$, it will become an independent SCC tree, and (v_7, v_4) becomes an edge between SCC trees. The bottom of Figure 6.7 shows G_{new} which is the graph of G_{old} where (v_6, v_7) is deleted and the resulting SCC trees which indeed represent the SCCs of G_{new}.

6.2.4 Deleting a set of vertices from an SCC tree

in [9] only an algorithm which deletes a single edge from a tree is given. This section will describe a function `removeVerticesFromTree(`D, T, X`)` which deletes edges of vertices in the set X in SCC tree T and updates the SCC tree T and the dynamic SCC decomposition D with as goal to remove any leaf node representing a vertex in X from T. This extra function is needed for Zielonka's algorithm where, after each iteration, the vertices in the attractor set have to be removed from the decomposition. To remove vertices from the decomposition their leaf nodes should be in an SCC tree of which all leaf nodes represent vertices that have to be removed. The function achieves this by removing leaf nodes of vertices which have to be removed, which are present in X, from the SCC tree T which also contains vertices that should stay.

The function `removeVerticesFromTree`, described in Alg 7, makes use of how the `UpdateSCC-Tree` function updates a node for the `deleteEdge` function. When vertices need to be removed from the dynamic SCC decomposition it will be necessary to first separate the leaf nodes representing these vertices from trees which also contain vertices that should not be removed. Assume there is a subtree rooted at node S in an SCC tree such that all leaf node descendants of S represent vertices that should be removed. Then removing all edges of vertices in the set X in the parent of S will make S unreachable in $D(P(S))$. By then moving S to the parent of the parent of S and repeating these steps, the subtree rooted at S is propagated upwards in the SCC tree until it becomes an independent tree in the decomposition. This independent tree with root node S can then be removed as a whole from the decomposition D.

The `UpdateSCC-Tree` function updates a tree bottom up. It starts at the changed node, and then updates all ancestors until there are no changes. For the `removeVerticesFromTree` function to be able to use this bottom up approach as well, it uses the function `topologicalSort-`

Algorithm 6 Deleting an edge from an SCC tree

```
function DeleteEdge(SCC-Tree $T$, $(u, v) \in E$)
    $N \leftarrow$ the node of $T$ containing $(u, v)$
    remove $(u, v)$ from $D(N)$
    UpdateSCC-Tree($N, \{u, v\}$)
end function

function UpdateSCC-Tree(SCC-Tree node $N$, $A \subseteq V(D(N))$)
    $(U, I) \leftarrow$ FindUnreachable($D(N), A$)
    if $U = \emptyset$ then
        return
    end if
    $d \leftarrow$ the split vertex for $N$
    $U' \leftarrow$ Merge($D(N)$)
    remove $(U', I)$ from $D(N)$
    if $D(N) = (\{d_{in}\}, \{d_{out}\}\}, \emptyset)$ then
        $D(N) \leftarrow$ Merge($D(N)$)
    end if
    $S \leftarrow$ subtrees of $N$ corresponding to elements of $U'$
    if $N$ is not the root node then
        add $U'$ and $I$ to $D(p(N))$
        for all $P \in S$ do
            set parent of $P$ to $p(N)$
        end for
        for all $u \in U'$ do
            for all $x \in u$ do
                $cv[x] \leftarrow$ vertex in $D(p(N))$ corresponding to $u$
            end for
        end for
        for all $v \in \bigcup U'$ do
            for all $(u, v) \in E(G)$ do
                if for some endpoint $x \in (u, v)$, $x \notin U'$ then
                    update edge $(u, v)$ in $D(p(N))$ replacing endpoint $x$ with $cv[x]$
                end if
            end for
        end for
        $v_N \leftarrow$ the vertex in $D((N))$ corresponding to $N$
        UpdateSCC-Tree($p(N), U \cup \{v_N\}$)
    else
        for all $P \in S$ do
            disconnect $P$ from $N$ and make a separate tree out of it
            $V_P \leftarrow$ vertices of the subgraph represented by $P$
            for all $v \in V_P$ do
                $SCC[v] \leftarrow P$
            end for
        end for
    end if
end function
```

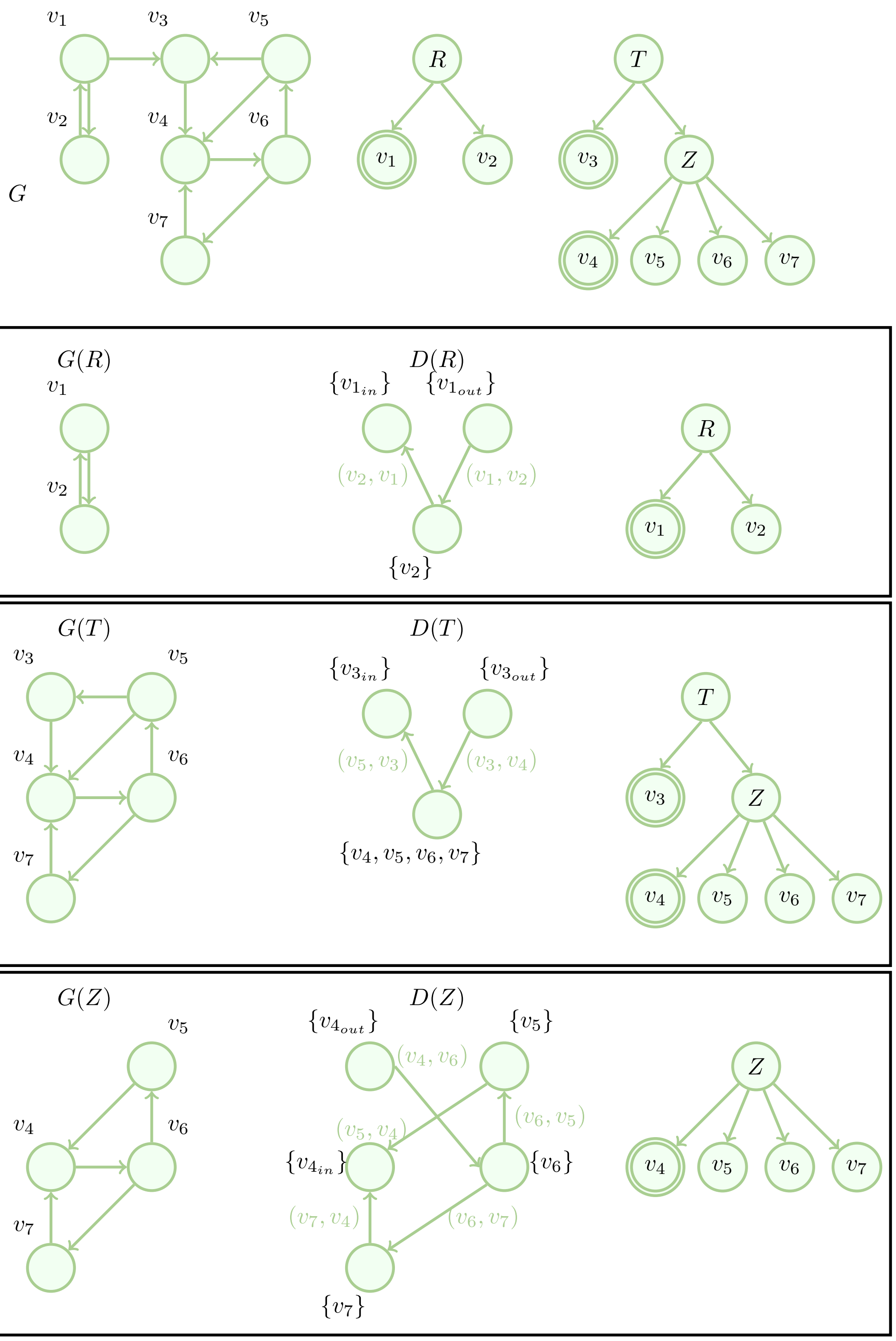


Figure 6.6: The black squares show $G(N)$ and $D(N)$ for every inner node N of the SCC trees.

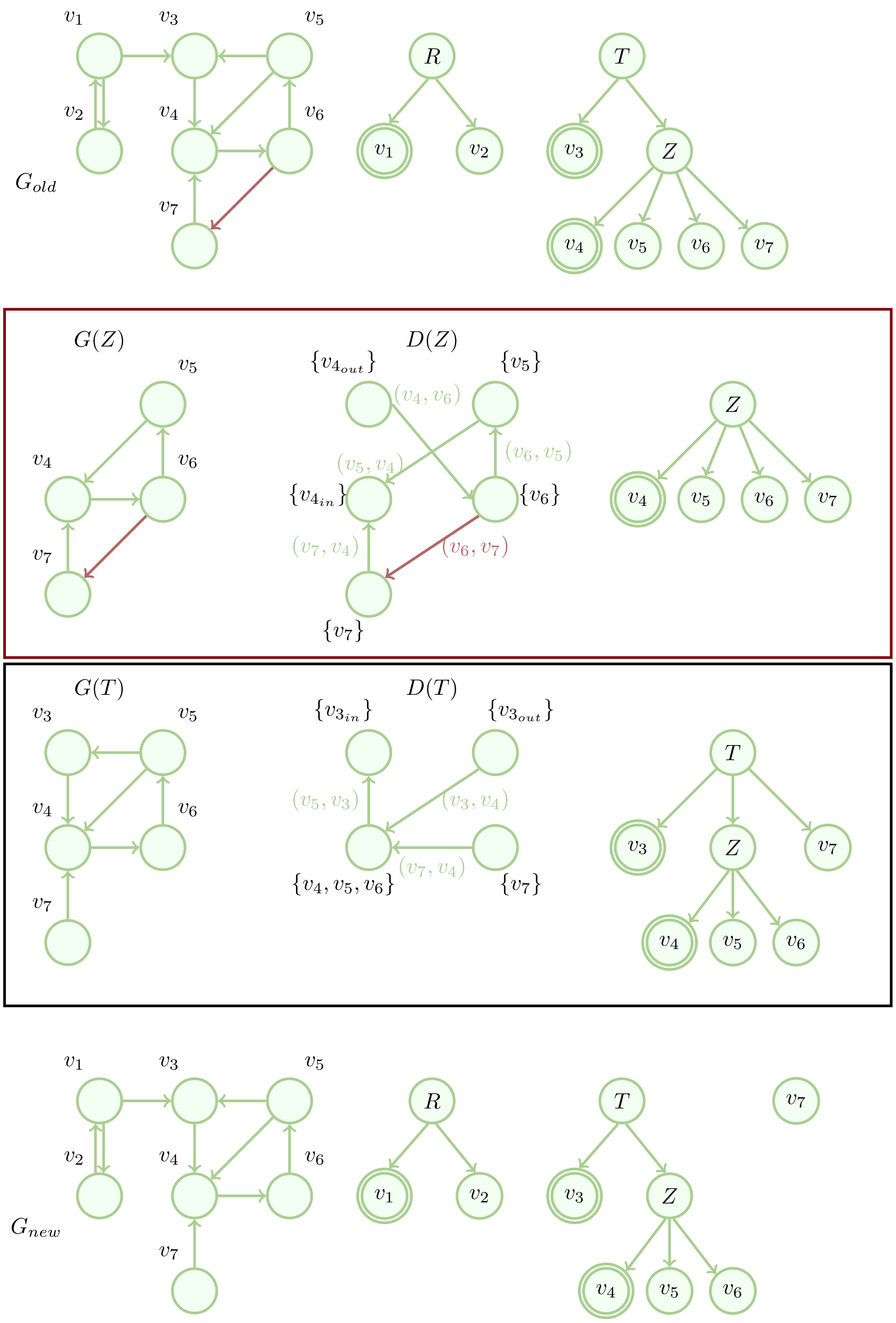


Figure 6.7: The black squares show the steps of deleting edge (v_6, v_7) from the SCC trees of graph G_{old} on top resulting in the SCC trees of G_{new} on the bottom.

Algorithm 7 Remove a set of vertices from an SCC tree

```
function removeVerticesFromTree(Dynamic Decomposition $D$, SCC-Tree $T$, $X \subseteq V$)
    $queue \leftarrow$ TopologicalSortAffected(root node of $T$, $X$)
    while $queue$ is not empty do
        $(p, M, r, l) \leftarrow queue.dequeue()$
        for all $x \in X$ do
            for all $(x, v) \in E$ do
                if $M$ contains $(x, v)$ then
                    remove $(x, v)$ from $D(M)$
                    $l \leftarrow l \cup \{x, v\}$
                end if
            end for
            for all $(v, x) \in E$ do
                if $M$ contains $(v, x)$ then
                    remove $(v, x)$ from $D(M)$
                    $l \leftarrow l \cup \{x, v\}$
                end if
            end for
        end for
        if $p \neq$ null then
            UpdateSCC-TreeNode($D, M, l, p.l$)
        else
            UpdateSCC-TreeNode($D, M, l$,null)
        end if
    end while
end function
```

`Affected(`N, X`)` to make a topological ordering of SCC tree with root node N. The pseudocode of `topologicalSortAffected(`N, X`)` is described in Algorithm 8. The `topological-SortAffected(`N, X`)` function removes any node of the topological ordering that either does not have a leaf node descendant representing a vertex in X or nodes for which all leaf node descendants represent vertices in X. The first can be removed because nodes only propagate upwards so they will not need to be updated. If all descendants of a node represent vertices in X then the subtree rooted at such node can be removed as a whole, so the node itself does not need to be updated, only its parent. Then the `removeVerticesFromTree` function uses the topological ordering to update the nodes with the `UpdateSCC-TreeNode` function starting at the end of the ordering. Algorithm 9 describes the `UpdateSCC-TreeNode` pseudocode. In this way it can be ensured that it is always the case that the nodes of descendants are updated before any nodes of their ancestors. Edges are always stored in the least common ancestor of both endpoints of the edge, so this means that subtrees that do not have a descendant representing a vertex in set X will be unaffected.

Example 5. As an example consider removing a set of vertices from the graph in the top left conder of Figure 6.8. If one would want to remove vertices v_2, v_7, v_8 and v_9 from the SCC tree of G, inner nodes A, B and R might have to be updated. Node C is unaffected since it does not contain any leaf node descendant representing v_2, v_7, v_8 or v_9. Node D contains only leaf nodes representing v_2, v_7, v_8 or v_9 so D does not need to be updated either since by updating B and R the subtree rooted at D will be removed from the tree as a whole.

To filter out those unaffected inner nodes that are not an ancestor of any of the vertices, the topological sort first executes a breath-first search on T starting at the root of T. During this breath-first search it stores a tuple (p, M, r, l, c) for each inner node M of T in the stack. Where p is a pointer to the tuple of the parent of M, r is a boolean which is set to true if the vertex of any of its leaf node children is present in X and c is the number of children of M for which all leaf

Algorithm 8 Make a topological sort of the nodes that should be updated to remove a set of vertices from the SCC tree

```
function TopologicalSortAffected(SCC-Tree node $N$, $(X \subseteq V)$)
    $stack \leftarrow []$ empty stack
    $queue \leftarrow \{(\text{null}, N, false, [], 0)\}$ fifo queue
    while $queue$ is not empty do
        $(p, M, r, l, c) \leftarrow queue.dequeue()$
        $stack.push(p, M, r, l, c)$
        for all subtrees $C$ of $M$ do
            $(V, E) = G(C)$
            if $|V| = 1$ and $V \subseteq X$ then
                $r \leftarrow true$
                $c \leftarrow c + 1$
            else if $|V| > 1$ then
                $queue.enqueue((p, M, r, l, c), C, false, [], 0)$
            end if
        end for
    end while
    while $stack$ is not empty do
        $(p, M, r, l, c) \leftarrow stack.pop()$
        if r then
            if $c =$ nr of subtrees of $M$ then
                if $p \neq$ null then
                    $p.r \leftarrow$ true
                    $p.c \leftarrow p.c + 1$
                end if
            else
                if $p \neq$ null then
                    $p.r \leftarrow$ true
                end if
                $queue.enqueue((p, M, r, l, c))$
            end if
        end if
    end while
    return $queue$
end function
```

Algorithm 9 Update a single node and find the affected nodes of the parent

```
function UpdateSCC-TreeNode(Dynamic Decomposition $D$, SCC-Tree node $N$, $A \subseteq V(D(N))$, $L$)
    $(U, I) \leftarrow$ FindUnreachable$(D(N), A)$
    if $U = \emptyset$ then
        return
    end if
    $d \leftarrow$ the split vertex for $N$
    $U' \leftarrow$ Merge$(D(N))$
    remove $(U', I)$ from $D(N)$
    if $D(N) = (\{d_{in}\}, \{d_{out}\}\}, \emptyset)$ then
        $D(N) \leftarrow$ Merge$(D(N))$
    end if
    $S \leftarrow$ subtrees of $N$ corresponding to elements of $U'$
    if $N$ is not the root node then
        add $U'$ and $I$ to $D(p(N))$
        for all $P \in S$ do
            set parent of $P$ to $p(N)$
        end for
        for all $u \in U'$ do
            for all $x \in u$ do
                $cv[x] \leftarrow$ vertex in $D(p(N))$ corresponding to $u$
            end for
        end for
        for all $v \in \bigcup U'$ do
            for all $(u, v) \in E(G)$ do
                if for some endpoint $x \in (u, v)$, $x \notin U'$ then
                    update edge $(u, v)$ in $D(p(N))$ replacing endpoint $x$ with $cv[x]$
                end if
            end for
        end for
        $v_N \leftarrow$ the vertex in $D((N))$ corresponding to $N$
        $L \leftarrow L \cup U \cup \{v_N\}$
    else
        for all $P \in S$ do
            disconnect $P$ from $N$ and make a separate tree out of it in $D$
            $V_P \leftarrow$ vertices of the subgraph represented by $P$
            for all $v \in V_P$ do
                $SCC[v] \leftarrow P$
            end for
        end for
    end if
end function
```

node descendants of the subtree rooted at the child represent vertices in X. Lastly l is an empty `ListSetPair` which can later be used to store the children of M that have been changed after an update on any of the children, or because of edge removal. Because of the breath-first traversal through the tree, popping items from the stack will always pop a child before any ancestor of the child. The tuples are popped one by one and for each tuple (p, M, r, l, c) if r is true then M is an ancestor of a leaf node in the set of vertices. If c is equal to the number of children of M then all leaf node descendants represent vertices in X, so increment $p.c$. If c is not equal then add the tuple to the queue. Since children are popped before their ancestors, they are also added to the queue before their ancestors, so they will be dequeued from the queue before their ancestors. Since M is an ancestor, $p.M$ is an ancestor as well, so set $p.r$ to true so that it is recognized as one by the time p is popped from the stack. If r is not true it means it is not an ancestor of a leaf node in the set of vertices because if it was, then either r would be true because it has the leaf node as child, or any of its children which are all popped earlier would have set r to true by transitivity. So by filling the queue in this way, `topologicalSortAffected` returns a queue of inner nodes such that any inner node in the tree which has leaf nodes descendants representing vertices that are in X and that aren't in X are present in the queue and any descendant is present before its ancestor.

Using this queue returned by `topologicalSortAffected` all relevant nodes can be updated bottom up. For each tuple (p, M, r, l, c) in the queue, first all edges in M that either have as source or as target a vertex in the set X are removed. The children of M that contain any of the endpoints of the removed edges are added to l. Then `UpdateSCC-TreeNode(`D, M, A, L`)` is called to update M. This function works exactly the same as `UpdateSCC-Tree` except that instead of recursing on arguments $p(N), U \cup \{v_N\}$ it adds $U \cup \{v_N\}$ to L. Here L is $p.l$ of the tuple, the list of changed vertices of its parent. Once it is time to dequeue the tuple of the parent the parent node will be updated. This way all nodes in the queue will be updated until all ancestors are updated and any subtree with leaf node descendants representing vertices in the set of vertices that have to be removed that were present in T have all become independent trees in X.

Example 6. As an example of the process of removing a set of vertices a graph,consider removing the vertices v_2, v_7, v_8 and v_9 from the SCC tree of graph G in Figure 6.8.

- First for the topological sort, a tuple for each inner node breath-first would be added to a stack, so the stack contains tuples for the inner nodes R, A, B, C and D with the tuple of D on top. The tuples of A and D would have r set equal to true since they contain the leaf node of one of the vertices. The tuple of D would be the only tuple for which c is equal to its number of children. When popping the tuple of D it will not be added to the queue since c is equal to its number of children so this whole subtree can be removed. Increment $p.c$ and set $p.r$, which is the tuple of B, to true. When C is popped next, r is false, so it is not added to the queue. Then the tuple of B is popped, r has become true for this tuple, so it is added to the queue and $p.r$ which is the tuple of R will be set to true. Next the tuple of A is popped, r is true so it is added to the queue and $p.r$ which is the tuple of R will be set to true. Lastly the tuple of R is popped, which has value true for r so it is added to the queue as well, but because it is the root node it has no parent tuple so no other r to set to true. In this way the queue ends with the tuples of nodes B, A and R in that order.
- Next these tuples are dequeued one by one, the edges are removed and the nodes are updated.
  - First the tuple $(p, B, true, [])$ is dequeued. The inner node B stores the edges $\{(\{v_6\}, D, v_6, v_7), (D, \{v_6\}, v_8, v_6)\}$. Both edges have an endpoint in $\{v_2, v_7, v_8, v_9\}$ so they removed and added to $l = [\{v_6\}, D]$. Then B is updated with $l = [\{v_6\}, D]$ resulting in the tree in the black square in Figure 6.8. In the update all vertices were unreachable, so the inner node D was moved to the parent R and since B had only the split node left it was replaced with a leaf node for v_6. These 2 nodes are then added to $p.l$, the `ListSetPair` of affected vertices of B.
  - Second the tuple $(p, A, true, [])$ is dequeued. The inner node A stores the edges $\{(\{v_2\}, C, v_2, v_3), (C, \{v_2\}, v_4, v_2)\}$. All these edges have an endpoint in $\{v_2, v_7, v_8, v_9\}$ so all

are removed, and all children are added to l. Then B is updated with $l = [\{v_2\}, C]$ resulting in the tree in the red square in Figure 6.8. In the update all vertices were unreachable, so the subtree C was moved to the parent R and since A had only the split node left it was replaced with a leaf node for v_2. These 2 nodes are then added to $p.l$, the `ListSetPair` of affected vertices of B.

– Last the tuple $(\text{null}, R, \mathit{true}, [\{v_6\}, D, \{v_2\}], C])$ is dequeued. The inner node R stores the edges $\{(\{v_1\}, \{v_6\}, v_1, v_6), (\{v_2\}, \{v_1\}, v_2, v_1), (D, C, v_7, v_4)\}$. Only $(\{v_2\}, \{v_1\}, v_2, v_1)$ and (D, C, v_7, v_4) have an endpoint in $\{v_2, v_7, v_8, v_9\}$ so they are removed and the endpoints are added to l. Then R is updated with $l = [\{v_6\}, C, D, \{v_2\}]$ resulting in the trees in the purple square in Figure 6.8. In the update all vertices were unreachable so they are all made into separate trees in the decomposition. Since R only has its split node left it was replaced with a leaf node for v_1.

After R is updated all nodes that have to be update in the tree are updated, so the decomposition has reached its final state for this call to `removeVerticesFromTree`. In Figure 6.8 the final state of the decomposition is the one in the purple square where the decomposition now contains single leaf node trees for v_1, v_2 and v_6 and 2 SCC trees with root C and root D. Now there are no longer leaf nodes representing vertices in $\{v_2, v_7, v_8, v_9\}$ in the same tree as leaf nodes representing vertices not in $\{v_2, v_7, v_8, v_9\}$.

Theorem 6. The `removeVerticesFromTree(`D, T, X`)` function can run in $O(|X||E|\delta)$ time.

Here a short proof of this complexity is given and in subsection D.3 an implementation of `removeVertices- FromTree(`D, T, X`)` is described with a more detailed proof that this implementation meets the complexity.

Proof. The `removeVerticesFromTree(`D, T, X`)` first makes a call to `topologicalSortAffected`. This function considers every node in the tree twice. Once to order them and once to check if they are an ancestor of a leaf node representing a vertex that should be removed. All actions done for each node take constant time and there are at most $O(|V|)$ nodes, so the `topologicalSortAffected` function will return a topological order in $O(|V|)$ time. There will be at most $O(|X|\delta)$ nodes in the queue since every leaf node representing a vertex in X has at most δ ancestors and non ancestors have not been added to the queue by `topologicalSortAffected`. For every node in the queue all edges with endpoints in X are removed. To remove the edges all edges in the node need to be checked. To remove all edges for all nodes in the queue will take $O(|E|)$ time. Each node then needs to be updated with `UpdateSCC-TreeNode` which just like `UpdateSCC-Tree` takes $O(|E|)$ time per node. So updating all nodes takes $O(|X||E|\delta)$ time which is the most expensive step. So the `removeVerticesFromTree(`D, T, X`)` runs in $O(|X||E|\delta)$ time. □

6.2.5 Deleting an SCC tree from a dynamic SCC decomposition

When an SCC tree is deleted from a dynamic SCC decomposition, several maps that store auxiliary values for the decomposition have to be updated. This section gives a short description on which values have to be updated when removing an SCC tree from a dynamic SCC decomposition.

First the SCC itself is removed from the dynamic SCC decomposition. The decomposition stores for each vertex the root of the SCC tree in a hashmap. So for each vertex in an SCC tree this value has to be deleted.

If the SCC tree has inner nodes, these might store edges. There is a hashmap that stores for each edge in which inner node they are present. Since the tree is to be deleted, these edges will no longer be stored anywhere so they have to be deleted from this hashmap.

There might be edges between SCC trees that have an endpoint in this SCC tree. These edges need to be removed as well since the vertices in this SCC tree will no longer exist.

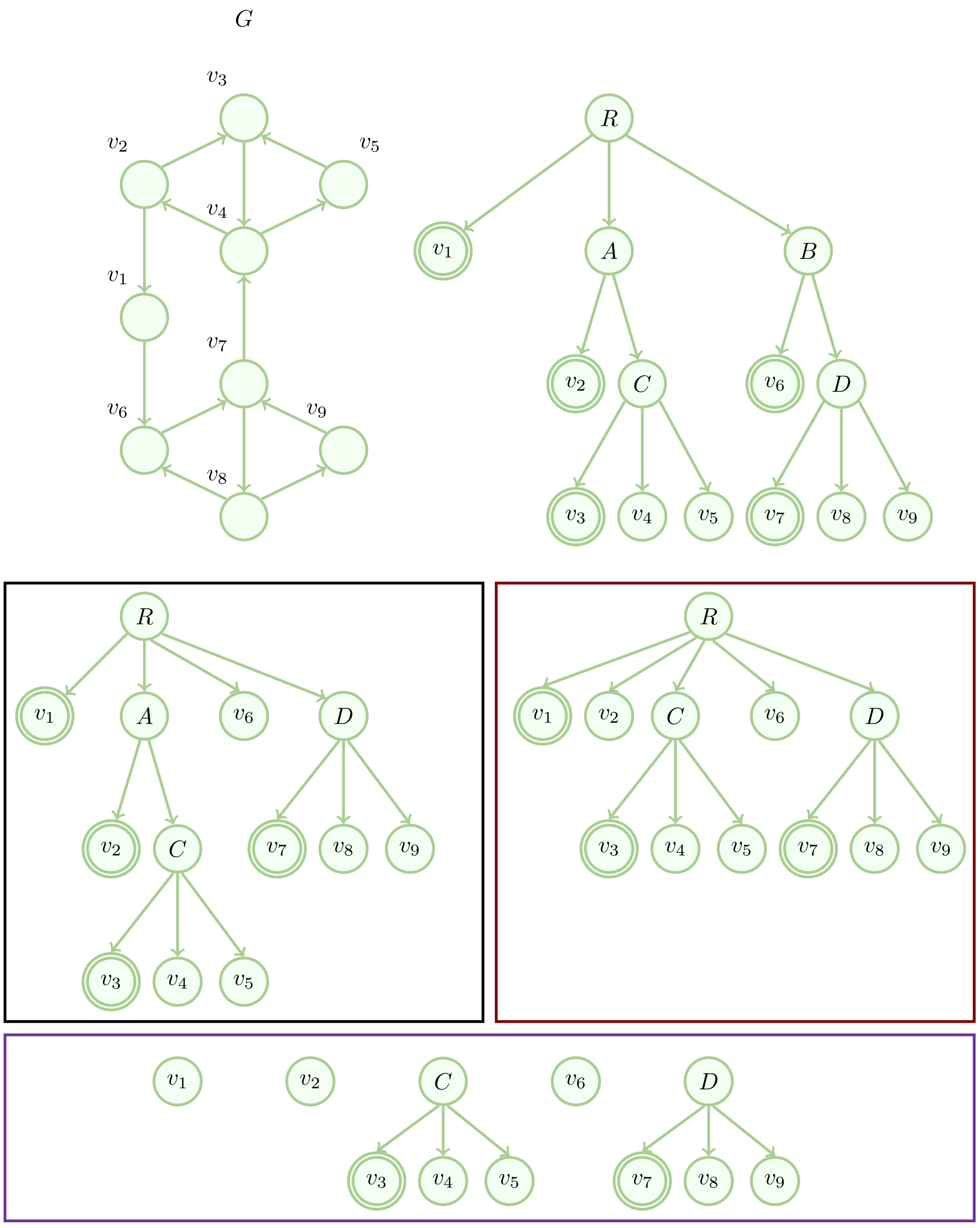


Figure 6.8: When deleting the edges of vertices v_2, v_7, v_8 and v_9 inner nodes A, B and R might have to be updated while C is unaffected.

Chapter 7

Experimental evaluation

Several experiments were executed to answer the research questions defined in chapter 4. These experiments and their results are described in this chapter.

In the experiments 3 versions of Zielonka's algorithm were compared:

- Zielonka's algorithm with Tarjan's algorithm as described in the literature. (version Tarjan)
- Zielonka's algorithm with partial re-decomposition, using Tarjan's algorithm for SCC decomposition. (version partial)
- Zielonka's algorithm with dynamic SCC maintenance, using Tarjan's algorithm to build the SCC trees. (version dynamic)

Each version was implemented in Java. A detailed description of the implementation can be found in Appendix D and E. In Appendix C a description how to use the program with commandline or unit tests is given.

The game data sets used for the experiment is described first in section 7.1. The first experiment, described in section 7.2 is an experiment which explores different possible choices that can be made to build the tree. The outcome of this experiment will be used to build the tree in the experiments described in sections 7.3 and 7.4. Section 7.3 takes a look at the difference in the number of vertices decomposed when running Zielonka's algorithm with Tarjan's algorithm or partial re-decomposition and also makes a comparison with the number of nodes updated in the SCC trees. The last experiment, described in section 7.4 measures the runtime differences between running Zielonka's algorithm with Tarjan's algorithm for SCC decomposition, partial re-decomposition or using dynamic SCC maintenance.

The experiments were executed using unit tests in IntelliJ. They were executed on a laptop with Java 12, windows 10, 8GB RAM and Intel(R) Core(TM) i7-6700 HQ CPU @ 2.60GHz 2.59GHz.

7.1 Game data sets

Most games tested in the experiments came from the benchmark of Keiren, described in [8], from now on called the benchmark. This benchmark contains the equivalence checking benchmark, MLSolver benchmark, modelchecking benchmark and PGSolver benchmark. Depending on the experiment some or all of the games in the benchmark will be tested and their result put in the digital repository Zenodo (`https://doi.org/10.5281/zenodo.5338737`). The result section will show a selection of the results of each benchmark to support the conclusions described in subsections 7.2.3, 7.3.3 and 7.4.3.

The data sets chosen for the result section are chosen based on 2 characteristics: the size (depth) of the tree and the number of Zielonka recursions/iterations. The size of the tree was divided in short trees, usually just depth 1, so an inner node with a bunch of leaf nodes, and long trees which were usually over depth 100. Data sets were chosen for the combinations:

- Short trees and few Zielonka recursions.
- Short trees with many Zielonka recursions.
- Long trees with few Zielonka recursions.
- Long trees with many Zielonka recursions.

The first set of games are the **random games**. A part of these games came from the PGSolver benchmark[8] and a part was made using the PGSolver tool [5]. Random games usually only have a few (< 10) Zielonka iterations. These games were chosen because in the PGSolver tool it is possible to control the minimum and maximum number of *outedges*. By choosing the maximum number of *outedges* as 1, games with very short trees are created. Increasing the maximum number of *outedges* to 2 already gives a long tree, but by increasing the maximum number *outedges* even more creates even longer trees. The games that were created for this experiment either have a maximum of 1, 2, 5 or 20 *outedges* and their sizes are 1k, 5k, 10k and 50k vertices. The games with a maximum of 20 *outedges* came from the benchmark, the others were created with the PGSolver tool. Since these games are random, 25 games of each type were created with the PGSolver tool, and the results shown in any result section are the average of these 25 games.

The second set of games are the **equivalence games**, a few games taken from the Equivalence checking benchmark. This benchmark encodes the problem of testing if 2 processes L_1 and L_2 are process equivalent. Four types of equivalences are considered: branching bisimulation, branching simulation, strong bisimulation and weak bisimulation. The equivalence is tested for several protocols and data sizes [8]. The protocols can be found in the name of the games. From this benchmark 9 games were chosen for the result section of the "number of vertices decomposed" experiment and "runtime differences" experiment. All chosen games have long (100+) trees, 4 games have very few (2) Zielonka recursions/iterations, and 5 games with slightly more (10-40) Zielonka recursions/iterations.

The third set of games are the **PGSolver games**, which are taken from the PGSolver benchmark. The games taken are the modelcheckerladder games. The dynamic SCC decomposition of these games consists of a single SCC tree of size 1, so a very short tree. So this tree is a single inner node with all vertices as leaf node children. These games have very few (2) Zielonka recursions/iterations.

The fourth set of games are the **modelchecking games**, which are taken from the modelchecking benchmark. The games in this benchmark encode the problem which checks if a model L satisfies a property φ. The models considered come, among other things, from a number of communication protocols. The properties are fairness, liveness and safety properties [8]. Of this benchmark 9 games were chosen for the result section of the "number of vertices decomposed" and "runtime differences" experiment. Of these the dynamic SCC decomposition of 4 games had relatively short (3-27) trees and 5 games had very long (1000+) games. Of the first 4 games, 2 games had very few (2) Zielonka recursions/iterations and 2 games slightly more (18). Of the latter 5 games had many (40+) zielonka iterations. The "split node choice" experiment used a different set of games from this benchmark to limit the time needed for this experiment. The games for this experiment had medium sized trees (7-449) in the dynamic SCC decomposition and slightly more (7-24) Zielonka recursions/iterations.

The fifth set of games are the **MLSolver games**, which are taken from the MLSolver benchmark. The games chosen from this benchmark all have slightly more Zielonka recursions/iterations than the size of the tree.

The sixth and last set of games are the **triangle games**. These games follow the structure of the game shown in Figure 4.1 in the problem statement. For every priority there are 3 vertices in a cycle and owned by player 0 if priority is even and owned by player 1 is the priority is odd. In every triangle there is 1 vertex with an *inedge* from a vertex with a lower priority and an *outedge* to a vertex with a higher priority. The other 2 vertices in the triangle only have edges within the triangle. These games should have short trees of 3 nodes and as many Zielonka iterations as there are triangles. Increasing the number of triangles should show the beneficial effect of both partial re-decomposition and dynamic SCC maintenance.

7.2 Experiment: split node choice

The paper [9] does not explicitly tell which vertex to choose for the split node to build a shorter tree. In the experiment in this section we will explore several options in choice of split node. First subsection 7.2.1 will describe the details of the experiment. Subsection 7.2.2 shows the results and subsection 7.2.3 discusses these results.

7.2.1 Experiment description

This experiment will explore several options in choosing the vertex to become the split node of an inner node. First the options are described and a rationale is provided why these options were chosen. This is followed by a description of the aspects that were measured and the games that were used for the measurements.

Experiment description: options

This experiment tested 4 options for choosing a vertex for the split node:

- The vertex that is first in the list.
- The vertex with the highest priority.
- The most connected vertex.
- A combination of the previous two: the most connected vertex with the highest priority.

Each of these choices will be discussed as for why they were chosen.

The first choice of split node is to just take whichever vertex happens to be first in the list of vertices of inner node and take this vertex for the split node. This is a fast and easy deterministic choice which can be executed in $O(1)$ time.

The second choice of split node is to take a vertex with the highest priority. The reason for this choice would be that Zielonka's algorithm always first removes the vertices with the highest priority and the vertices in the attractor set from the tree. Therefore if vertices with a higher priority are located higher in the tree, less inner nodes will need an update when these vertices are removed. Taking the vertex with highest priority was done by checking the priority of each vertex, thus taking $O(|V|)$ time. This is still in line with the $O(m)$ time allowed to build a layer of the tree.

The third choice of split node is to take the most connected vertex, so the vertex with the most *inedges+outedges*. The idea behind this choice is that if a vertex is very connected, splitting this vertex would most likely leave the graph in an inner node less connected and therefore splitting it apart in more smaller SCCs which will result in a smaller SCC tree. Like with choosing the vertex with the highest priority, choosing the most connected vertex takes $O(|V|)$ time as the connectivity of each vertex in the graph of the inner node will have to be checked.

The fourth choice was added based on the test results of the first 3 options. As can be read in subsections 7.2.2 and 7.2.3, both choosing the vertex with the highest priority and choosing the most connected vertex give good results. A last and fourth choice of split node is to combine these 2 options. Since choosing the most connected vertex gave the best results, the fourth choice of split node first chooses the most connected vertex, and if a vertex with the same connectivity but with a higher priority is found then that vertex is chosen instead. This choice of split node still only takes a single iteration over all vertices in the graph of the inner node, thus taking $O(|V|)$ time. This fourth choice of split node is considered the second iterations of this experiment as it was added based on the results of the other 3 choices. In the result section it will only be compared to the results of choosing the most connected vertex as the first part of this experiment showed that this was the best option.

Experiment description: measurements

The experiment measures effect of the choice of the split node on the following aspects:

- The size of the tree.
- The number of nodes updated.
- The time the algorithm spents building the tree.
- The time the algorithm spents updating the dynamic decomposition, which consists of both the time to update the nodes and the time to remove trees from the decomposition.
- The total runtime of Zielonka's algorithm with dynamic SCC maintentance, when the SCC trees in the dynamic decomposition are built using a specific option for selecting the vertex for the split node.

Because choosing the most connected vertex as the split node was expected to result in smaller trees the size of the tree was chosen as measurement. This choice no longer takes a constant time to choose the vertex but might make up in time because of the size of the tree so fewer layers will be built. Therefore the time the algorithm spents building the tree is measured as well.

The idea behind choosing the vertex with the highest priority was that fewer updates of the tree might be needed. For this the number of nodes that are updated and the time the algorithm spents on updating the decomposition are chosen as measurement. These numbers might also be smaller when choosing the most connected vertex if it really has smaller trees, since there are fewer ancestors to update when a vertex is removed.

The last and most general measurement is the total runtime of Zielonka's algorithm. This measurement is the overall goal: to make Zielonka's algorithm run as fast as possible. For games with very short running times where this measurement measures with millisecond precision, the running time might be influenced by the Java garbage collector or just-in-time compilation. Therefore for runs with runtime less than 1 minute, the game is solved 10 times and the average of the 6 fastest runs will be taken to exclude any influences of Java on the outcome. For a runtime between 1-3 minutes the game is solved 5 times and the average of the 3 fastest runs is taken. If the run time is over 3 minutes the game will just be solved once since a difference of 1 or 2 second becomes trivial here.

Experiment description: data sets

This experiment was a preliminary experiment whose results would be used in the other 2 experiments, so not the whole benchmark was tested. The sets used for this experiment were a subset of the random games, MLSolver games and modelchecking games. These sets were chosen for their gradual increase in tree size in the dynamic SCC decomposition with the increase of the size of the game. This allows to see the effect of the split node choice for different tree sizes for similar games. Next to these sets a subset of games of the PGSolver games was used as well because the trees of these games have a size of 1. This allows to check if a bad choice of split node could increase their tree size.

7.2.2 Results

For the random games, table 7.1 will show the results of measurements:

- Whichever vertex happens to be first in the list. (F)
- The vertex with the highest priority. (P)
- The most connected vertex. (C)

And table 7.2 show the results of measurements for the split node choices:

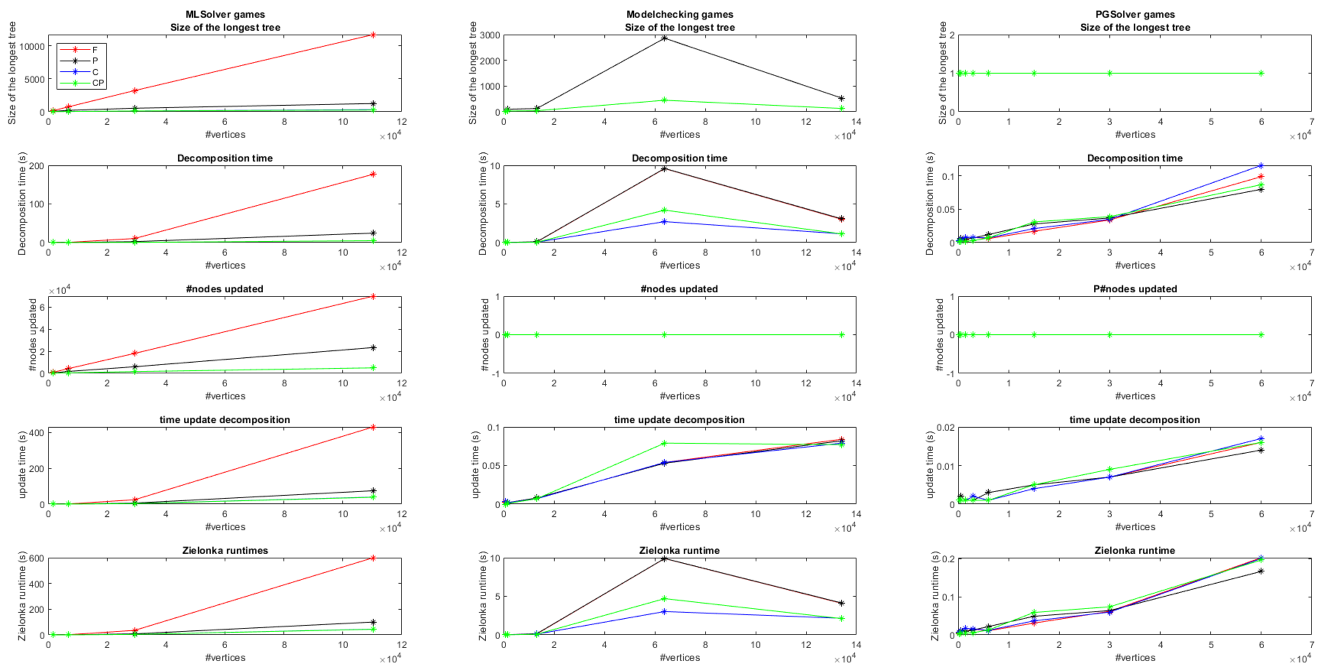


Figure 7.1: The results of the split node experiment for the MLSolver (1st column), modelchecking (2nd column) and PGSolver games (3rd column). Whenever the red is not visible it is hidden behind black. Whenever blue is not visible it is hidden behind green.

- The most connected vertex (C) which had the best results in the first part, and
- A combination of the two: the most connected vertex with the highest priority. (CP)

The combined results of the other games are shown in Figure 7.1.

7.2.3 Conclusions

In this section we first look at the effect of the split node choice on the size of the tree and the decomposition time. Then we take a look at the effect on the number of nodes updated and the total decomposition time. Lastly we look at the effect of the split node choice on the runtime of Zielonka's algorithm.

Conclusions part 1: size of the tree

It was expected that by choosing the most connected vertex as the split node, the size of the trees would be shortest and that saving the time to build those extra layers compensates the extra time to choose the vertex.

The PGSolver games were included in this experiment to see if any choice would make very short trees longer. As Figure 7.1 shows, none of the choices of split node gives a tree longer than 1, so clearly such thing did not happen. The decomposition time was roughly the same as well, so going through all vertices once when making the table did not really take extra time. So **in terms of tree size and decomposition time for games with only very short trees in their dynamic SCC decomposition it does not matter which vertex is chosen as split node**.

The other sets of games tested in this experiment had trees of varying sizes. The set of random games were games without any particular structure. The set of MLSolver games and the set of

Random games									
general information				avg size longest tree			avg decomposition time		
max *#outedges*	#vertices	avg #trivial SCCs	avg #non trivial SCCs	F	P	C	F	P	C
1	1000	971	3	1	1	1	0.002s	0.002s	0.002s
2	1000	415	2	227	105	**81**	0.027s	0.016s	**0.013s**
5	1000	62	1	695	416	**295**	0.133s	0.090s	**0.064s**
20	1000	0	1	920	797	**617**	0.3s	0.292s	0.**230s**
1	5000	4902	4	1	1	1	0.006s	0.006s	0.007s
2	5000	2082	2	1175	525	**415**	0.694s	0.324s	**0.244s**
5	5000	296	1	3557	2093	**1516**	4.014s	2.991s	**1.851s**
20	5000	0	1	4629	3978	**3078**	9.239s	15.113s	**7.236s**
1	10000	9858	4	1	1	1	0.01s	0.014s	0.01s
2	10000	4171	1	2362	1044	**836**	3.098s	2.591s	**1.146s**
5	10000	591	1	7145	4191	**3038**	29.381s	21.731s	**9.019s**
1	50000	49747	5	1	1	1	**0.064s**	0.09s	0.066s
2	50000	20805	1	11996	5301	**4223**	3m7s	1m26s	**53.184s**

	avg #nodes updated			avg time update decomposition			avg runtime Zielonka		
max *#outedges* (vertices)	F	P	C	F	P	C	F	P	C
1 (1000)	0	0	0	0.001s	0.001s	0.001s	0.004s	0.004s	0.004s
2 (1000)	278	136	**102**	0.037s	0.023s	**0.018s**	0.067s	0.041s	**0.034s**
5 (1000)	1373	835	**593**	0.335s	0.213s	**0.169s**	0.474s	0.311s	**0.240s**
20 (1000)	1814	1571	**1218**	0.278s	**0.258s**	0.266s	0.586s	0.559s	**0.505s**
1 (5000)	0	0	0	0.003s	0.003s	0.003s	0.014s	0.014s	0.017s
2 (5000)	2319	1071	**826**	1.003s	0.43s	**0.31s**	1.724s	0.774s	**0.569s**
5 (5000)	7201	4281	**3122**	8.885s	6.041s	**4.35s**	12.959s	9.086s	**6.25s**
20 (5000)	9167	7886	**6087**	6.816s	9.269s	**6.012s**	16.125s	24.496s	**13.325s**
1 (10000)	0	0	0	0.004s	0.006s	**0.005s**	**0.022s**	0.031s	0.024s
2 (10000)	4824	2192	**1706**	4.233s	3.135s	**1.391s**	7.382s	5.783s	**2.585s**
5 (10000)	14227	8466	**6065**	1m7s	41.44s	**20.249s**	1m37s	1m3s	**29.408s**
1 (50000)	0	0	0	**0.025s**	0.037s	0.026s	**0.142s**	0.207s	0.144s
2 (50000)	17073	7460	**6126**	3m13s	1m19s	**48.672s**	6m18s	2m45s	**1m42s**

Table 7.1: The average results of the split node experiment part 1 for the random games.

Random games							
general information				avg size longest tree		avg decomposition time	
max *#outedges*	#vertices	avg #trivial SCCs	avg #non trivial SCCs	C	CP	C	CP
1	1000	971	3	1	1	0.002s	0.001s
2	1000	415	2	81	**72**	0.013s	**0.011s**
5	1000	62	1	295	**274**	0.064s	**0.063s**
20	1000	0	1	617	**610**	**0.230s**	0.238s
1	5000	4902	4	1	1	0.007s	**0.004s**
2	5000	2082	2	415	**385**	0.244s	**0.231s**
5	5000	296	1	1516	**1419**	1.851s	**1.78s**
20	5000	0	1	3078	**3051**	**7.236s**	10.071s
1	10000	9858	4	1	1	**0.01s**	0.009s
2	10000	4171	1	836	**780**	1.146s	**1.036s**
5	10000	591	1	3038	**2882**	**9.019s**	12.146s
1	50000	49747	5	1	1	**0.066s**	0.112s
2	50000	20805	1	4223	**3965**	**53.184s**	58.433s

		avg #nodes updated		avg time update decomposition		avg runtime Zielonka	
max *#outedges*	#vertices	C	CP	C	CP	C	CP
1	1000	0	0	0.001s	**0s**	0.004s	**0.003s**
2	1000	102	**90**	0.018s	**0.015s**	0.034s	**0.030s**
5	1000	593	**553**	0.169s	**0.156s**	0.240s	**0.226s**
20	1000	1218	**1207**	0.266s	**0.262s**	**0.505s**	0.510s
1	5000	0	0	0.003s	**0.001s**	0.017s	**0.009s**
2	5000	826	**770**	0.31s	**0.291s**	0.569s	**0.536s**
5	5000	3122	**2943**	4.35s	**4.136s**	6.25s	**5.957s**
20	5000	6087	**6036**	**6.012s**	8.635s	**13.325s**	18.817s
1	10000	0	0	0.005s	0.005s	0.024s	**0.021s**
2	10000	1706	**1586**	1.391s	**1.288s**	2.585s	**2.355s**
5	10000	6065	**5770**	**20.249s**	25.895s	**29.408s**	38.211s
1	50000	0	0	**0.026s**	0.044s	**0.144s**	0.253s
2	50000	6126	**5676**	48.672s	**47.294s**	**1m42s**	1m46s

Table 7.2: The average results of the split node experiment part 2 for the random games.

modelchecking games should follow a particular structure to solve their corresponding problems. First we look at the results of the tree size and tree build time of the MLSolver games and modelchecking games in Figure 7.1. For the modelchecking games the red line is hidden right behind the black line and the blue line is hidden right behind the green line. As expected choosing the most connected vertex gave the shortest trees. Sometimes with huge differences, up to 11K layers, compared to just choosing the first vertex in the list as split node. For example for the game `Nestern=6_compact` of the MLSolver games, choosing the first vertex gave a maximum tree size of 11693, while the largest tree when choosing the most connected vertex had only size 323. So a reduction of tree size by 97%. Surprisingly choosing the vertex with the maximum priority also sometimes reduces the tree size, but not as much as choosing the most connected node. When looking at the decomposition time, the benefits in tree size when choosing the most connected node outweigh the extra time spent to find this vertex. The decomposition time is almost always lowest when the most connected vertex is chosen as split node, and when not it is only a few $1/100^{th}$ of a second off. So **in terms of tree size and decomposition time for games with structure with varying tree sizes in their dynamic SCC decomposition choosing the most connected vertex clearly is the best choice**.

The set of random games (table 7.1) shows similar results to the MLSolver games and modelchecking games. In this set of games where the *outedge* varied, the size of the trees varied as well. This shows for games with the same size that a higher out degree increases the size of the longest tree. Although the benefit of choosing the most connected vertex as split node can be seen for these random games as well, it is not as large as for the MLSolver games and modelchecking games, though a decrease of tree size by more than half can still be seen. The tree size when choosing the vertex with the highest priority gives shorter trees compared to choosing the first vertex in the list as well, but these trees are still larger than when choosing the most connected vertex. The benefit is seen in the decomposition time as well: the larger the difference in tree sizes, the larger the differences in decomposition time can be seen as well. So the extra time spent in choosing the vertex is not a problem at all. So **in terms of tree size and decomposition time for games without structure with varying tree sizes in their dynamic SCC decomposition choosing the most connected vertex clearly is the best choice**.

Conclusion. So in conclusion when considering choosing the first vertex in the list, choosing the vertex with the highest priority and choosing the most connected vertex as split node, choosing the most connected vertex is the best choice when considering the size of the tree and the decomposition time.

Conclusions part 1: update and runtime

It was expected that choosing the vertex with the highest priority as a split node would give the least number of nodes updated. This was because in Zielonka's algorithm the highest priority vertices (and their attractor set) are always removed first from the SCC. So if those highest priority vertices are higher up in the tree then fewer ancestors have to be updated when they are removed. Choosing the most connected vertex as split node was expected to rank second in number of nodes updated, since shorter trees will give fewer ancestors to update.

When looking at the results of the sets of games with short trees (size 1), the PGSolver games Figure 7.1, no nodes are removed. So the short trees seem to be removed in their entirety and so no nodes are updated there. The time to update the decomposition, which contains both updating the nodes and removing the trees is almost equal for any of the vertex choices for the split node. The same goes for the runtime of Zielonka's algorithm. All 3 choices have about the same runtime. So **in terms of number of nodes updated, update time and Zielonka runtime, for games with only very short trees in their dynamic SCC decomposition it does not matter which vertex is chosen as split node**.

The other sets of games tested in this experiment had trees of varying sizes. The set of random games were games without any particular structure. The set of MLSolver games and the set of modelchecking games should follow a particular structure to solve their corresponding problems. First we look at the results of the MLSolver games and modelchecking games in Figure 7.1. Although the games in the modelchecking set have longer sized trees (length 21-33 for C or 166-11K for F) in their dynamic SCC decomposition, no nodes are updated during Zielonka's algorithm. So the trees are removed as a whole and therefore no big changes in the update time can be seen for any choice. The runtime of Zielonka's algorithm still does show a preference for choosing the most connected vertex because of the time saved when making the decomposition. For the modelchecking games when choosing the most connected vertex, Zielonka's algorithm saves between 18% and 69% in runtime compared to choosing the first vertex, with the exception of the outlier, the smallest game which has an increase in runtime of 13%, or 3 milliseconds. The games in the MLSolver set do have a lot of nodes updated during Zielonka's algorithm. Here the effect of the difference in tree size can be clearly seen. The bigger the difference in tree size the bigger the difference in the number of nodes updated. Choosing the most connected vertex, 82 to 93% less nodes need to be updated compared to choosing the first vertex. It was expected that choosing the vertex with the highest priority would result in the least number of nodes updated, however this effect is either overshadowed by the differences in tree size or not present at all. Choosing the vertex with the highest priority only gives a decrease between 59 to 67% in the number of nodes updated compared to choosing the first vertex. This difference in number of nodes updated can also be seen in the update time. For example for `Nestern=6_compact`, choosing the first vertex causes almost 70K nodes to be updated, taking roughly 7 minutes, while choosing the most connected vertex causes only 5K nodes to need an update taking only 39 seconds. The time saved when updating the decomposition and the time saved when building a decomposition with shorter trees, also clearly reflects in a decrease in runtime of Zielonka's algorithm when choosing the most connected vertex. So **in terms of number of nodes updated, update time and Zielonka runtime, for games with structure with varying tree sizes in their dynamic SCC decomposition, choosing the most connected vertex is the best choice out of the 3**.

For the random games (table 7.1) the choice of split node has a less big effect. The number of nodes updated is still least when choosing the most connected vertex, but the differences are not as big as for the MLSolver games. The decrease in number of nodes updated for the random games is only between 32 and 64%. This also reflects in the update time, where the update time is clearly least when choosing the most connected vertex, but it only saves seconds at most, not minutes like with the MLSolver games. The decrease in update time is highest for the random graphs with a maximum of 5 or more *outedges*, with a decrease of more than 50% in update time compared to choosing the first vertex. When looking at the runtime, the combined effect of some decrease in decomposition time and some decrease in update time, still shows a clear decrease in runtime of Zielonka's algorithm when choosing the most connected vertex. So **in terms of number of nodes updated, update time and Zielonka runtime, for games without structure with varying tree sizes in their dynamic SCC decomposition, choosing the most connected vertex is the best choice out of the 3**.

Conclusion. In conclusion when considering choosing the first vertex in the list, choosing the vertex with the highest priority and choosing the most connected vertex as split node, choosing the most connected vertex is the best choice when considering the number of nodes updated, update time and runtime of Zielonka's algorithm.

Conclusions part 2: size of the tree

Because both choosing the most connected vertex and choosing the vertex with the highest priority performed better than choosing the first vertex, another option was added: first choosing the most connected vertex, and then between all vertices with this highest connectivity choose the one with the highest priority. This new option will be called the mixed option in the rest of the discussion.

There was still no impact on the size of the short trees in the PGSolver games (Figure 7.1) and no real difference in the decomposition time for these games as well. The highest difference was 6 milliseconds. For the random games choosing the mixed option still gives slightly shorter trees. With the mixed option the size of the trees of the random games decrease between 0 and 258 nodes, or 0 to 11% with an average of 4%. The decomposition time does not really give any differences, sometimes choosing the most connected vertex is slightly faster and sometimes the mixed option is slightly faster, with differences up to 20% for both sides. But still this is talking about milliseconds. It seems that the time saved on building those extra few layers is about the same as any extra time spent on checking 2 parameters. The same is the case for the MLSolver and modelchecking games (Figure 7.1). For these, with the exception of 1 game, **the mixed option performs a bit better both in terms of tree size, with a decrease between 0 and 15%, and decomposition time, with on average a decrease of 20%.**.

Conclusion. In conclusion, just looking at the size of the longest tree and the decomposition time the mixed option would be the better choice but not by much.

Conclusions part 2: update and runtime

Choosing the mixed option gives a slight decrease in number of nodes updated for the random games and MLSolver games. This is a decrease between 0 and 300 nodes, or 0 and 7% with on average a decrease of 3%. The impact on the average time to update the decomposition is very small, most of the time the mixed option performs slightly better, with the exception of 3 games in the random games and the differences are really small. When the mixed option performs better, there is a decrease between 2 and 66%, with a decrease of 16% on average. So **for updating the dynamic SCC decomposition choosing the mixed option seems to have a slight advantage for some games but not by a lot**.

Comparing the runtime of Zielonka's algorithm for the sets of games there again does not seem to be a big difference between choosing the most connected vertex and the mixed option. For the random games one time the first is slightly faster and another time the second. These differences range between a increase of 30% and a decrease of 15%, with as median a decrease of 5%, when comparing the mixed option to the most connected vertex option. For the MLSolver and modelchecking games, the mixed option always performs a bit better, with the exception of 1 game. The decrease in the runtime of Zielonka's algorithm with the mixed option for these games is between 2 and 67%, with on average a decrease of 30%. The exception happened to be a game where the tree size did not decrease for the mixed option but stayed the same. So **in terms of the runtime of Zielonka's algorithm, choosing the mixed option performs a bit better**.

Conclusion. In conclusion the option of choosing the most connected vertex and the mixed option perform about the same, but most of the time choosing the mixed option performs just a bit better. For the rest of the experiments the mixed option will be used because this option performed best in case of the longest tree size and did not perform worse than the most connected vertex in case of runtime.

7.3 Experiment: number of vertices decomposed

The experiment in this section was done to answer the following research questions:

RQ2 What is the difference in the total number of vertices decomposed into SCCs when recalculating SCCs whose vertices have been removed from the graph compared to recalculating all SCCs?

RQ5 How many nodes in the SCC tree are updated during the entire run of Zielonka's recursive algorithm in comparison to the number of vertices decomposed by partial decomposition or decomposition of the whole graph.

First subsection 7.3.1 will describe the details of the experiment. Subsection 7.3.2 shows the results and subsection 7.3.3 discusses these results.

7.3.1 Experiment description

To answer research questions 2 and 5 this experiment is divided into 2 parts. The first part of the experiment will measure the number of vertices decomposed into SCCs during the run of Zielonka's algorithm. The second part will measure the total number of nodes that are updated in any of the SCC trees during the run of Zielonka's algorithm.

Experiment description: measuring the number of vertices

The idea behind partial re-decomposition (partial) was that by only running Tarjan's algorithm on the SCCs of which vertices have been removed, a lot of the work of Tarjan's algorithm can be saved. There is also some extra work however to find the vertices of those SCCs that should be re-decomposed. If this extra work takes longer than just running Tarjan's algorithm then partial re-decomposition is not worth it. So to compare these 2 methods and to answer research question 2 the following 5 points are measured:

- The cumulative number of vertices in graphs received by any call to Tarjan's algorithm when not using partial re-decomposition.
- The cumulative number of vertices in graphs received by any call to Tarjan's algorithm when using partial re-decomposition.
- The time spent on SCC decomposition when not using partial re-decomposition.
- The time spent on preparing the graph for the SCC decomposition when using partial re-decomposition.
- The time spent on the actual SCC decomposition when using partial re-decomposition.

Experiment description: measuring the number of nodes

When dynamic SCC maintenance is used, SCC decomposition is no longer required after every iteration or recursion of Zielonka's algorithm. However the SCC trees will need to be built for the game and updated during the run of Zielonka's algorithm. When updating an SCC tree node N, no specific vertices are addressed, just other nodes or condensed vertices of $D(N)$ so the measurement of the number of vertices cannot be used here. To see if using dynamic SCC maintenance is beneficial we measure the following aspects and compare them to the results of the first half of this experiment:

- The cumulative number of nodes updated of any SCC tree during the run of Zielonka's algorithm with dynamic SCC maintenance.
- The time it takes to build all SCC trees for the game.
- The total time dynamic SCC maintenance spends on the maintenance, which includes both the time to update nodes in trees and the time to delete trees.

Experiment description: data sets

This experiment was execute on all games of the benchmark. Because there are too many games to discuss in this report, the results of all games in the benchmark that could be tested are uploaded to the digital repository Zenodo (`https://doi.org/10.5281/zenodo.5338737`). In section 7.3.2 scatter plots of the results of games with more than 1000 vertices will be shown. Next to the games in the benchmark the set of random games generated by PGSolver was also tested, as well as the set of triangle games.

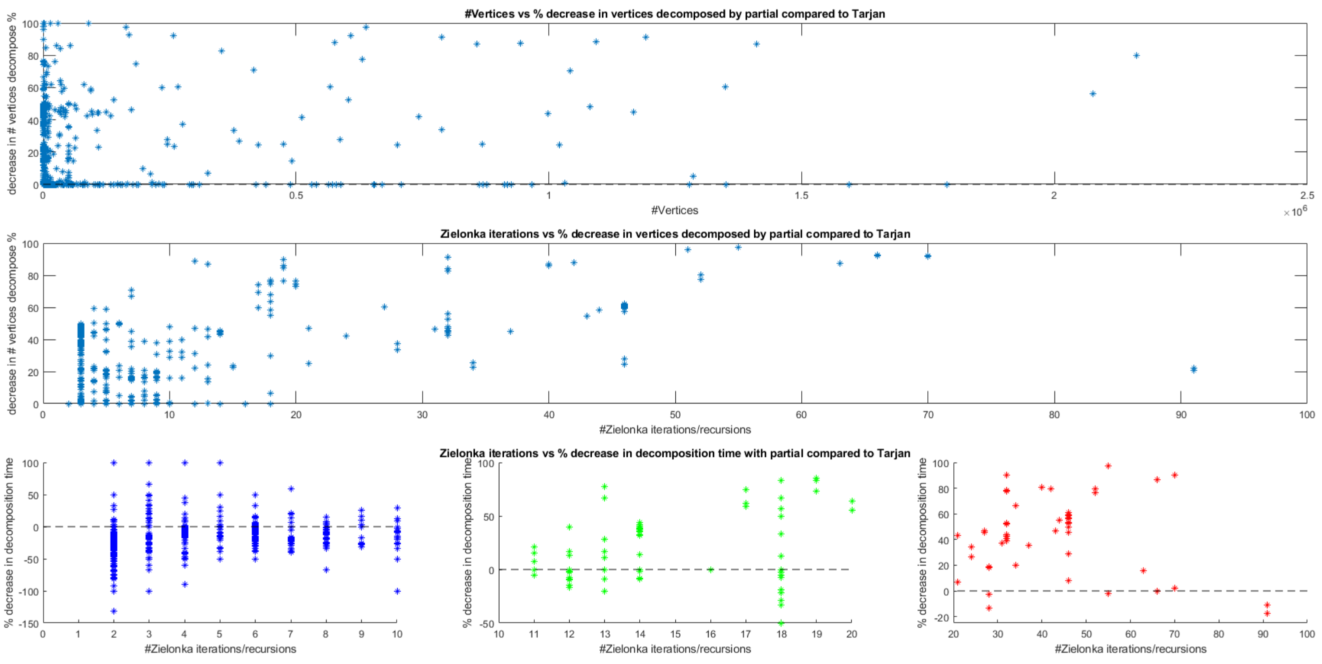


Figure 7.2: The results for all games with more than 1000 vertices for the "number of vertices decomposed" experiment, comparing partial re-decomposition with Tarjan.

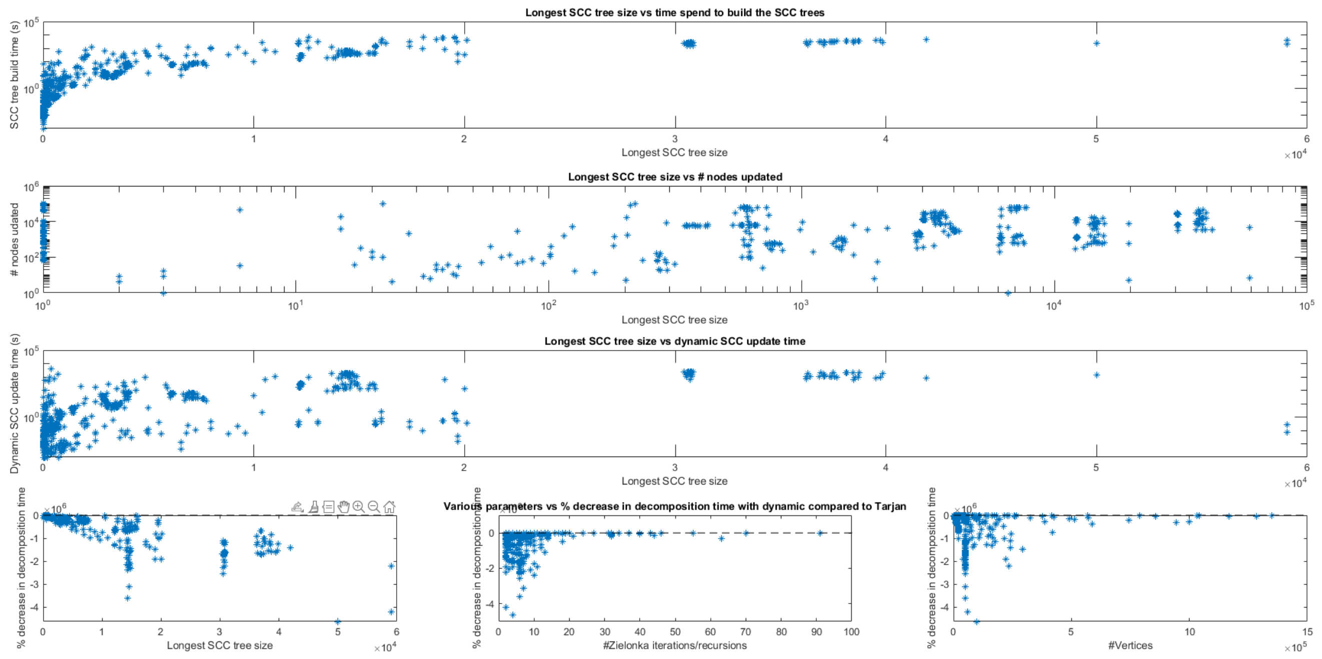


Figure 7.3: The results for all games with more than 1000 vertices for the "number of vertices decomposed" experiment, comparing partial dynamic SCC maintenance with Tarjan.

Triangle games								
#triangles	$\|V\|$	#trivial SCC	#non trivial SCC	size longest tree	#tarjan calls	#vertices tarjan	#vertices Partial	#nodes updated
50	150	0	50	1	51	3,8K	**150**	0
100	300	0	100	1	101	15K	**300**	0
500	1,5K	0	500	1	501	376K	**1,5K**	0
1000	3K	0	1K	1	1K	1,5M	**3K**	0
5000	15K	0	5K	1	5K	38M	**15K**	0
10000	30K	0	10K	1	10K	150M	**30K**	0
30000	90K	0	30K	1	30K	1350M	**90K**	0
100000	300K	0	100K	1	na	na	**300K**	0
300000	900K	0	300K	1	na	na	na	na

#triangles	$\|V\|$	size longest tree	#tarjan calls	time make tree	time update decom-position	time tarjan	time partial prep	time partial tarjan
50	150	1	51	0,001s	0,001s	0,005s	0s	0s
100	300	1	101	0,002s	0,001s	0,018s	0,001s	0,001s
500	1,5K	1	501	0,006s	0,003s	0,128s	0,002s	0,001s
1000	3K	1	1K	0,005s	0,002s	0,445s	0,006s	0,001s
5000	15K	1	5K	0,034s	0,007s	13,332s	0,184s	0,009s
10000	30K	1	10K	0,093s	0,017s	57,96s	0,816s	0,03s
30000	90K	1	30K	0,629s	0,093s	15m	9,708s	0,19s
100000	300K	1	na	8,518s	1,027s	na	5m7s	11,097s
300000	900K	1	na	1m7s	na	na	na	na

Table 7.3: The average results of the number of vertices decomposed experiment for the triangle games.

7.3.2 Results

Figures 7.2 and 7.3 show the results of measurements for the "number of vertices decomposed" experiment.

7.3.3 Conclusions

In this section we first look at the effect of using partial re-decomposition with Zielonka's algorithm on the number of vertices decomposed into SCCs. Then we check if the preparation work of partial re-decomposition does not take too much time to be beneficial. Then we compare the former results with the results of using dynamic SCC maintenance with Zielonka's algorithm to see if building SCC trees and updating the dynamic decomposition is beneficial compared to simply using Tarjan's algorithm and using partial re-decomposition.

In the upper 2 scatter plots of Figure 7.2 we compare the number of vertices decomposed with using partial re-decomposition to the number of vertices decomposed when using Tarjan's algorithm. Since partial re-decomposition was expected to decrease the number of vertices decomposed, these figures show the percentage of vertices saved:

$$\frac{(\#vertices\ decomposed\ Tarjan\ -\ \#vertices\ decomposed\ partial)}{\#vertices\ decomposed\ Tarjan} * 100$$

In the top figure his decrease in vertices decomposed is compared to the number of vertices in the game, to see if there is a higher decrease for bigger games. It can be seen that there is no relation between the decrease in number of vertices decomposed and the number of vertices in the game. There are small games with a decrease of almost 100% in the number of vertices, but also small games with no decrease at all. The same goes for big games, there are games with a decrease of over 90% but also games with no decrease at all. This effect of a high decrease can be seen most clearly in the triangle games (table 7.3), which regardless of size, 150 to 900K vertices, all have a decrease of more than 99% in the number of vertices decomposed. The towersofhanoi games from the PGSolver benchmark are the exact opposite, these games have 0.9K to 708K vertices, but a decrease of 0%. So **there is no relation between the number of vertices in the game and the decrease in number of vertices decomposed into SCCs during Zielonka's algorithm when using partial re-decomposition compared to Tarjan's algorithm.**

Since there was no relation between the decrease in the number of vertices decomposed with the number of vertices in the game, the second plot shows the relation between the decrease in the number of vertices decomposed and the number of iterations/recursions of Zielonka's algorithm needed to solver the game, up to 100 iterations. These results can be divided in 3 parts. The games with few iterations (0-10) show the least and most fluctuating decrease. The decrease is mostly between 0 and 60%. The games with some more iterations (10-20) only still contain a few games which show no decrease at all. The games that do show a decrease have a decrease between 20 and 90%. The games with a lot of iterations (20-100) always show a decrease in the number of vertices decomposed, and this decrease is between 20 and almost 100%. The decrease shows a tendency to increase with a higher number of iterations. So **with the increase of the number of iterations/recursions of Zielonka's algorithm there is an increase in the decrease in number of vertices decomposed into SCCs during Zielonka's algorithm when using partial re-decomposition compared to Tarjan's algorithm.**

Since there is a decrease in the number of vertices decomposed, we also need to check if the preparation work of partial re-decomposition does not take too much time to be beneficial even when there is a decrease. For this the scatter plots in the bottom row of Figure 7.2 were made. The results were divided in 3 scatter plots to be able to scale the results better. The plots show the decrease in time spent on decomposing SCCs in percentages:

$$\frac{(decomposition\ time\ Tarjan\ -\ decomposition\ time\ partial)}{decomposition\ time\ Tarjan} * 100$$

The first scatter plot with 0 to 10 iterations shows that with so few iterations, it is better for most games to just use Tarjan. The decrease in the number of vertices decomposed is not enough to outweigh the time spent on preparation in partial re-decomposition even if the actual time spent on preparation is very small. In the second plot which shows the results for games with 10 to 20 iterations, the shift is already visible. There are much fewer games where partial re-decomposition increases the time spent on decomposition, and when it does the relative increase is much smaller. Like with the decrease in number of vertices decomposed, with the increase of iterations there is also an increase in the relative decrease in time spent on decomposing SCCs. This trend is even more clear on the third figure, which shows the results for 20-100 iterations. There are only 5 games with a small increase in decomposition time, and a trend that shows an increase in the decrease if time spent on decomposing SCCs with partial re-decomposition. So **for games which take less than 10 iterations to solve, the preparation of partial re-decomposition often takes to much time to be beneficial, but for games which take more than 10 iterations to solve the time saved by decrease in the number of vertices decomposed is more than enough to outweigh the time spent on preparation.**

To compare the former results with the results of using dynamic SCC maintenance with Zielonka's algorithm Figure 7.3 shows several scatter plots for comparing dynamic SCC maintenance with the use of Tarjan's algorithm. For dynamic SCC maintenance, the total decomposition time is split between the time spent to build all SCC trees, which is done at the start of the first iteration of Zielonka's algorithm, and the time spent to update the dynamic SCC decomposition. Since both building has a factor δ, so size of the tree, in its complexity, the first plot of Figure 7.3 shows the relation between size of the tree and the time spent on building the SCC trees. Here it can be seen that indeed the longer the size of the longest SCC tree, the more time spent on building these trees. The time seems to stop increasing after a tree size of $3*10^4$, but this is not the case. This is the effect of the cut off time, after 2 hours the program will stop attempting to build the tree and the time measured will be set to 2h+. The results of the games whose SCC trees took more than 2 hours to build are not shown in the plot and therefore it just looks like the building time won't increase for even longer trees. So **with an increase of the size of the longest tree, there also is an increase in time to build the dynamic SCC decomposition.**

Since after deleting an edge, or when removing vertices, all ancestors of some nodes have to be updated, it was expected that for long trees there should be more nodes updated which will cost more time. To test this the second plot of Figure 7.3 shows the relation between the size of the tree and the number of nodes updated during Zielonka's algorithm and the third plot shows the relation between the size of the tree and the time spent on updating the dynamic SCC decomposition. Both plots are shown on log scale. In both plots can be seen that the expectations were both true. There were a few exceptions, for example there were a few games with very short trees but with high update times. These usually took many iterations of Zielonka's algorithm to solve. For example the game `DemriKillerFormulan=2` of the MLSolver benchmark which 34K vertices, a longest tree size of only 6, 121 trees with at least 1 inner node and took 37 iterations/recursions of Zielonka's algorithm to solve. However during Zielonka's algorithm it had to update 319 nodes. This is most likely caused by the recursion calls. Before the recursion call, a copy is made of the SCC tree of the SCC of that iteration, and attractor set A or B has to be removed from this copy to pass the updated dynamic SCC decomposition to the recursive call. But without a few outliers, **an increase in the size of the longest tree gives an increase in the number of nodes updated and the time spent on updating the dynamic SCC decomposition**.

The last row of plots in Figure 7.3 takes the total decomposition time of dynamic SCC maintenance, so the time spent to build the tree and the time spent on updating the decomposition. It then compares the decrease in decomposition time when using dynamic SCC maintenance compared to Tarjan's algorithm to the longest tree size, the number of iterations and the number of vertices in the game. From all 3 plots it can be clearly seen that there rarely is a decrease in decomposition time. For a few games with very short trees (<5) and many iterations (>20), using dynamic SCC maintenance gives a slight decrease in decomposition time. **For games with long trees, the longer the tree the higher the increase in decomposition time when using dynamic SCC maintenance. When looking at the relationship between the number of iterations/recursions of Zielonka's algorithm and the decrease in decomposition time, it can be clearly seen that the lower the number of iterations, the higher the increase**

in decomposition time when using dynamic SCC maintenance. The same goes for the number of vertices in the game, the lower the number of vertices the more relative increase of the decomposition time when using dynamic SCC maintenance. The conclusions of these last 3 plots can be influenced by the cut off time however. Games with took more than 2 hours to solve are left out from these plots. Especially in the last plot, there could be a set of large games with large trees that just took to much time to build or update, but these can't be seen.

Conclusion. In conclusion, compared to Zielonka's algorithm with Tarjan's algorithm, dynamic SCC maintenance might be beneficial for games which result in very short trees and have many iterations. For all other games the use of partial re-decomposition clearly gives the best option and can save a lot of decomposition time.

7.4 Experiment: runtime differences

The experiment in this section was done to answer the following research questions:

RQ1 What is the impact on the runtime of Zielonka's recursive algorithm of only recalculating SCCs whose vertices have been removed from the graph instead of recalculating all SCCs?

RQ3 What is the impact on the runtime of Zielonka's algorithm of using dynamic SCC maintenance instead of recalculating the SCCs after each iteration?

RQ4 Does the size of the SCC tree matter for the runtime of Zielonka's algorithm with dynamic SCC maintenance? And if so, what is the relation between size and runtime?

First subsection 7.4.1 will describe the details of the experiment. Subsection 7.4.2 shows the results and subsection 7.4.3 discusses these results.

7.4.1 Experiment description

To answer research questions 1, 3 and 4 Zielonka's algorithm was run with the normal SCC decomposition with Tarjan's algorithm, with partial re-decomposition and with dynamic SCC maintenance.

To answer research question 4, the height of the tree of the tree is also measured while building the tree. Next to these the number of trivial and non trivial SCCs is measured. A trivial SCC being an SCC existing just of 1 vertex. If the trees takes more than 2 hours to build the value "na" will be shown instead.

Experiment description: data sets

This experiment was execute on all games of the benchmark. Because there are too many games to discuss in this report, the results of all games in the benchmark that could be tested are uploaded to the digital repository Zenodo (`https://doi.org/10.5281/zenodo.5338737`). Some interesting games, the ones discussed in section 7.1, of these are chosen as summary and will be shown in the results in subsection 7.4.2. Next to the games in the benchmark the set of random games generated by PGSolver was also tested, as well as the set of triangle games.

7.4.2 Results

Tables 7.6 to 7.5 show the of the results of measurements for the runtime differences experiment together with several characteristics of the games. For the random games the average of 25 games will be shown. The standard deviation and median will be included in appendix G When the runtime of Zielonka's algorithm is longer than 2 hours, the measurements belonging to that run will be set to na. The exception to this are the number of trivial and non trivial SCCs, the size of the longest tree and the decomposition time. These come from the run of Zielonka's algorithm with dynamic SCC maintenance. If the dynamic decomposition is completed within 2 hours these results will still be shown even if the game was not solved in 2 hours with dynamic SCC maintentance. If the dynamic decomposition was not completed within 2 hours, its result will be set to 2h+ and the size of the longest tree to na.

7.4.3 Conclusions

The runtime differences experiment shows similar results as the number of vertices decomposed experiment.

First we look at the games whose dynamic decomposition have only short trees, which are the PGSolver and triangle games. In the PGSolver games with only a few iterations/recursions of Zielonka's algorithm all results are similar, but the run with just using Tarjan's algorithm normally scores best. Most likely for these games there are simply not enough iterations to compensate for the extra work done to build the trees and the preparation work of the partial re-decomposition. The triangle games, which do have a lot of iterations/recursions of Zielonka's algorithm, do show clear differences. For these games partial re-decomposition scores best, followed by dynamic SCC maintenance. In the "number of vertices decomposed" experiment, the dynamic SCC maintenance did score better for the triangle games with regards to the time to build and update the dynamic SCC decomposition compared to the time preparation and SCC decomposition of partial re-decomposition. We hypothesize that even though time was saved on that aspect, it might take more time in the dynamic SCC decomposition to find the final SCC or to get the list of vertices in the SCC, which is needed to calculate the attractor set, from a tree since you have to walk through the whole tree.

The games whose dynamic SCC decomposition does have long trees are the random games, ML-Solver games, equivalence games and modelchecking games. When the games next to long trees also have many iterations/recursions of Zielonka's algorithm, partial re-decomposition clearly performs best. This can be best seen in the `Onebitdatasize=3_infinitely_often_read_write` game where Zielonka's algorithm with partial re-decomposition only takes 8 seconds, while Zielonka's algorithm without partial re-decomposition takes 24 seconds, and Zielonka's algorithm with dynamic SCC maintenance takes over 2 hours. So using partial re-decomposition has a big advantage for these games. In the random games, which have only a few iterations/recursions of Zielonka's algorithm, these advantages are no longer clearly visible. Here especially dynamic SCC maintenance scores worse because of the cost of building those long trees. Zielonka's algorithm with or without partial re-decomposition performs about the same, so most likely the preparation overhead is small enough here not to show any disadvantages.

Conclusion. In conclusion, for games with very few iterations, Zielonka's algorithm with Tarjan's algorithm has the shortest runtime. For all other games partial re-decomposition has the shortest runtime.

Pgsolver games								
name	$\|V\|$	#trivial SCC	#non trivial SCC	size longest tree	#tarjan calls	Zielonka+ tarjan runtime	Zielonka+ partial runtime	Zielonka+ dynamic runtime
modelcheckerladder(100)	301	0	1	1	2	**0,001s**	0,002s	0,003s
modelcheckerladder(200)	601	0	1	1	2	0,002s	0,002s	0,005s
modelcheckerladder(500)	1,5K	0	1	1	2	0,003s	0,003s	0,005s
modelcheckerladder(1000)	3K	0	1	1	2	0,003s	0,003s	0,006s
modelcheckerladder(2000)	6K	0	1	1	2	0,007s	**0,005s**	0,013s
modelcheckerladder(5000)	15K	0	1	1	2	**0,039s**	0,041s	0,059s
modelcheckerladder(10000)	30K	0	1	1	2	0,045s	**0,035s**	0,074s
modelcheckerladder(20000)	60K	0	1	1	2	**0,092s**	0,096s	0,197s

Mlsolver games								
name	$\|V\|$	#trivial SCC	#non trivial SCC	size longest tree	#tarjan calls	Zielonka+ tarjan runtime	Zielonka+ partial runtime	Zielonka+ dynamic runtime
Nestern=3_compact	1,6K	409	9	22	34	0,004s	**0,003s**	0,024s
Nestern=4_compact	6,8K	1,6K	12	42	91	**0,015s**	0,017s	0,24s
Nestern=5_compact	29K	6,1K	13	127	211	**0,071s**	0,075s	3,724s
Nestern=6_compact	110K	23K	19	272	557	0,415s	**0,387s**	43,733s
Nestern=7_compact	426K	86K	20	704	1,4K	2,875s	**2,488s**	29m
ParityAndBuechin=3	34K	17K	132	17	110	0,112s	**0,084s**	0,296s
ParityAndBuechin=4	1M	576K	915	95	1,4K	13,323s	**9,712s**	1m58s

Equivalence games								
protocol (data size)	$\|V\|$	#trivial SCC	#non trivial SCC	size longest tree	#tarjan calls	Zielonka+ tarjan runtime	Zielonka+ partial runtime	Zielonka+ dynamic runtime
Buffer SWP (2)	2,3K	74	1	111	13	0,005s	**0,003s**	0,21s
Par SWP (2)	40K	218	74	3K	14	0,325s	**0,242s**	2m17s
Par SWP (4)	108K	866	290	4,3K	14	1,215s	**0,872s**	14m
CABP SWP (4)	788K	1	3,7K	na	28	21,546s	**18,955s**	2h+
Buffer Onebit (3)	2,1M	58K	1	na	32	38,25s	**28,992s**	2h+
Buffer CABP (2)	3,5K	1	1	203	2	0,006s	**0,005s**	0,234s
ABP CABP (2)	58K	1	1	3,9K	2	**0,151s**	0,162s	3m2s
ABP CABP (4)	134K	1	1	2,1K	2	**0,421s**	0,501s	5m31s
SWP SWP (3)	489K	1	1	na	2	**1,707s**	1,82s	2h+

Table 7.4: The results of the runtime experiment for the PGSolver, MLSolver and equivalence games. These games all had capacity 1 and windowsize 1 with branching-bisim equivalence.

Modelchecking games								
name	$\|V\|$	#trivial SCC	#non trivial SCC	size longest tree	#tarjan calls	Zielonka+ tarjan runtime	Zielonka+ partial runtime	Zielonka+ dynamic runtime
Pardatasize=8 infinitely often enabled then infinitely often taken	11K	5,6K	16	3	18	**0,027s**	0,029s	0,039s
ABP(BW)datasize=8 infinitely often enabled then infinitely often taken	12K	5,9K	16	3	18	**0,013s**	0,017s	0,036s
Lift (Correct) nlifts=4 liveness 2 2	45K	14K	52	27	4	0,578s	**0,454s**	0,652s
BRPdatasize=4 nodeadlock	49K	1	1	6	2	**0,049s**	0,063s	0,287s
SWPdatasize=2 windowsize=4 no generation of messages	95K	563	1,4K	3,7K	44	0,313s	**0,228s**	3m31s
Onebitdatasize=2 infinitely often read write	171K	1	1	16K	66	0,19s	**0,189s**	2h+
Onebitdatasize=2 no duplication of messages	258K	12K	10K	16K	70	7,289s	**2,156s**	25m
Onebitdatasize=3 infinitely often read write	608K	1	1	na	66	24,771s	**8,822s**	2h+
Onebitdatasize=3 no duplication of messages	1,2M	51K	39K	na	70	**2,52s**	2,632s	2h+

Triangle games								
#triangles	$\|V\|$	#trivial SCC	#non trivial SCC	size longest tree	#tarjan calls	Zielonka+ tarjan runtime	Zielonka+ partial runtime	Zielonka+ dynamic runtime
50	150	0	50	1	51	0,006s	**0,001s**	0,005s
100	300	0	100	1	101	0,019s	**0,002s**	0,008s
500	1,5K	0	500	1	501	0,132s	**0,01s**	0,037s
1000	3K	0	1K	1	1K	0,455s	**0,026s**	0,071s
5000	15K	0	5K	1	5K	13,811s	**0,609s**	1,338s
10000	30K	0	10K	1	10K	1m0s	**2,596s**	6,834s
30000	90K	0	30K	1	30K	15m	**35,517s**	1m49s
100000	300K	0	100K	1	na	2h+	**15m**	47m

Table 7.5: The results of the runtime experiment for the modelchecking and triangle games.

Random games								
max #*out-edges*	$\|V\|$	avg #trivial SCC	avg #non trivial SCC	avg size longest tree	avg #tarjan calls	avg Zielonka+ tarjan runtime	avg Zielonka+ partial runtime	avg Zielonka+ dynamic runtime
1	1K	971	3	1	2	0,002s	0,002s	0,003s
2	1K	415	2	72	6	0,003s	0,003s	0,030s
5	1K	62	1	274	7	0,006s	0,006s	0,226s
20	1K	0	1	610	6	0,010s	0,010s	0,510s
1	5K	4,9K	4	1	2	**0,006s**	0,007s	0,009s
2	5K	2,1K	2	385	7	**0,014s**	0,016s	0,536s
5	5K	296	1	1,4K	7	**0,041s**	0,043s	5,957s
20	5K	0	1	3,1K	6	**0,100s**	0,103s	18,817s
1	10K	9,9K	4	1	2	**0,013s**	0,017s	0,021s
2	10K	4,2K	1	780	7	**0,037s**	0,039s	2,355s
5	10K	591	1	2,9K	7	**0,146s**	0,149s	38,211s
20	10K	1	1	6,1K	6	0,327s	0,327s	1m56s
1	50K	50K	5	1	2	**0,133s**	0,183s	0,253s
2	50K	21K	1	4K	5	**0,293s**	0,305s	1m46s
5	50K	3K	1	14K	7	**1,399s**	1,407s	33m
20	50K	2	1	31K	6	**2.077s**	2.093s	1h12m

Table 7.6: The average results of the runtime experiment for the random games.

Chapter 8

Threats to validity

In this thesis there are 3 threats to the validity. The first one is that all observations of runtime measurements are based on the best of our capabilities to program them. We tried to keep the part of the program that runs Zielonka's algorithm as similar as possible for all 3 versions, with the exception of the part that decomposes or keeps track of the SCCs. However it is possible that somewhere in the code some computations are done in linear time even though they could have been done in constant time. This might have caused deviations in the observations made.

The second one is that all observations and implementations of dynamic SCC maintenance are based on our interpretation of the paper describing this mechanism. The paper however left out a lot of details which were filled in by us to meet all complexity details, but the author might originally have had a different idea. So it could be that we interpreted some details wrongly which might influence any of the measurements of Zielonka's algorithm with dynamic SCC maintenance and therefore those conclusions as well. However our implementation did meet all complexity requirements.

The third threat is a bug in the code which could cause some games to give the wrong answer. To test the correctness of the program as much as possible, we took a random selection of a set of games of each benchmark. These games were then solved by our program and by PGSolver and the solutions given by both solvers were compared to see if both were the same. The solutions of this set of games were all the same. However not all possible games were tested and checked so it is still possible for there to be a bug which causes a game to be solved incorrectly.

Chapter 9

Conclusions

In this thesis we looked at 2 optimizations for Zielonka's algorithm with SCC decomposition. The first was partial re-decomposition, in which only the part of the graph containing vertices of SCCs which had 1 or more vertices removed is re-decomposed. The second optimization was dynamic SCC maintenance, which builds an SCC tree for each SCC and then maintains those when vertices or edges have to be removed from the graph. Then we executed 3 experiments: the "split node choice" experiment, the "number of vertices decomposed" experiment and the "runtime differences" experiment. The first experiment tested several options of choosing the vertex for the split node, which is an essential point when building the tree. This experiment concluded that first choosing the most connected vertex, and of these vertices the vertex with the highest priority, gave the best results. Then in the "number of vertices decomposed" experiment, the decrease in the number of vertices decomposed during Zielonka's algorithm with partial re-decomposition was tested, along with the time spent decomposing. The time spent decomposing was also compared with the time spent on building and maintaining the SCC trees. The last experiment, the "runtime differences" experiment tested the difference in runtime of Zielonka's algorithm for all 3 versions. The last 2 experiments came to the following conclusions for the research questions:

1. What is the impact on the runtime of Zielonka's recursive algorithm of only recalculating SCCs whose vertices have been removed from the graph instead of recalculating all SCCs?
    - There is a positive impact. The runtime either decreases or does not increase.
2. What is the difference in the total number of vertices decomposed into SCCs when recalculating SCCs whose vertices have been removed from the graph compared of recalculating all SCCs?
    - There is a positive impact especially for runs with many iterations/recursions of Zielonka's algorithm. The number of vertices decomposed into SCCs when only recalculating SCCs whose vertices have been removed from the graph is much less than the number of vertices decomposed into SCCs when recalculating all SCCs. Sometimes this decrease in number of vertices decomposed is more than 99%.
3. What is the impact on the runtime of Zielonka's algorithm of using dynamic SCC maintenance instead of recalculating the SCCs after each iteration?
    - It depends on the game. For games with many iterations/recursions of Zielonka's algorithm and decompositions with very short trees there is a clear positive impact. But for games that do not have these characteristics the impact is negative and the overhead of building the dynamic SCC decomposition is too large.
4. Does the size of the SCC tree matter for the runtime of Zielonka's algorithm with dynamic SCC maintenance? And if so, what is the relation between size and runtime?

- The size of the longest SCC tree definitely matters for the runtime of Zielonka's algorithm with dynamic SCC maintenance. Dynamically keeping track of SCCs is only effective for short SCC trees and not for long ones. An increment in the size of the longest tree gives an increment in the runtime.

5. How many nodes in the SCC tree are updated during the entire run of Zielonka's recursive algorithm in comparison the number of vertices decomposed by partial decomposition or decomposition of the whole graph.

   - The number of nodes in the SCC tree that are updated is usually less than the number of vertices decomposed by partial decomposition or decomposition of the whole graph. Sometimes for short trees of only height 1, no nodes will be updated and such trees will be deleted as a whole from the decomposition.

To conclude from these research questions and the results, which version of Zielonka's algorithm would be the best performing one overall, it would be the version that only recalculates SCCs whose vertices have been removed, so partial re-decomposition. This version performs well in any type of graph. Even if it does not perform the best it performs close to the best. When there are few iterations in Zielonka's algorithm the best choice would be to just recalculate all SCCs, but this number of iterations is not known beforehand, so a solver would not be able to use this knowledge to choose a version.

A future work could be to see if Tarjan's algorithm could be adapted such that what it is running it could already see any non maximal SCCs inside the SCCs in the decomposition. These then might be useful in building the SCC trees and thereby speed up the dynamic SCC decomposition and maybe break this bottleneck.

Next to these it might be interesting to test an implementation of the improved dynamic SCC maintenance algorithm in [1] and compare this to the version used in this thesis that came from [9]. Especially in terms of building the tree, would the GES trees used by the improved version still take as much time to build as the SCC trees? If not then the improved version might be more useful in combination with Zielonka's algorithm.

Bibliography

[1] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. pages 365–376, 2019. 3, 4, 64

[2] E. Allen Emerson. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. DIMACS/AMS, 1996. 1

[3] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. pages 37–54, 2007. 1

[4] Oliver Friedmann and Martin Lange. Solving parity games in practice. In *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2009. 3, 6, 7, 8

[5] Oliver Friedmann and Martin Lange. The pgsolver collection of parity game solvers. Techchnical report, Institut für Informatik, Ludwig-Maximilians-Universität Munchen, Germany 2010. 42

[6] Maciej Gazda and Tim A. C. Willemse. Zielonka's recursive algorithm: dull, weak and solitaire games and tighter bounds. In *GandALF*, volume 119 of *EPTCS*, pages 7–20, 2013. 1, 7, 8

[7] Marcin Jurdzinski. Small progress measures for solving parity games. In *STACS*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000. 3, 8

[8] Jeroen J. A. Keiren. Benchmarks for parity games. In *FSEN*, volume 9392 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2015. 41, 42

[9] Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013. 2, 3, 14, 19, 20, 21, 22, 24, 26, 27, 28, 30, 43, 64

[10] Lisette Sanchez, Wieger Wesselink, and Tim A. C. Willemse. A comparison of bdd-based parity game solvers. In Andrea Orlandini and Martin Zimmermann, editors, *Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26-28th September 2018*, volume 277 of *EPTCS*, pages 103–117, 2018. 14

[11] W.R.M. Schols. TU/e master thesis: Verification of an iterative implementation of tarjan's algorithm for strongly connected components using dafny, November 2020. 8, 68

[12] Antonio Di Stasio, Aniello Murano, Vincenzo Prignano, and Loredana Sorrentino. Solving parity games in Scala. In *FACS*, volume 8997 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2014. 3

[13] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. 5, 7, 8, 67, 68

[14] Tom van Dijk. Attracting tangles to solve parity games. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 198–215. Springer, 2018. 3, 6

[15] Jens Vöge and Marcin Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 202–215. Springer, 2000. 3

[16] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. 2000. 1

[17] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998. 6

Appendix A

Tarjan's Algorithm

One of the simplest and most efficient algorithms to decompose a graph G in SCCs is Tarjan's algorithm [13]. The algorithm which works similarly to a depth first search (DFS) is shown in algorithm 10.

Algorithm 10 Tarjan's algorithm

```
function TARJAN($G = (V, E)$)
    $i \leftarrow 0$
    $S \leftarrow$ empty stack
    for all $v \in V$ do
        $number(v) \leftarrow \infty$
    end for
    while $\exists_{v \in V} : number(v) = \infty$ do
        pick $v \in \{w \in V \mid number(w) = \infty\}$
        STRONGCONNECT($G, S, v, i$)
    end while
end function

function STRONGCONNECT($G = (V, E), S, v, i$)
    $lowlink(v) \leftarrow i$
    $number(v) \leftarrow i$
    $i \leftarrow i + 1$
    $S.push(v)$
    for all $(v, w) \in E$ do
        if $number(w) = \infty$ then
            STRONGCONNECT($G, S, w, i$)
            $lowlink(v) \leftarrow min\{lowlink(v), lowlink(w)\}$
        else if $number(w) < number(v) \wedge S.contains(w)$ then
            $lowlink(v) \leftarrow min\{lowlink(v), lowlink(w)\}$
        end if
    end for
    if $lowlink(v) = number(v)$ then
        while $number(S.peek()) \geq number(v)$ do
            $S.pop()$
        end while
    end if
end function
```

In Tarjan's algorithm the DFS is executed starting from a random vertex, which has not yet been assigned to an SCC, in the graph. Then for each vertex $v \in V$ in the DFS it is first given a *number* and a *lowlink* and stored on the stack of visited vertices in this DFS. Then like in DFS the algorithm checks its outgoing edges. For the target w of each edge $(v, w) \in E$:

- If w has not been visited then the algorithm recurses on w, so in this call $v = w$ instead of a random v. When the recursion call returns, the *lowlink* of v is set to the minimum of the newly acquired *lowlink* of w and the *lowlink* of v. By storing the lowest *lowlink* it stores the earliest vertex in the DFS which can be reached by v.

- If w was already visited and is not on the stack we do nothing. In this case the vertex w was not in this DFS and has already been assigned to a different SCC. If v had been in an SCC with w then during the DFS from w, v would have been explored.

- If w was already visited and is on the stack, then again the *lowlink* is set to the minimum *lowlink* of v and w.

After all targets of edges of v have been visited, v can be in 2 situations:

- The *lowlink* of v still is equal to its *number*. In this case no cycle to a vertex earlier in the DFS has been found so v is the root of an SCC and all other vertices of the SCC have been explored and will have the same *lowlink* value. These are then removed from the stack so that they will not change the *lowlink* values of other not visited vertices that might have an edge to this SCC but are not part of it.

- The *lowlink* of v is lower than its *number*. In this case v is part of an SCC with a root at a vertex earlier in the DFS.

If after exploring the whole DFS there still are vertices that have not been visited we pick another random vertex from these and start over. When the algorithm ends and all vertices have been visited, we can construct the SSCs by looking at the *lowlink* values. Vertices with the same *lowlink* value belong to the same SCC. An example of running Tarjan's algorithm on a graph is shown in Appendix B.

Theorem 7. The recursive version of Tarjan's algorithm, `Tarjan`(G), returns the SCC decomposition of the graph G in $O(|V| + |E|)$ time.

This theorem has been proven by R.E. Tarjan in [13].

One of the possible problems with running Tarjan's algorithm on big graphs in practice is that there might be a lot of recursions of the algorithm stored on the program stack at the same time. To solve this W.R.M. Schols described an iterative version of Tarjan's algorithm in his master thesis [11] and showed its correctness. This iterative version of Tarjan's algorithm is also given in Algorithm 11. Because some graphs of the parity games considered in the experiments later will be very big, the iterative version of Tarjan's algorithm will be used for the SCC decomposition in Zielonka's algorithm.

Theorem 8. The iterative version of Tarjan's algorithm, `TarjanItter`(G), returns the SCC decomposition of the graph G in $O(|V| + |E|)$ time.

In [11] W.R.M. Schols has proven that the iterative version works correctly and just like the recursive version also has a complexity of $O(|V| + |E|)$.

Algorithm 11 Tarjan's algorithm iterative version

```
function TarjanItter($G = (V, E)$)
    $stack, result \leftarrow []$ empty stack
    $low, disc \leftarrow \{:\}$ empty map
    for all $u \in V$ do
        if $u \notin low$ then
            Strongconnect($G, S, u, low, disc, result$)
        end if
    end for
    return $result$
end function

function StrongconnectItter($G = (V, E), S, u_0, low, disc, result$)
    $work \leftarrow [(u_0, 0)]$
    while $|work| > 0$ do
        $(u, j) \leftarrow work.pop()$
        if $j = 0$ then
            $k \leftarrow |disc|$
            $disc[u], low[u] \leftarrow k$
            $stack.push(u)$
        end if
        $recurse \leftarrow false$
        for $i = j$ to $|successors(u)| - 1$ do
            $v \leftarrow successors(u)[i]$
            if $v \notin low$ then
                $work.push((u, i + 1))$
                $work.push((v, 0))$
                $recurse \leftarrow true$
                break
            else if $v \in stack$ then
                $low[u] \leftarrow \min(low[u], disc[v])$
            end if
        end for
        if $\neg recurse$ then
            if $low[u] = disc[u]$ then
                $comp \leftarrow []$
                while true do
                    $v \leftarrow stack.pop()$
                    $comp.push(v)$
                    if $v = u$ then
                        break
                    end if
                end while
                $result.push(comp)$
            end if
            if $|work| > 0$ then
                $v \leftarrow u$
                $(u, j) \leftarrow work.peep()$
                $low[u] \leftarrow \min(low[u], low[v]$
            end if
        end if
    end while
end function
```

Appendix B

Example of running Tarjan's algorithm

As an example of Tarjan's algorithm, the following steps are produced when running it on the graph in the left topmost corner of Figure B.1. When referring to a graph corresponding to one of the steps in the figure, we will number them as (r, c) where r is the row and c is the column.

(1, 1) Shows the graph G.

(1, 2) A random vertex is chosen and given the *number* (in the vertex) and a *lowlink* value 0 and added to the stack.

(1, 3) An outgoing edge is chosen to the right, this vertex is unvisited, so it recurses on it and which is then given the *number* and a *lowlink* value 1 and added to the stack.

(2, 1) An outgoing edge is chosen to the top, this vertex is unvisited, so it recurses on it and which is then given the *number* and a *lowlink* value 2 and added to the stack.

(2, 2) An outgoing edge is chosen to the vertex with *number* 0, this vertex was already visited and is on the stack, so the *lowlink* value of the vertex with *number* 2 is set to the minimum of 0 and 2, which is 0.

(2, 3) The vertex with *number* 2 has no other outgoing edges so the function returns and the vertex with *number* 1 is set to the minimum of 0 and 1, which is 0.

(3, 1) The vertex with *number* 1 has no other outgoing edges so the function returns and the vertex with *number* 0 is set to the minimum of 0 and 0, which is 0.

(3, 2) The vertex with *number* 0 has no other outgoing edges. Its *number* is equal to its *lowlink* value so the vertices with *number*s 0, 1, and 2 are removed from the stack.

(3, 3) There still are unvisited edges in G so a random vertex is chosen from the unvisited edges and given the *number* and a *lowlink* value 3 and added to the stack.

(4, 1) An outgoing edge is chosen to the right, this vertex is unvisited, so it recurses on it and which is then given the *number* and a *lowlink* value 4 and added to the stack.

(4, 2) An outgoing edge is chosen to the bottom, this vertex is unvisited, so it recurses on it and which is then given the *number* and a *lowlink* value 5 and added to the stack.

(4, 3) An outgoing edge is chosen to the right, this vertex has already been visited and is not on the stack so we ignore it. Another outgoing edge is chosen to the top right, this vertex is unvisited, so it recurses on it and which is then given the *number* and a *lowlink* value 6 and added to the stack.

(5, 1) An outgoing edge is chosen to the right, this vertex has already been visited and is not on the stack so we ignore it. A second outgoing edge is chosen to the bottom, this vertex also has already been visited and is not on the stack so we ignore it as well. The last outgoing edge is to the left, this vertex was already visited and is on the stack, so the *lowlink* value of the vertex with *number* 6 is set to the minimum of 4 and 6, which is 4.

(5, 2) The vertex with *number* 6 has no other outgoing edges so the function returns and the vertex with *number* 5 is set to the minimum of 4 and 5, which is 4.

(5, 3) The vertex with *number* 5 has no other outgoing edges so the function returns and the vertex with *number* 4 is set to the minimum of 4 and 4, which is 4.

(6, 1) The vertex with *number* 4 has no other outgoing edges. Its *number* is equal to its *lowlink* value so the vertices with *number*s 4, 5, and 6 are removed from the stack. The vertex with *number* 3 is not removed as its *lowlink* value is smaller than 4.

(6,2) The vertex with *number* 3 has another outgoing edge to the bottom, this vertex is unvisited, so it recurses on it and which is then given the *number* and a *lowlink* value 7 and added to the stack.

(6, 3) An outgoing edge is chosen to the vertex with *number* 3, this vertex was already visited and is on the stack, so the *lowlink* value of the vertex with *number* 7 is set to the minimum of 3 and 7, which is 3.

(7, 1) The vertex with *number* 7 has no other outgoing edges so the function returns and the vertex with *number* 3 is set to the minimum of 3 and 3, which is 3.

(7, 2) The vertex with *number* 3 has no other outgoing edges. Its *number* is equal to its *lowlink* value so the vertices with *number*s 3 and 7 are removed from the stack. All vertices have been visited. As can be seen in this last graph there are 3 SCCs that can be distinguised by their *lowlink* values.

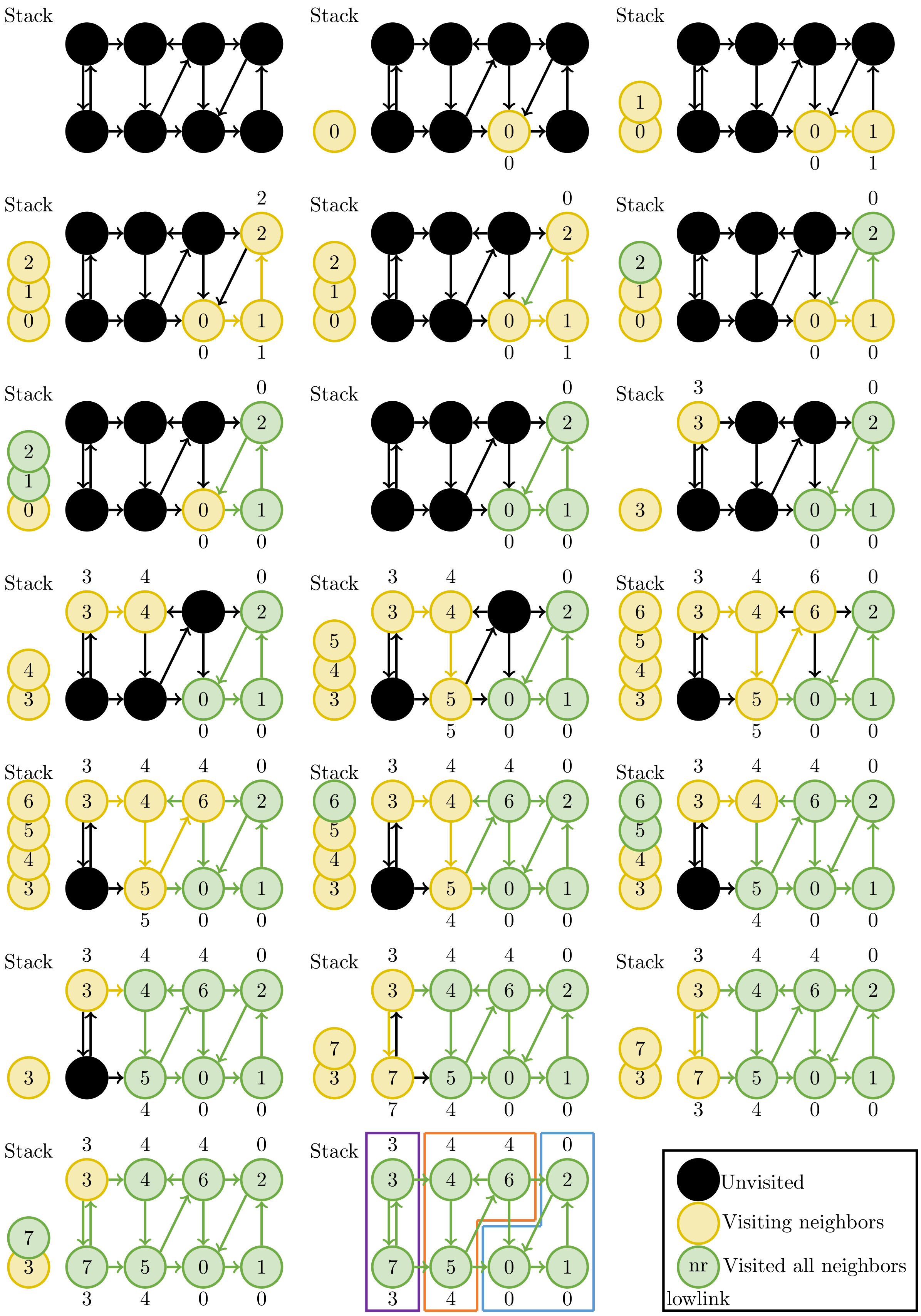


Figure B.1: Steps of tarjans algorithm, read from left to right.

Appendix C

Program

This chapter describes the input and the way the use the program made for the experiments which includes all before described implementations and can be used to solve a game with any of the 3 versions of Zielonka's algorithm. Section C.1 will first describe the format of the input files. There are 2 ways to use the program. The first way is to compile the code and start the program with command line commands. The commands and output will be described in section C.2. The second way is to import the code in an IDE like Intelij and then run the code using Junit tests. This way will be described in section C.3. The code itself can be found at: `https://github.com/Odonja/MasterThesis-final` and requires maven to compile.

C.1 Program input file format

The program only accepts .gm files. These files must have the following format:

- The first line is:

  `parity X;`

  , where X is the number of the highest vertex
- Optionally the second line can have the format:

  `start Y;`

  with Y any number. This line will be ignored by the program.
- Then there is a line for each vertex in the game of the format:

  `index parity owner outedges "optional label";`

  - The `index` must increment by 1 each line, so 0 in the first line, 1 in the second line, 2 in the third line etc.
  - The `parity` can have at most 8 digits, if the parity is larger than 8 digits only the smallest 8 digits will be used.
  - The `owner` can either be 0 or 1
  - The `outedges` are a list of *outedges* of the format 1,2,3,4,5,etc of comma separated target vertices without or without spaces in between.
  - The `"optional label"` is a label that can be present, but will not be used by the program. It must be put between quotes.
- Optionally the last line can be of the format:

  `timeout....`

  where `....` can be replaced with anything. Any line starting with timeout will be ignored.

C.2 Program with command line

To use the program with command line, first navigate to the MasterThesis folder and open the command line. Compile with the following command:

```
mvn package
```

Then the code can be executed with the following command:

```
java -cp ".\zielonka\target\zielonka-1.0-SNAPSHOT.jar;.\common\target\common-1.0-
    SNAPSHOT.jar" org.anhu.Application severalArguments yourFilOrFolderURL
```

In this line `severalArguments` can be replaced with one or more arguments with spaces in between. The following arguments can be used:

- a: print the answer of the game
- i: print all additional information
- t: print the timing of Zielonka's algorithm
- h: print all commands
- F: file specified is a folder, runs all .gm files found in the specified folder
- D: run Zielonka's algorithm with dynamic SCC maintenance
- P: run Zielonka's algorithm with partial re-decomposition
- T: run Zielonka's algorithm with Tarjan's algorithm

The `yourFilOrFolderURL` argument can be replaced by the URL of the game, or the URL of a folder if F is present in `severalArguments`.

C.3 Program with Junit

The second option is to import the code into an IDE and use Junit tests to solve the games. All experiments used the Junit test method. This second method does not have as many options as the command line version. It always runs all 3 versions of Zielonka's algorithm on the game. When solving all games in a folder it will solve them in order of size and cut them off if they take more than roughly 2 hours. When solving just a single game the game will not be cut off after 2 hours.

To run the code in an IDE using Junit:

- Go to `Zielonka\src\test\Application`
- To run a single file enter the file URL in the `file` variable and run the `runFile()` test case.
- To run all .gm files in a folder enter the folder URL in the `folder` variable and run the `runAllFilesInFolder()` test case In this test case. When using this test case there must be a folder named `_results` present in which the results will be stored.

Either test case will print the timing and statistics used in the experiments. Next to these it will write the answer to `_results\Answer_dateTime.txt`, the statistic details are written to `_results\dateTime.txt` and `_results\Excel_dateTime.txt` will contain the same results but in a line that can be copied into excel with variables in the following order: name #vertices #trivial_SCC #non_trivial_SCC size_longest_tree #tarjan_calls #vertices_tarjan #vertices_partial #nodes_updated time_tarjan time_partial_prep time_partial_tarjan time_make_tree time_update_decomposition runtime_tarjan runtime_partial runtime_dynamic

Appendix D

Dynamic SCC Maintenance Implementation

This chapter describes for each of the functions described in chapter 6 the implementation of that function. An overview of the datastructures used in the implementations are given in Figure F.1. Together with the implementation, an analysis of the complexity is given to show that these implementation meet the runtime requirements. The analysis might make use of some operations on the data structures. An overview of the operations on the data structures and their complexity is presented in Table F.1. In chapter F a more detailed description of the datastructures and their operations is given.

D.1 SCC tree implementation

This subsection describes the implementation of the process which builds the dynamic SCC decomposition for a graph, the `Dynamic_SCC_decomposition(`G`)` function described in Algorithm 4. The data structures which are the input for the process of building the SCC tree are:

- the original graph $G = (V, E)$ with $V = \{0, 1, ..., n\}$ stored as as a `Graph` object.
- A boolean array `vertexInGame` of size n which has the value true for each vertex present in the graph.
- The number of vertices currently in the graph, stored in an integer `nrOfVerticesInGame`.

Using this input, the process first needs to find the SCCs of the input graph for which SCC trees need to be build and then call a function to turn these SCCs into an SCC tree. The process of finding these SCCs follows the following steps, where the line numbers correspond to the numbers as shown in algorithm 4:

(2-4) Create a `DynamicSCCDecomposition` to store all trees and information in. Creating an empty object takes $O(1)$ time.

(5) The input graph may exist of multiple SCCs, so first run the iterative version of Tarjan's algorithm on the input graph to find the SCC for which an SCC tree has to be made. The result is stored in `SCCdecomposition`. Running Tarjan's algorithm takes $O(|V| + |E|)$ time.

(6) Then an SCC tree for each SCC `TemporarySCC` in `SCCdecomposition.TemporarySCCList` has to be created. Iterating over the list `SCCdecomposition.TemporarySCCList` gives $O(|TemporarySCCList|)$ iterations.

  (7-8) If `TemporarySCC.members` contains only 1 member then the SCC tree is a single node. Create a `leafNode` object. This step takes $O(1)$ time as its just the creation of an object.

(12) If `TemporarySCC.members` contains more than 1 member then the SCC will be an SCC tree with an `innerNode` object as root. A stack containing a new `innerNode` for the root with as split node the first vertex in `TemporarySCC.members` is created as argument in $O(1)$ time. Then the function which creates an SCC tree for an SCC is called, which takes $O(|E_{SCC}|\delta)$ time.

(10, 16) Add `leafNode` or `innerNode` that was created this to the `DynamicSCCDecomposition`. Adding a node to a `DynamicSCCDecomposition` object takes $O(1)$ time.

(9, 13-15) For each vertex in the SCC store the created `leafNode` or `innerNode` as the root of its SCC tree: for each vertex in `TemporarySCC.members` add the `leafNode` or `innerNode` in the `DynamicSCCDecomposition` as root. Adding a node as root for a vertex in a `DynamicSCCDecomposition` object takes $O(1)$ time.

(18) Store the edges that go from vertices in this SCC to vertices in other SCCs which are present in `TemporarySCC.outgoingEdges` in the `DynamicSCCDecomposition` as edges between trees. Adding an edge between trees to an `DynamicSCCDecomposition` object takes $O(1)$ time. So adding all the edges in the list for all iterations takes $O(|E|)$ time.

So the whole process of creating a dynamic SCC decomposition with SCC trees for a graph takes $O(1+|V|+|E|+|TemporarySCCList|*(|E_{SCC}|\delta+1+1)+|E|) = O(|V|+|E|\delta) = O(|E|\delta)$ since every vertex has at least 1 outgoing edge in a parity game.

Building an SCC tree for an SCC with more than 1 vertex is a recursive process. To build an SCC tree in $O(|E_{SCC}|\delta)$ time each recursion should not take more than $O(|E_{SCC}|)$ time so that each layer of the tree can be created in $O(|E_{SCC}|)$ time. The function that builds the SCC tree for an SCC, which is described in `BuildTree(`G, C`)` function in Algorithm 4, receives the vertices of the SCC in a list `members`, the `DynamicSCCDecomposition` to store the node where edges in the SCC are stored and a stack containing the root inner node. It then follows the following process where the line numbers correspond to the numbers as shown in algorithm 4, steps without line numbers contain details omitted in the algorithm:

(24-25) These steps are omitted since they are received as argument.

(26) Check that there still are inner nodes to be created on the stack if not then return. Since there are at most as many inner nodes as vertices, this step takes $O(|V_{SCC}|)$ time for all recursions together.

(27) Get the current inner node to be created from the stack in $O(1)$ time, or $O(|V_{SCC}|)$ time for all recursions together.

(28-30) A boolean array of size `nrOfVertices` is created, which is initialized to false and set to true for each vertex in `members`. Then a vertex d is chosen in $O(|V_{SCC}|)$ time and set to false in the boolean array. This symbolizes the `split` function, since after the first vertex is split, the created vertices both get their own SCC. Iterating over a list of at most $O(|V_{SCC}|)$ items and setting the value in the boolean array takes $O(|V_{SCC}|)$ time.

(28-30) The graph of the SCC without the first vertex, expressed by the boolean array, is then again decomposed into SCCs using the iterative version of Tarjan's algorithm. The resulting SCCs found can be seen as the condensed vertices for which a node in the tree has to be made. The result is stored in `SCCdecomposition`. Running Tarjan's algorithm takes $O(|V_{SCC}|+|E_{SCC}|)$ time.

1. A hashmap `sccToNode` is created to be able to store for each returned `TemporarySCC` in `SCCdecomposition.TemporarySCCList` which object this SCC was stored in. This map can later be used to map the endpoints of the edges to child nodes they are stored in. Creating a hashmap takes $O(1)$ time.

(31) Then a child node has to be created in the `innerNode` object for each SCC `TemporarySCC` in `SCCdecomposition.TemporarySCCList`:

(34-35) If `TemporarySCC.members` contains only 1 member then the child is a leaf node. Create a `leafNode` object. Checking the size of a list and creating an object take $O(1)$ time.

(36-38, 41-43) If `TemporarySCC.members` contains more than 1 member then the SCC will be an SCC tree with an `innerNode` object as root. Add a new `innerNode` to the stack for this SCC with as split node a vertex in `TemporarySCC.members`. Creating an item and adding it to the stack takes $O(1)$ time, the rest of the time is for the next layer.

(32-33) Add the created `leafNode` or `innerNode` child to the `innerNode` object. Adding a child node to an `innerNode` takes $O(1)$ time.

(a) Store which child the `TemporarySCC` is stored in in the hashmap `sccToNode`. Adding an item to a hashmap takes $O(1)$ time.

2. Get the list of outgoing and incoming edges of the split node from the `Game` object. Getting an object from a hashmap takes $O(1)$ time.
3. Iterate over the list of outgoing edges of the vertex of the split node to add all outgoing edges of the split node to other vertices in the SCC to the `innerNode` object object. To know if the target of the edge is in the SCC the earlier created boolean array can be used to check. Iterating over this list of outgoing edges with a check will give less than $O(|E_{SCC}|)$ iterations.

   (a) Find the node where the target of the edge is stored in $O(1)$ time:
   `targetNode = sccToNode.get(SCCdecomposition.vertexToSCC.get(edge.target)).`

   (b) Add the edge to the `innerNode` object. Adding an edge to a `innerNode` object takes $O(1)$ time.

   (c) Store where the edge is stored. Adding the storage place of an edge to a `DynamicSCCDecomposition` object takes $O(1)$ time.

4. Iterate over the list of incoming edges of the vertex of the split node to add all incoming edges of the split node to other vertices in the SCC to the `innerNode` object object. To know if the target of the edge is in the SCC the earlier created boolean array can be used to check. Iterating over this list of outgoing edges with a check will give less than $O(|E_{SCC}|)$ iterations.

   (a) Find the node where the source of the edge is stored in $O(1)$ time:
   `sourceNode = sccToNode.get(SCCdecomposition.vertexToSCC.get(edge.source)).`

   (b) Add the edge to the `innerNode` object. Adding an edge to a `innerNode` object takes $O(1)$ time.

   (c) Store where the edge is stored. Adding the storage place of an edge to a `DynamicSCCDecomposition` object takes $O(1)$ time.

5. Then the other edges between `TemporarySCCs`, so between the children, have to be stored in the inner node. All edges between children of this inner node are stored in 1 of the lists `TemporarySCC.outgoingEdges`. Iterate over all the edges in these lists will give $O(|E_{SCC}|)$ iterations at most. For each edge do:

   (a) Find the node where the source of the edge is stored in $O(1)$ time:
   `sourceNode = sccToNode.get(SCCdecomposition.vertexToSCC.get(edge.source)).`

   (b) Find the node where the target of the edge is stored in $O(1)$ time:
   `targetNode = sccToNode.get(SCCdecomposition.vertexToSCC.get(edge.target)).`

   (c) Add the edge to the `innerNode` object. Adding an edge to a `innerNode` object takes $O(1)$ time.

   (d) Store where the edge is stored. Adding the storage place of an edge to a `DynamicSCCDecomposition` object takes $O(1)$ time.

6. This innernode is now completed so the process can start over at the start if there are still inner nodes in the stack.

In the step of creating a single layer of the the SCC tree, the step with highest complexity is $O(|V_{SCC}| + |E_{SCC}|) = O(|E_{SCC}|) = O(m)$, meeting the time requirements. So to create an SCC tree with δ layers takes $O(m\delta)$ time.

D.1.1 Find unreachable implementation

This subsection explains how `findUnreachable(`G, S`)` was implemented such that it can find the minimal set of condensed vertices and their incident edges such that if you remove these condensed vertices, G will again be an SCC, in $O(|S| + |V_{SCC}| + |E_{SCC}|)$ time.

The data structures that are the input for the implementation of `findUnreachable(`G, S`)` are:

- A `DAG` object of the condensed graph of the SCC represented by the node.
- A list of `Node` objects, which contain the possible sources and sinks.

First the implementation of `findUnreachable(`G, S`)` will be discussed, followed by the implementation of its 2 helper functions. The line numbers correspond to the numbers as shown in the algorithm.

() Before making the call to `findUnreachableDown` and `findUnreachableUp` a `Unreachables` object is created, which can be passed to those functions as argument so that in line 4 the found nodes and edges don't have to be merged. Creating an object takes $O(1)$ time.

(2-3) These lines make calls to the helper functions `findUnreachableDown` and `findUnreachableUp` to find the nodes that cannot be reached from or cannot reach the split node. Each call as will be shown below takes $O(|S| + |V_{SCC}| + |E_{SCC}|)$ time. So lines 2 and 3 take $O(|S| + |V_{SCC}| + |E_{SCC}|)$ time.

(4) In the pseudocode the results of `findUnreachableDown` and `findUnreachableUp` are merged. However in the implementation the `Unreachables` object was passed as argument to both of them, and the `ListSetPair` made sure that there will not be any doubles present, so line 4 does not need to merge anything and can just return the `Unreachables` object, which takes $O(1)$ time.

So the implementation of `findUnreachable(`G, S`)` takes $O(|S| + |V_{SCC}| + |E_{SCC}|)$ time as long as `findUnreachableDown` and `findUnreachableUp` take $O(|S| + |V_{SCC}| + |E_{SCC}|)$ time.

The implementation of `findUnreachableDown` of Algorithm 5 will be discussed first. Next to the arguments of `findUnreachable(`G, S`)` it also receives an `Unreachables` object to which it will add U and I. Again the line numbers correspond to the numbers as shown in the algorithm.

(8-10) The sets do not have to be created, as the `Unreachables` object from the argument will be used instead. An empty queue is created to store sources which have not yet been processed. An object can be created in $O(1)$ time.

(11-15) These lines check for the list of possible sources given in the argument, if the elements of the list indeed are sources and if so adds them to the queue to be processed. Iterating over the list of possible sources S takes $O(1)$ time per element x. Checking if x is the split node w can be done by checking its class which can be done in $O(1)$ time. The number of *inedges* of x can be received from the `DAG` and then comparing this value to 0. Both these actions take $O(1)$ time as well. Finally if needed x is added the the queue, which can also be done in $O(1)$ time. So all actions in the for loop take $O(1)$ and there are $O(|S|)$ iterations. So lines 11 to 15 take $O(|S|)$ time.

(16) Checking if there are still sources that need processing can be done in $O(1)$ time, and this will be checked at most $|V_{SCC}| + 1$ times.

(17-18) Taking a source node from the queue for processing and adding it to the `Unreachables` object both take $O(1)$ time. Each condensed vertex can be added the the queue at most once, so lines 17 and 18 are executed at most $|V_{SCC}|$ times resulting in complexity $O(|V_{SCC}|)$ for lines 17 and 18 in the whole function.

(19) Adding the incident edges of the source means only adding *outedges*, since sources do not have *inedges*. Since each edge can only be an *outedge* of a single condensed vertex and each condensed vertex is processed at most once, it means that in line 19 at most $|E_{SCC}|$ edges can be added to the set of incident edges. So over the whole function, line 19 will take at most $O(|E_{SCC}|)$ time as adding an edge to an `Unreachables` object takes $O(1)$ time..

(20-25) This loop removes the *outedges* of the source to check if the target vertices of these edges now have become sources. The algorithm does not care which *inedges* the target has, only if it has still *inedges* left to see if it became a source. So to save time in the implementation, edges are not really deleted, but the counter `incomingEdgesCount` in the `DAG` keeps track for each condensed vertex (`Node` object) how many *inedges* a condensed vertex still has. So the loop in lines 20 to 25 is implemented as follows: iterating over the list of *outedges* of the source, which can be retrieved from the `DAG`, decrement `incomingEdgesCount` by 1 for each target and check if the new count has become 0. In this case the target has become a source so add it the the queue. Getting the list of outgoing edges from the `DAG` takes $O(1)$ time, decrementing the number of *inedges* in the `DAG` takes $O(1)$ time, doing a comparison to 0 and adding an object to the queue takes $O(1)$ as well. Iterating over the list of outgoing edges takes $O(1)$ time per edge. As with line 19 each edge will only be processed in 1 iteration of this for loop, so in the whole duration of the function, there will be at most $|E_{SCC}|$ iterations, resulting in a complexity of $O(|E_{SCC}|)$ for this loop for the whole function.

(27) The sets U and I do not have to be returned since they were passed in the argument.

So this results in a complexity of $O(1 + |S| + |V_{SCC}| + |V_{SCC}| + |E_{SCC}| + |E_{SCC}|) = O(|S| + |V_{SCC}| + |E_{SCC}|)$ for `findUnreachableDown`. The steps for `findUnreachableUp` as described in the pseudocode of Algorithm 5 are very similar:

(31-33) The sets do not have to be created, as the `Unreachables` object from the argument will be used instead. An empty queue is created to store sources which have not yet been processed. An object can be created in $O(1)$ time.

(34-38) These lines check for the list of possible sinks given in the argument, if the elements of the list indeed are sinks and if so adds them to the queue to be processed. Iterating over the list of possible sinks S takes $O(1)$ time per element x. Checking if x is the split node w can be done by checking its class which can be done in $O(1)$ time. The number of *outedges* of x can be received from the `DAG` and then comparing this value to 0. Both these actions take $O(1)$ time as well. Finally if needed x is added the the queue, which can also be done in $O(1)$ time. So all actions in the for loop take $O(1)$ and there are $O(|S|)$ iterations. So lines 34 to 38 take $O(|S|)$ time.

(39) Checking if there are still sinks that need processing can be done in $O(1)$ time, and this will be checked at most $|V_{SCC}| + 1$ times.

(40-41) Taking a sink from the queue for processing and adding it to the `Unreachables` object both take $O(1)$ time. Each condensed vertex can be added the the queue at most once, so lines 17 and 18 are executed at most $|V_{SCC}|$ times resulting in complexity $O(|V_{SCC}|)$ for lines 40 and 41 in the whole function.

(42) Adding the incident edges of the sink means only adding *inedges*, since sinks do not have *outedges*. Since each edge can only be an *inedge* of a single condensed vertex and each condensed vertex is processed at most once, it means that in line 42 at most $|E_{SCC}|$ edges can be added to the set of incident edges. So over the whole function, line 42 will take at most $O(|E_{SCC}|)$ time as adding an edge to an `Unreachables` object takes $O(1)$ time..

(43-48) This loop removes the *inedges* of the source to check if the source vertices of these edges now have become sinks. The algorithm does not care which *outedges* the target has, only if it has still *outedges* left to see if it became a sink. So to save time in the implementation, edges are not really deleted, but the counter `outgoingEdgesCount` in the `DAG` keeps track for each condensed vertex (`Node` object) how many *outedges* a condensed vertex still has. So the loop in lines 20 to 25 is implemented as follows: iterating over the list of *inedges* of the source, which can be retrieved from the `DAG`, decrement `outgoingEdgesCount` by 1 for each source and check if the new count has become 0. In this case the source has become a sink so add it the the queue. Getting the list of incoming edges from the `DAG` takes $O(1)$ time, decrementing the number of *outedges* in the `DAG` takes $O(1)$ time, doing a comparison to 0 and adding an object to the queue takes $O(1)$ as well. Iterating over the list of incoming edges takes $O(1)$ time per edge. As with line 19 each edge will only be processed in 1 iteration of this for loop, so in the whole duration of the function, there will be at most $|E_{SCC}|$ iterations, resulting in a complexity of $O(|E_{SCC}|)$ for this loop for the whole function.

(50) The sets U and I do not have to be returned since they were passed in the argument.

So this results in a complexity of $O(1+|S|+|V_{SCC}|+|V_{SCC}|+|E_{SCC}|+|E_{SCC}|) = O(|S|+|V_{SCC}|+|E_{SCC}|)$ for `findUnreachableUp` just like `findUnreachableDown`. And since both of them have this complexity, `findUnreachable` meets the complexity constraint of $O(|S| + |V_{SCC}| + |E_{SCC}|)$ as well.

D.2 Edge deletion implementation

This subsection explains how `deleteEge(`T, e`)` was implemented such that it can delete the edge e and update SCC tree T in $O(mn)$ time.

In the pseudocode of `DeleteEdge` the function receives an SCC tree T, in the implementation however when an edge is deleted from a root node and some of the children become independent SCC trees, the function will need access to the whole decomposition. To facilitate this instead of an SCC tree T the whole dynamic SCC decomposition is passed as an argument as an `DynamicSCCDecomposition` object. This object contains all information needed by `DeleteEdge`, such as node in which the edge is stored or the place to store the newly independent SCC trees. Aside from an `DynamicSCCDecomposition` object the function also receives an `Edge` object containing the edge to be deleted.

After deleting the edge from the corresponding node, the `DeleteEdge` makes a call to the `UpdateSCC-Tree` function to update the tree. The `UpdateSCC-Tree` function receives as arguments:

- The `Node` object N which has to be updated.
- The list of children of N which might have been modified A, for example because of edge deletion or because they have been added to this node in a former iteration.
- The `DynamicSCCDecomposition` object in case some children of the root node become independent SCC trees.

When deleting an edge from the lowest inner node of an SCC tree, it might be the case that all ancestors of this inner node have to be updated. An SCC tree has at most height n, so to meet the $O(mn)$ update time after edge deletion, each call to `UpdateSCC-Tree` should not take more than $O(m)$ time, and the steps of deleting the edge in `DeleteEdge` should not take more than $O(mn)$.

When n is very large there might be very deep trees. To prevent heap becomming full because there are to many recursions, the `UpdateSCC-Tree` function was turned recursive. There is a boolean `recurse` which is true at the start, and will be set to false when normally the function would return instead of recurse. When the function normally would recurse, which is all the way the end when the function would have no more actions, the input node is set to its parent and the list of children that have been modified in the argument is changed to $U \cup \{v_N\}$. This way while `recurse` is true it keeps looping over the whole content of `UpdateSCC-Tree` mimicking the recursions.

First the implementation of the steps of deleting the edge in `DeleteEdge` will be discussed. The line numbers correspond to the numbers as shown in algorithm 6.

(2) First the place where the edge is stored has to be found, this location can be retrieved from the `DynamicSCCDecomposition` object received as argument. First it checks if it is an edge between trees, if so the implementation just deletes it and it is done as no trees have to be adapted. Checking if an edge is an edge between trees an removing an edge between trees both take $O(1)$ time in an `DynamicSCCDecomposition` object. If the edge is not an edge between SCCs, the node N where the edge is stored can be retrieved from the `DynamicSCCDecomposition` in $O(1)$ time. So line 2 takes $O(1)$ time.

(3) If line 3 is reached the edge is an edge in an SCC stored in a node. Removing this node from an `Node` object takes $O(|E_{SCC}|)$ time.

(4) To call `UpdateSCC-Tree` the list of affected children is needed. When removing the edge from `Node` in line 3, the `NodeEdge` is returned. This `NodeEdge` stores which children of N contain the source and target vertices of the edge, so the affected children can be found in $O(1)$ time, and adding them to a list takes $O(1)$ time as well. So all preparations for the call to `UpdateSCC-Tree` take $O(1)$ time.

So the steps of deleting an edge without yet updating the tree take $O(1+|E_{SCC}|+1) = O(|E_{SCC}|)$ time. Next to be discussed are the steps of updating a single inner Node of the tree, so a single run of `UpdateSCC-Tree`.

(8) To call find `unreachable` the `DAG` $D(N)$ has to be created. A `DAG` can be created from an `InnerNode` in $O(|V_{SCC}| + |E_{SCC}|)$ time. Then `findUnreachable` is called which as shown in subsection D.1.1 takes $O(|S| + |V_{SCC}| + |E_{SCC}|)$ time. So in total line 8 can be executed in $O(|V_{SCC}| + |E_{SCC}| + |S| + |V_{SCC}| + |E_{SCC}|) = O(|S| + |V_{SCC}| + |E_{SCC}|)$ time.

(9-11) If all condensed vertices are still reachable, then the node still represents an SCC, no changes have to be made so the function is done. Checking if `unreachableVertices.list` in `Unreachables` is empty can be done in $O(1)$ time.

(12-14) The goal of these steps is to first remove the split node from the list of `unreachableVertices` if it was in there, since it should not be removed from the inner node, and then to remove all nodes of unreachable condensed vertices and incident edges from N. The split node is stored in the inner node, so it can easily be found. Removing an node from a `Unreachables` object takes $O(\text{size of list}) = O(|V_{SCC}|)$ time. Removing the children and their incident edges present in the `Unreachables` object takes $O(|V_{SCC}| + |E_{SCC}|)$ time. So steps 12-14 can be execute4d in $O(1 + |V_{SCC}| + |V_{SCC}| + |E_{SCC}|) = O(|V_{SCC}| + |E_{SCC}|)$ time.

(15-17) In these steps the algorithm checks if N still has any children aside from the split node. If not it will remove the inner node with split node and replace it with a leaf node representing the vertex of the split node. Checking if there are still children aside from the split node can be done by comparing the size of the list of children to 1 in $O(1)$ time. If there are no other children then a leaf node is created with the vertex of the split node, which takes $O(1)$ time. If N is a root node then the leaf node replaces the old inner node in the `DynamicSCCDecomposition` in $O(1)$ time. If N is not a root node, then the leaf node replaces the old inner node in the parent `Node` also in $O(|V_{SCC}| + |E_{SCC}|)$ time. So steps 15-17 will take either $O(1)$ time or $O(|V_{SCC}| + |E_{SCC}|)$ time.

- A step which is needed for the implementation but was not present in the algorithm is to create a hashmap `vertexToChild` which will store for each vertex in `unreachableVertices` as well as the split node in which child of $P(N)$ it is stored. Creating an hashmap takes $O(1)$ time.

(18) Step 18 can be omitted in the implementation as those subtrees, the children, of N were directly used as condensed vertices in `findUnreachable`, so they are already present in `unreachableVertices`.

(19) This step checks if N is the root node, which can be checked by checking if `parent` is equal to 0, in $O(1)$ time.

(20-23) If N is not a root node, the nodes and edges previously removed from N are added to $P(N)$ and their parent is changed. Iterating over the list `unreachableVertices.list` takes $O(1)$ per item, adding them to the parent `Node` and setting their parent takes $O(1)$ per item as well. So handling the vertices takes $O(|V_{SCC}|)$ time. Iterating over the list of `incidentEdges.list` takes $O(1)$ per item and adding them to the parent `Node` also takes $O(1)$, resulting in $O(|E_{SCC}|)$ time for processing the edges. So in total the steps 20-23 can be done in $O(|V_{SCC}| + |E_{SCC}|)$ time.

(24-35) Steps 24-35 come down to correcting the incident edges that were just stored in $P(N)$, as the nodes stored in the `NodeEdge` for these vertices still all point to N. First for each of the vertices in any of the nodes in `unreachableVertices.list` store their node in `vertexToChild`. This can be done in $O(|V_{SCC}|)$ time. Then give this hashmap to the parent `Node` to correct the edges. Correcting the edges for a given hashmap in a `Node` takes $O(|E_{SCC}|)$ time. So steps 24-35 will take $O(|V_{SCC}| + |E_{SCC}|)$ time.

(36-37) In these steps the arguments for the next recursive call or next iteration of `UpdateSCC-Tree`, which updates the parent of N, have to be prepared. The parent node is stored in `parent` in N, U is already present as the list `unreachableVertices.list` and adding $\{v_N\}$ comes down to adding N to the list. So these preparations can all be done in $O(1)$ time and the process can start over for the parent until the ancestor that has no parent.

(39) If N is the root node, then iterating over the list of nodes in `unreachableVertices.list` takes $O(1)$ per item.

(40) For each node remove their parent, making them a root and add them to the dynamic SCC decomposition by adding the node to `sccTrees`, both can be done in $O(1)$ time.

(41) Step 41 finds all vertices stored in the node. Getting a list of all vertices present in a `Node` takes $O(|V_{SCC}|)$ time for all iterations of line 39.

(42-44) Those vertices found in step 41 have to have their root updated since they are now part of a different SCC tree. Through all iterations of line 39 $O(|V_{SCC}|)$ vertices have their root updated in `DynamicSCCDecomposition`, taking $O(|V_{SCC}|)$ time for all of them.

So all these steps together to update a single node of the SCC tree take $O(|S| + |V_{SCC}| + |E_{SCC}| + 1) = O(|E_{SCC}|) = O(m)$ time, meeting the required complexity.

D.3 Deleting a set of vertices from an SCC tree implementation

This subsections how `removeVerticesFromTree(`D, T, X`)` was implemented such that it can delete all edges with an endpoint present in X that are present in SCC tree T from T and update the nodes in T in $O(|X||E|\delta)$ time. All data structures used in this implementation can be found in Figure F.1 and a summary of all operations on the data structures is present in Table F.1.

The in the implementation the `removeVerticesFromTree(`D, T, X`)` function receives as arguments an `DynamicSCCDecomposition` object, a `Node` object which is the root node of SCC tree T and a set of vertices $X \subseteq V$. The implementation of the `topologicalSortAffected(`N, X`)` function in Algorithm 8, receives as arguments a `Node` object which is the root node of SCC tree T and a set of vertices $X \subseteq V$. Lastly the implementation of the `UpdateSCC-TreeNode(`D, N, A, L`)`

function of algorithm 9 receives as arguments an `DynamicSCCDecomposition` object, the `Node` N which has to be updated, a `ListSetPair` A which contains children of N that have been modified and a `ListSetPair` L to store any children of the parent that this function call modifies.

First the implementation of the steps making the topological ordering of T with the `topologicalSortAffected(`T, function will be discussed. The line numbers correspond to the numbers as shown in algorithm 8.

(2-3) These lines create the stack and queue needed to make the topological ordering. The stack and queue hold `Tuple` objects. Creating a stack, queue and a `Tuple` object take $O(1)$ time. Adding an object to the queue takes $O(1)$ time as well.

(4-16) These lines perform the breath-first search on the tree, adding them to the stack as found to create the topological order. These lines check if there are still still `Tuple`s in the queue, if so it dequeues the first and puts it on the stack so it will be popped later from the stack than the `Tuple` its children. It checks for the inner node in the `Tuple` object that was just dequeued all its children. For each child if the child is an leaf node and the vertex of the child is in X then this inner node might have an edge with an endpoint in X, so set `relevant` to true. If the child is an inner node create an `Tuple` object for it, with this `Tuple` object as `parentTuple` and add it to the queue. Checking if there is an item in the queue, dequeueing an item and pushing an item to the stack each take $O(1)$ time. Iterating over the list of children takes $O(1)$ per child. These lines will be executed once for each inner node in the tree. There are at most $O(|V_{SCC}|)$ inner nodes in the tree so during the function lines 17-19 will take in total $O(|V_{SCC}|)$ time. Checking if a node is a leaf node and checking if an item is in a set both take $O(1)$ time, and so does setting a boolean to true. Checking if a node is an inner node, creating an object and adding an item to the queue take $O(1)$ time. So lines 20-26 take $O(1)$ time to execute. Each node is the child of only 1 parent, so for all iterations together the loop starting at line 20 is iterated at most $O(|V_{SCC}|)$ times, thus lines 20-26 take $O(|V_{SCC}|)$ time as well.

(17-32) These lines add the `Tuple` objects to the queue in reverse topological order by popping them from the stack to which they were added in topological order. A node is only added to the queue if it is an ancestor of the leaf node of a vertex in X, so if `relevant` is true. If this node still h as a parent the parent is also an ancestor so `parentTuple.relevant` is set to true as well. Checking if an stack is empty, popping an item from the stack and checking a boolean take $O(1)$ time. Adding an item to the queue, checking if an the parent of a `Node` is null and setting a boolean to true take $O(1)$ as well. The stack will contain a `Tuple` object for each inner node in the tree, so the loop starting at line 28 is executed at most $O(|V_{SCC}|)$ times, making lines 28-36 take $O(|V_{SCC}|)$ time.

(33) The queue is returned. This takes $O(1)$ time.

Taking all these times together makes that the `topologicalSortAffected(`T, X`)` function takes $O(1 + |V_{SCC}| + |V_{SCC}| + 1) = O(|V_{SCC}|)$ time.

The implementation of the steps of deleting the edges and updating the nodes with the `removeVerticesFromTree(`D, T, X`)` function will be discussed next. The line numbers correspond to the numbers as shown in algorithm 7.

(2) A call is made to the `topologicalSortAffected` function passing on the already existing arguments. A call to `topologicalSortAffected` takes $O(|V_{SCC}|)$ time.

(3-4) These lines get the `Tuples` and with these the nodes from the queue one by one. The queue will contain at most $O(|V_{SCC}|)$ tuples so the loop starting at line 3 will have at most $O(|V_{SCC}|)$ iterations. Checking if a queue still has an item and getting an item from the queue $O(|V_{SCC}|)$ times takes $O(|V_{SCC}|)$ time.

(5-18) Thesse lines remove all edges that have endpoints in the set X from the `Node` in the `Tuple`. It passes the `changes` `ListSetPair` in the `Tuple` as argument so that the children that have been modified are all collected in there. Removing all edges with endpoints in given set of vertices from an `InnerNode` takes $O(|E_{SCC}|)$ time.

(19-23) These lines make a call to `UpdateSCC-TreeNode` with the `DynamicSCCDecompsotion` D from the argument, the `Node node` in the `Tuple`, the `ListSetPair changes` in the tuple and the `ListSetPair changes` in the `parentTuple` as arguments. The complexity of `UpdateSCC-TreeNode` is the same as for `UpdateSCC-Tree` as the only thing that has changed is that now instead of adding the nodes present in the list `unreachableVertices.list` and $\{v_N\}$ to a list for the next iteration, now it is added to the `ListSetPair changes` of the `parentTuple` and no iterations will be made. A single iteration of `UpdateSCC-Tree` took $O(|E_{SCC}|)$ time so this call to `UpdateSCC-TreeNode` takes $O(|E_{SCC}|)$ time.

As for each vertex in X only the ancestors of the leaf node representing the vertex are updated, it is the case that lines 5-10 are executed $O(|X|\delta)$ times. So `removeVerticesFromTree(`D, T, X`)` takes in worst case $O(|V_{SCC}| + |V_{SCC}| + |X|\delta|E_{SCC}| + |X|\delta|E_{SCC}|) = O(|X||E_{SCC}|\delta)$ time.

D.4 Deleting an SCC tree from a dynamic SCC decomposition implementation

This subsection will describe the implementation of deleting an SCC tree from a dynamic SCC decomposition, so from a `DynamicSCCDecomposition` object. The function `removeTree(`G, D, T`)` receives a `Game` object G, the `DynamicSCCDecomposition` object D and the root `Node` of the SCC tree T that has to be deleted. Updating D is implemented in the following steps:

- Remove T from the hashmap `sccTees` to delete the tree. Then all vertices of leaf nodes in T have to be removed from the `vertexToRoot` hashmap. To do this first get the list of all vertices in T and then for each vertex in the list remove it from `vertexToRoot`. Removing an item from a hashmap takes $O(1)$ time. Getting the list of all vertices in the SCC tree from a `Node` takes $O(|V_{SCC}|)$ time. Removing each of these vertices from a hashmap also takes $O(|V_{SCC}|)$ time. So removing the tree and updating `vertexToRoot` takes $O(|V_{SCC}|)$ time.

- If T is not a single leaf node tree then it will conta*inedges*. Removing the storage place of these edges follows the following steps: make a stack, put root `Node` T in the queue, then pop an `Node` from the stack and iterate over its edges, for each edge remove it from the `edgesStoragePlace` hashmap, then iterate over its children, if the child is an inner node then push it to the stack, after all children are processed take the next `Node` from the stack if it is not empty. Since a tree is non cyclic each node in the tree is accessed exactly once and so is every edge in the tree. Iterating over the edges takes only $O(1)$ per edge as they are stored in a list. Each edge is has a remove operation on a hashmap which takes $O(1)$ time. There are $O(|E_{SCC}|)$ edges so just removing the edges takes $O(|E_{SCC}|)$ time. Iterating over the children to find all edges takes $O(1)$ per child as they ares stored in a list, and so does adding a value to a stack. There are at most $O(|V_{SCC}|)$ nodes in a tree, so to push and pop all inner nodes of T to the stack takes $O(|V_{SCC}|)$ time. Therefore removing the storage place of the edges in T takes $O(|E_{SCC}| + |V_{SCC}|)$ time.

- To remove the edges between trees for a vertex consists of removing the *inedges* and removing the *outedges* between trees. Removing the *outedges* is simple, just remove the vertex from the `edgesBetweenTrees` hashmap. To remove the *inedges* first get the list of *inedges* from the `Graph` G. Then for each source in the list of *inedges*, if `edgesBetweenTrees` contains the source as key, then remove the vertex from the the set value of this key. If after removing the vertex that set is empty then remove the source from the `edgesBetweenTrees` hashmap as well. Removing a value from a hashmap or set takes $O(1)$ time. Iterating over a list of *inedges* takes $O(1)$ per edge and there are at most $O(|E|)$ *inedges*, so to remove the edges between trees for a vertex takes $O(|E|)$ time.

  To remove the edges between trees for all vertices in T follows the following steps: make a stack, put root `Node` T in the queue, then pop an `Node` from the stack and iterate over its children, if the child is a leaf node then for the vertex of the leaf node remove the edges between trees, if the child is an inner node then push it to the stack, after all children are processed take the next `Node` from the stack if it is not empty. Since a tree is non cyclic each node in the tree is accessed exactly once. Iterating over the children takes $O(1)$ per child as they ares stored in a list Removing the edges between trees is done for each leaf node, thus done $O(|V_{SCC}|)$ times and takes $(|E|)$ time each time. Therefore removing the edges between trees for all vertices in T takes $O(|E||V_{SCC}|)$ time.

So to update all corresponding values when an SCC tree is deleted from a dynamic SCC decomposition still takes $O(|V_{SCC}| + |E_{SCC}| + |V_{SCC}| + |E||V_{SCC}|) = O(|E||V_{SCC}|)$ time.

Appendix E

Implementing 3 versions of Zielonka's Algorithm

In this Appenxix we will give a description of how SCC decomposition was done for each of the 3 versions of Zielonka's algorithm. So basically a description of how line 4 of algorithm 2 was implemented for each version. In the implementation Zielonka's algorithm contained a boolean array is kept, for which each index i is true if vertex $i \in G \backslash (W_0^G \cup W_1^G)$ and false otherwise. This array is updated at the end of each iteration, which would be after line 22 of algorithm 2. Tarjan's algorithm uses this array to know which vertices currently belong to G.

E.1 Zielonka's algorithm with Tarjan's algorithm

The implementation of this version is the simplest. Line 4 of algorithm 2 is literally implemented as a call to the code of Tarjan's algorithm, passing the complete game in a `Graph` object and using the boolean array to provide which vertices it has to decompose.

E.2 Zielonka's algorithm with partial re-decomposition

For this version, when the program reaches line 4 of algorithm 2, a call is made to the function which does the preparation work for partial re-decomposition. This function receives the following:

- A `Graph` object containing the whole game.
- The old SCC decomposition in an `SCCdecomposition` object.
- A list of integers containing the vertices in the attractorset of player 0.
- A list of integers containing the vertices in the attractorset of player 1.

Using this input, the following preparation steps are made to find the vertices that need to be decomposed into SCCs, where the line numbers correspond to the numbers as shown in algorithm 3:

(2) An empty list of integers and a new boolean array, initialised at false, is made to contain the vertices that need to be decomposed. Creating an empty list takes $O(1)$ time.

(2.5) Between step 2 and 3, the `SCCdecomposition` which contains the old SCCs as a list, is put into a set so it is later easy to check if the SCC is still in there or already has been removed. Since each SCC has to be added to the set, where adding to a set takes constant time, and there are at most $|V|$ SCCs, this step takes at most $O(|V|)$ time. Next to these the attractor sets are also combined into a list and a set, which takes $O(|V|)$ time. So step 2.5 takes $O(|V|)$ time in total.

(3) Then each vertex in the combined list of attractor sets is considered. Iterating over the list gives $O(|X|)$ iterations.

  (4) The SCC the vertex is part of can be found in the `vertexToSCC` hashmap in the received `SCCdecomposition` object. Retrieving an object from a hashmap takes $O(1)$ time.

  (5-6) Attempt to remove the SCC of the vertex from the set of old SCCs. If it succeeds the if statement is taken. Removing an object from a set takes $O(1)$ time. So this line takes at most $O(|V|)$ time over all iterations of the loop at line 3.

  (7) If the if statement is taken then iterate over the list of `members` of the SCC, set the new boolean array to true for each member and add it to the list of vertices that need to be decomposed. Also remove the vertex from the hashmap `vertexToSCC` since it will be assigned to a new SCC. Setting a value of a boolean array, adding the the end of a list and removing an object from a hashmap take $O(1)$ time. At most $|V|$ vertices can be considered in total for all iterations in line 7, so this line takes at most $O(|V|)$ time over all iterations of the loop at line 3.

(10) A call is made to the code of Tarjan's algorithm, passing the complete game in the received `Graph` object and using the new boolean array and list of vertices to provide which vertices it has to decompose, as well as the hashmap `vertexToSCC` so the new references can be added and old references will be saved. It receives the new SCCs in a `SCCdecomposition` object. A call to Tarjan's algorithm takes $O(|V| + |E|)$ time.

(11) Iterate over the list of old SCCs, and if the SCC is still in the set of old SCCs then add it to the new `SCCdecomposition` object. There were at most $|V|$ SCCs, so this step takes at most $O(|V|)$ time.

So as mentioned before in chapter 5, partial redecomposition still takes $O(|V| + |E|)$ time.

E.3 Zielonka's algorithm with dynamic SCC maintenance

For this version, before the first call to Zielonka's algorithm starts a call is made to the code that builds the SCC tree, which is described in D.1. This call returns an `DynamicSCCDecomposition` object, which will be passed to each call to Zielonka's algorithm. In lines 12 and 17 of Zielonka's algorithm, algorithm 2, the function recurses. Before this recursion, a new `DynamicSCCDecomposition` object is made containing only a copy of the SCC tree of the SCC of that iteration. This new `DynamicSCCDecomposition` object is then updated to remove the attractor set A or B in a way similar to the process at line 4. At line 4 the call to delete vertices from the dynamic SCC decomposition receives the following arguments:

- A `Graph` object containing the whole game.
- The dynamic SCC decomposition in an `DynamicSCCDecomposition` object.
- A set of integers X containing the vertices that need to be removed.

Using this input, the following steps are made update the dynamic SCC decomposition:

- Convert the received set of integers X to a list. This takes $O(|V|)$ time.
- For each vertex in X ($O(|X|)$ iterations):
  - If the vertex is not in the dynamic SCC decomposition then skip, if it still is, which can be found by checking the `vertexToRoot` hashmap in $O(1)$ time, then:
  - Get the root of the tree the vertex is part of from the `vertexToRoot` hashmap in $O(1)$ time.

- If All vertices in this tree are in $|X|$ then remove the tree as described in Appendix D.4 in $O(|E||V|)$ time for all iterations together.
- If not then remove $|X|$ from the tree with the `removeVerticesFromTree` function whose implementation is described in D.3 in $O(|X||E_|\delta)$ time for all iterations together.
- For each vertex that was both in the tree and in $|X|$ find the new root of the SCC tree they are part of and remove that tree as described in Appendix D.4 in $O(|E||V|)$ time for all iterations together.

So although updating a tree can be done in $O(|X||E_|\delta)$ time, to remove a set of vertices from a dynamic SCC decomposition still takes $O(|E||V|)$ time.

Appendix F

Data structures

This chapter will first describe the data structures used for the implementation of Tarjan's algorithm, building SCC trees and of the `findUnreachable`, `deleteEdge` and `removeVerticesFromTree` functions. Followed by this subsections F.1 to F.6 will describe several operations on the data-structures which will be used in any of the implementations. Figure F.1 shows as an overview a class diagram of all data structures and Table F.1 gives a summary of the complexity of each operation on any of the data structures.

Data Structure (Graph)**.** The `Graph` data structure is the data structure that stores the input graph $G = (V, E)$ with $V = \{0, 1, ..., n\}$ for which an SCC tree has to be build for each SCC. It contains the following:

- A list of outgoing edges for each vertex. This is stored in a Hashmap `outgoingEdges` with as key the vertex and as value the list of targets of the outgoing edges of that vertex. So for each $(v, w) \in E$ we have that the list `outgoingEdges.get(`v`)` contains w.
- A list of incoming edges for each vertex. This is stored in a Hashmap `incomingEdges` with as key the vertex and as value the list of sources of the incoming edges of that vertex. So for each $(v, w) \in E$ we have that the list `incomingEdges.get(`w`)` contains v.
- The total number of vertices in the whole original graph, stored in an integer `nrOfVertices`. So `nrOfVertices`$= n$.

Data Structure (Node)**.** The `Node` objects store the nodes of the SCC tree. There are 3 types of nodes, leaf nodes, inner nodes and split nodes, each get their own object `leafNode`, `innerNode` and `splitNode`. All 3 types store which `Node` is their parent in the SCC tree. Leaf nodes and split nodes also store the vertex they represent. An inner node can have children and store edges, so the `innerNode` object also stores:

- A `splitNode` object.
- A List `children` to store the nodes of its children.
- A List `edges` which stores all edges between its children. For each edge it stores the edge in a `NodeEdge` object containing the endpoints of the edge in the original graph in the integers `source` and `target` as well as for both endpoints which child they are stored in, in `sourceNode` and `targetNode`.

Using this data structure an SCC can be stored in $O(|V| + |E|)$ space. Each vertex in the SCC is stored in at most 1 leaf or split node which contain an integer and a reference to their parent. So to just store the leaf nodes takes $O(|V|)$ space. Each inner node stores at least 1 vertex, namely in the split node. So there can be at most $O(|V|)$ inner nodes. Each inner node stores a reference for the split node. so at most $O(|V|)$ references. It stores a list of children. Each child is only stored

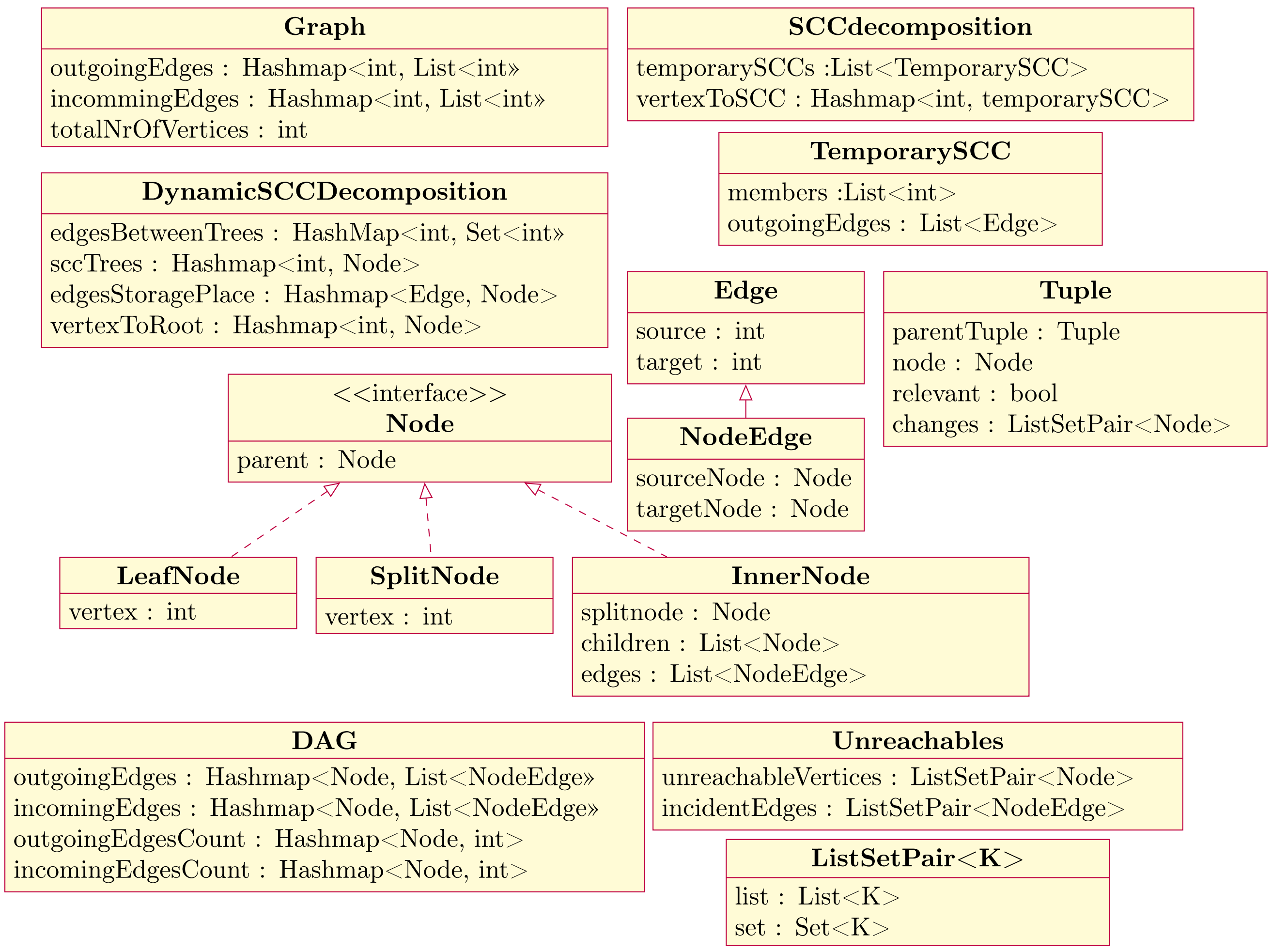

Graph
outgoingEdges : Hashmap<int, List<int»
incommingEdges : Hashmap<int, List<int»
totalNrOfVertices : int
SCCdecomposition
temporarySCCs :List<TemporarySCC>
vertexToSCC : Hashmap<int, temporarySCC>
TemporarySCC
members :List<int>
outgoingEdges : List<Edge>
DynamicSCCDecomposition
edgesBetweenTrees : HashMap<int, Set<int»
sccTrees : Hashmap<int, Node>
edgesStoragePlace : Hashmap<Edge, Node>
vertexToRoot : Hashmap<int, Node>
Edge
source : int
target : int
Tuple
parentTuple : Tuple
node : Node
relevant : bool
changes : ListSetPair<Node>
<<interface>>
Node
parent : Node
NodeEdge
sourceNode : Node
targetNode : Node
LeafNode
vertex : int
SplitNode
vertex : int
InnerNode
splitnode : Node
children : List<Node>
edges : List<NodeEdge>
DAG
outgoingEdges : Hashmap<Node, List<NodeEdge»
incomingEdges : Hashmap<Node, List<NodeEdge»
outgoingEdgesCount : Hashmap<Node, int>
incomingEdgesCount : Hashmap<Node, int>
Unreachables
unreachableVertices : ListSetPair<Node>
incidentEdges : ListSetPair<NodeEdge>
ListSetPair<K>
list : List<K>
set : Set<K>

Figure F.1: An overview of the data structures used to store, create or update the dynamic SCC decomposition of a graph. Each vertex is represented by an integer.

Data structure	**Operation**	**Complexity**
Graph	Get the list of *inedges* or *outedges* of a vertex.	$O(1)$
	Get the total number of vertices of the complete graph.	$O(1)$
DynamicSCC-Decomposition	Add a node to the decomposition.	$O(1)$
	Add an edge between trees.	$O(1)$
	Store, get, remove or change the root of the tree in which a vertex is present.	$O(1)$
	Add, get, remove or change the node in which an edge is stored.	$O(1)$
	Replace an given inner node by a given leaf node and update the reference for the vertex that it is stored in the root leaf node.	$O(1)$
	Check if an edge is an edge between trees.	$O(1)$
	Removing leaf node trees for a given list of vertices.	$O(\|V\| + \|E\|)$
	Removing leaf or inner node tree.	$O(\|V\| + \|E\|)$
	Get a final SCC tree.	$O(\|V\|)$
Node	Get parent or vertex	$O(1)$
InnerNode	Get the split node	$O(1)$
	Add a child node.	$O(1)$
	Add an edge.	$O(1)$
	Get all vertices in the tree starting at this node.	$O(\|V_{SCC}\|)$
	Remove a single edge.	$O(\|E_{SCC}\|)$
	Remove all edges with endpoints in a given set of vertices.	$O(\|E_{SCC}\|)$
	Get the number of children.	$O(1)$
	Remove all nodes and edges present in a given `Unreachables` object.	$O(\|V_{SCC}\| + \|E_{SCC}\|)$
	Replace a given inner node by a given leaf node and replace all occurrences of the inner node in the edges stored in this node by the leaf node.	$O(\|V_{SCC}\| + \|E_{SCC}\|)$
	Correct the edges stored in the inner node for all endpoints stored in a given HashMap<int, Node> such that if any `source` or `target` in the `NodeEdge` is present in the HashMap then afterwards `SourceNode` and `TargetNode` will contain the values of those keys in the HashMap.	$O(\|E_{SCC}\|)$
	Get the DAG of the condensed graph of the SCC represented by the node.	$O(\|V_{SCC}\| + \|E_{SCC}\|)$
	Given a set of vertices that will be removed, check if all vertices of the SCC represented by this inner node will be removed.	$O(\|V_{SCC}\|)$
DAG	Get the list of *inedges* or *outedges* of a node.	$O(1)$
	Decrement or get the number of *inedges* or *outedges* of a node.	$O(1)$
ListSetPair	Add an item if it is not present.	$O(1)$
	Check if an item is present.	$O(1)$
	Iterate over all items.	$O(1)$ per item
	Remove an item.	$O(\|list\|)$
Unreachables	Add a node or edge if it is not present.	$O(1)$
	Check if a node or edge is present.	$O(1)$
	Remove a node or edge	$O(\|list\|)$
	Iterate over the nodes or edges.	$O(1)$ per item

Table F.1: Summary of the complexity of several operations data structures.

in its parent, so all lists of children in inner nodes in a tree together take up at most $O(|V|)$ space. Each edge is stored in only 1 inner node, so all lists of edges in inner nodes in a tree together take up at most $O(|E|)$ space. So to store this all together takes up $O(|V| + |E|)$ space.

Data Structure (DynamicSCCDecomposition)**.** The collection of SCC trees, containing an SCC tree for each SCCs of the input graph are stored in an `DynamicSCCDecomposition` object which stores:

- A hashmap `edgesBetweenTrees` containing the edges between SCC trees such that that the key is the source of the edge, and the value is a set of targets of all edges between SCC trees for that source vertex.
- A hashmap `sccTrees` which stores the SCC trees of the original graph. For an SCC tree that is just a single leaf node it has as key the value of the vertex and the leaf node itself as value. For an SCC tree consisting of multiple nodes, the key is the value of the vertex of the split node of the root node and the value is the root node.
- A hashmap `edgesStoragePlace` which stores for each edge, which is the key, the SCC tree node in which they are stored.
- A hashmap `vertexToRoot` which stores for each vertex, which is the key, the root of the SCC tree which they are part of.

Data Structure (SCCdecomposition)**.** The process of building the SCC tree will need Tarjan's algorithm to decompose graphs into SCCs. The iterative version Tarjan's algorithm as described in Chapter A will be used for this, which takes $O(|V| + |E|)$ time. In a parity game or in a graph of an SCC each vertex has at least 1 outgoing edge, so $|V| \leq |E|$ which means that $O(|V| + |E|)$ $= O(|E|)$ is the complexity for the iterative version of Tarjan's algorithm for the graphs it will be run on. To adapt the iterative version of Tarjan's algorithm for use in the creation of SCC trees, a small changes has been made to the implementation as described in section 3.3, namely that it also finds all outgoing edges of an SCC and stores these in `Edge` objects. The return object `SCCdecomposition` contains the following:

- A list of `TemporarySCC` objects. Each of these objects represents an SCC and contains:
  - A list of integers `members`, which contains all vertices in the SCC the object represents.
  - A list of edges `outgoingEdges`, which contains all edges from vertices in this SCC to vertices in other SCCs. Each `Edge` object contains the source vertex and target vertex.
- An Hashmap `vertexToSCC` which has as key a vertex and as value the `TemporarySCC` object the vertex is part of.

Data Structure (DAG)**.** The input graph (V, E) for the `FindUnreachable` function needs to be able to answer several queries in the function: checking if there are in edges, if there are out edges, to get all in and out edges as well as to remove edges. To meet the running time requirements of `findUnreachable`, any of these queries or actions is only allowed to take constant time. To meet these demands the condensed graph is passed to the function as a `DAG` object which contains the following:

- To get all in edges for a condensed vertex it contains an Hashmap `incomingEdges` with the condensed vertex (`Node` object) as key and a list of edges as value.
- To get all out edges for a condensed vertex it contains an Hashmap `outgoingEdges` with the condensed vertex (`Node` object) as key and a list of edges as value.
- To get the number of in or out edges for a condensed vertex, there are corresponding Hashmaps `incomingEdgesCount` and `outgoingEdgesCount` which contain the condensed vertex (`Node` object) as key and the number of in or out edges as value. This allows the

function to "delete" an in\out edge of a node and see of the other end point of the edge still has out\in edges left without having the time consuming action of having to remove values from a list or to have to iterate over a set.

The condensed vertices are directly passed as `Node` objects in the `DAG` as the child nodes in an inner node of the tree represent each of these condensed vertices and the edges between condensed vertices are also stored as edges between these child nodes.

Data Structure (ListSetPair)**.** While choosing what data structures to use, there were several occasions where neither a set or a list totally fitted. When adding to the data structure, the item should only be added if not present, so this would make a set more suitable. However the functions would often need to iterate over the data structures, which in turn would make a list more suitable. To solve this problem a `ListSetPair` object was created. It contains both a list and a set with the same items.

Data Structure (Unreachables)**.** The `FindUnreachable` function as described in the pseudocode returns (U, I) the set of unreachable vertices and their incident edges, with both U and I a set. In the implementation in the return value which is an `FindUnreachable` object which contains the following:

- A `ListSetPair<Node> unreachableVertices` which contains U, so it contains the nodes that cannot reach or cannot be reached from the split node. A `ListSetPair` was used because the `deleteEdge` function will need to iterate over the list, and be able to check it if nodes are present..
- A `ListSetPair<NodeEdge> incidentEdges` which contains I, so it contains the incident edges of all nodes in `unreachableVertices`. A `ListSetPair` was used because the `deleteEdge` function will need to iterate over the list to add the edges to a parent node, and be able to check it if edges are present to efficiently remove edges in the current inner node.

Data Structure (Tuple)**.** The `Tuple` data structure is a simple data structure that stores the information the `topologicalSortAffected` and `deleteEdgesOfVertices` functions need for a node. A `Tuple` object stores:

- The `Tuple parentTuple`, p from the pseudocode, of the parent of the node $p(N)$. This value is stored so that if this tuple turns out to be an ancestor of the leaf node of a vertex in X then it able to set `relevant` to true in the `Tuple` of its parent, which is an ancestor as well.
- The `Node node`, N from the pseudocode, which is the node this `Tuple` stores information for.
- A boolean `relevant`, r from the pseudocode, which can be set to true when it is discovered that this node is an ancestor of the leaf node of a vertex in X.
- A `ListSetPair changes`, L from the pseudocode, which will be used by `UpdateSCC-TreeNode` as L when it is called on any of its children to store any nodes that it modifies. Next to this it is also used when the edges in the node are deleted to add the endpoints since they are modified and is then passed to `UpdateSCC-TreeNode` as A as the list of vertices that were modified for this node.

F.1 Operations on the Graph data structure

This section will give a description how certain operations are executed on the `Graph` data structure. Next to this a complexity analysis of these operations is given as well.

Operation (Getting the list of in edges of a vertex)**.** For each vertex a list of in edges is stored in the `incomingEdges` hashmap. So getting the list of in edges of a vertex is done by retrieving this list from the `incomingEdges` hashmap which takes $O(1)$ time.

Operation (Getting the list of out edges of a vertex)**.** For each vertex a list of out edges is stored in the `incomingEdges` hashmap. So getting the list of out edges of a vertex is done by retrieving this list from the `outgoingEdges` hashmap which takes $O(1)$ time.

Operation (Getting the total number of vertices of the complete graph)**.** The total number of vertices in the whole original graph is stored in an integer `nrOfVertices`. So the total number of vertices of the complete graph can be accessed in $O(1)$ time.

F.2 Operations on the Node data structure

This section will give a description how certain operations are executed on the `Node` and `InnerNode` data structures. Next to this a complexity analysis of these operations is given as well.

Operation (Get the parent or vertex)**.** The parent is stored in the field `parent` so this value can be accessed in $O(1)$ time. A `SplitNode` or `LeafNode` stores the vertex in the `vertex` field allowing access in $O(1)$ time. An `InnerNode` stores its split node in the `splitNode` field and will return the vertex value of its split node. So getting the vertex of an `InnerNode` takes $O(1)$ time as well.

Operation (Get the split node)**.** The `splitNode` field stores the splitnode. So getting the split node of an `InnerNode` takes only $O(1)$ time.

Operation (Add a child node)**.** A List `children` stores the children. Adding a child to the list can be done in $O(1)$ time.

Operation (Add an edge)**.** The List `edges` stores all edges between the children of the node. Adding an edge to the `innerNode` is adding an edge to this list, which takes $O(1)$ time.

Operation (Get all vertices in the tree starting at this node)**.** It might be needed to know all vertices in a subtree starting at a `Node`, for example when a subtree becomes disconnected from another tree and becomes a tree on itself, all vertices in this subtree need to have an update of the root of the tree they are in. Getting a list of all vertices in a tree starting at `Node` follows the following steps: make a queue, put the `Node` in the queue, then take an `Node` from the queue and iterate over its children, if the child is an inner node then add it to the queue, if the child is a leaf node then add its value to the list, after all children are processed take the next `Node` from the queue if it is not empty. Since a tree is non cyclic each node in the tree is accessed exactly once, and iterating over the children takes only $O(1)$ as it is stored in a list, and so does adding a value to a list. There are at most $O(|V_{SCC}|)$ nodes in a tree, so to get all vertices in a tree takes $O(|V_{SCC}|)$ time.

Operation (Remove a single edge)**.** When deleting an edge, the step of the actual deletion comes down to removing an object from the list of edges in a `Node` object. Removing an item from this list will take $O(|E_{SCC}|)$ time.

Operation (Remove all edges with endpoints in a given set of vertices)**.** To remove all edges with endpoints in a given set of vertices, the list of edges is replaced by an empty list and while iterating over the old list of edges, if the edge does not have an endpoint in the given set of vertices then add it to the list of edges. In this way of removing each edge of the `Node` is processed only once, and for each one the actions of checking if they are in the set of vertices and adding them to the list both only take $O(1)$ time. Therefore to remove all edges with endpoints in a given set of vertices, takes $O(|E_{SCC}|)$ time.

Operation (Get the number of children)**.** A List `children` stores the children. AGetting the number of children can be done by checking the size of the list which can be done in $O(1)$ time.

Operation (Remove all nodes and edges present n a given `Unreachables` object)**.** When there are condensed vertices that have become unreachable because of the edge deletion, the `Nodes` corresponding to these condensed vertices and their incident edges have to be removed from their parent `Node`. To remove all nodes in a `Unreachables` object, the `Node` stores the old list of children and creates an empty list as their list of children. Then it iterates over the old list of children and checks for each child if it is present in the `Unreachables` object. If not then it adds it to the list of children. The incident edges proceed in the same way, the old list of edges is stored, the list of edges is replaced by an empty list and while iterating over the old list of edges, if the edge is not present in the `Unreachables` object then add it to the list of edges. In this way of removing each child and edge of the `Node` is processed only once, and for each one the actions of checking if they are in the `Unreachables` object and adding them to the list both only take $O(1)$ time. Therefore to remove all nodes and edges present in a given `Unreachables` object takes $O(|V_{SCC}| + |E_{SCC}|)$ time.

Operation (Replace a given inner node by a given leaf node and replace all occurrences of the inner node in the edges stored in this node by the leaf node)**.** When deleting an edge from an inner node it can be the case that while updating this node it no longer has children aside from its split node. In this case the inner node with split node has to be replaced by a leaf node in its parent if it was not the root node. To do this first the old inner node has to be deleted from the list of children of its parent which takes $O(|V_SCC|))$ time. Then a new leaf node has to be created and added to the list of children, adding to a list takes $O(1)$ time. Then the edges have to be updated. All references to the old inner node have to be replaced by the new leaf node. This can be done by iterating over the list of edges and checking the `sourceNode` and `targetNode`, if it equals the old inner node, then replace it by the new leaf node. Checking and replacing can both be done in $O(1)$ time and these steps have to be done for all edges, so updating the edges takes $O(|E_{SCC}|)$ time. So to replace an inner node by a given leaf node and to update the edges of the old inner node takes $O(|V_{SCC}| + |E_{SCC}|)$ time.

Operation (Correct the edges stored in the inner node for all endpoints stored in a given HashMap<int, Node> such that if any `source` or `target` in the `NodeEdge` is present in the HashMap then afterwards `SourceNode` and `TargetNode` will contain the values of those keys in the HashMap)**.** After moving the child nodes in the `Unreachables` object to the parent of the node, the edges of the parent have to be updated. There can be edges in the parent whose end points were vertices in these child nodes. Before the corresponding `sourceNode` or `targetNode` of this edge pointed to the node, but now it should point to the child node. To update the edges, the parent node takes a hashmap with the vertex as key and the value the current child of the parent this vertex is in. Only the vertices of children that have been newly added to the parent are present in the hashmap. Then the updating of the edges is done by iterating over the edges of the parent. For each edge check if the endpoints are present in the hashmap, and if so update the corresponding `sourceNode` or `targetNode` with the value in the hashmap. Checking and replacing can both be done in $O(1)$ time and these steps have to be done for all edges, so updating the edges to the values of a hashmap takes $O(|E_{SCC}|)$ time.

Operation (Get the DAG of the condensed graph of the SCC represented by the node)**.** The `findUnreachable` function called after edge deletion needs to receive a `DAG` object as argument. This `DAG` object has to be created from the `InnerNode`. First by iterating over the list of children of the `InnerNode` for each child an empty list is added to the `outgoingEdges` and `incomingEdges` hashmaps, as well as the value 0 to the `outgoingEdgesCount` and `incomingEdgesCount`. These operations take $O(1)$ time per child, so $O(|V_{SCC}|$ time in total. Then adding the edges is done by iterating over the edges of the `InnerNode` and for each `NodeEdge` add it to the lists of `outgoingEdges` and `incomingEdges` for the keys `sourceNode` and `targetNode`. Next to this also increment the `outgoingEdgesCount` and `incomingEdgesCount` for the keys `sourceNode` and `targetNode`. These operations take $O(1)$ time per edge, so $O(|E_{SCC}|$ time in total. Then to get the effect of the split node, which for `findUnreachableDown` should not have in edges and for `findUnreachableUp` should not have out edge, the lists and counts in the hashmaps for the split node are set to an empty list and count equal to 0. These 4 actions take $O(1)$ time each. So to create a `DAG` for an `InnerNode` takes in total $O(|V_{SCC}| + |E_{SCC}|)$ time.

Operation (Given a set of vertices that will be removed, check if all vertices of the SCC represented by this inner node will be removed)**.** text

Operation (Remove all edges with endpoints in a given set X)**.** At some point in the `deleteEdgesOfVertices` the edges that have endpoints in the set x have to be actually deleted from the `InnerNodes`. So the `InnerNode` receives the set X and then to have only edges with no endpoints in X it first stores the old list of edges. The list of edges is replaced by an empty list and while iterating over the old list of edges, if neither of the endpoints of the edge is present in X then add it to the list of edges. In this way of removing each edge of the `InnerNode` is processed only once, and for each one the actions of checking if they are in X and adding them to the list both only take $O(1)$ time. Therefore to remove all edges that have an endpoint in a given set X from an `InnerNode` object takes $O(|E_{SCC}|)$ time. Next to the set X it also receives a `ListSetPair` for the nodes of the endpoings that are removed. Every time an edge does have an endpoint in X its `sourceNode` and `targetNode` are added to the `ListSetPair`. Adding an item to a `ListSetPair` takes $O(1)$ times and at most $O(|E_{SCC}|)$ are added. So to remove all edges with endpoints in a given set X from an `InnerNode` takes $O(|E_{SCC}|)$ time.

F.3 Operations on the DynamicSCCDecomposition data structure

This section will give a description how certain operations are executed on the `DynamicSCCDecomposition` data structure. Next to this a complexity analysis of these operations is given as well.

Operation (Add a node to the decomposition)**.** Adding a root node is as simple as adding an item to the `sccTrees` hashmap, thus taking $O(1)$ time.

Operation (Add an edge between trees)**.** Adding an edge $(v, w) \in E$ comes down to checking if the v already has a key in the `edgesBetweenTrees` hashmap. If not a new set is created for this key, and w is added to the set of V. All of this is done in $O(1)$ time.

Operation (Store, get, remove or change the root of the tree in which a vertex is present)**.** The `vertexToRoot` hashmap stores for each vertex the root of the SCC tree which they are part of. Adding or removing the root of the tree containing a vertex or retrieving the root of the tree in which a vertex is stored can be achieved by either an add, remove or get operation on this hashmap in $O(1)$ time.

Operation (Add, get, remove or change the node in which an edge is stored)**.** The `edgesStoragePlace` hashmap stores for each edge the SCC tree node in which they are stored. Adding the storage place for an edge is adding an item to this hashmap, and retrieving in which node an edge is stored is done by doing get on this hashmap. Both take $O(1)$ time. To remove or change in which node the edge is stored is done by removing or adding to this hashmap as well, taking $O(1)$ time.

Operation (Replace an given inner node by a given leaf node and update the reference for the vertex that is stored in the root leaf node)**.** When deleting an edge from an inner node it can be the case that while updating this node it no longer has children aside from its split node. In this case the inner node with split node has to be replaced by a leaf node in its parent or if it was a root node in the `DynamicSCCDecomposition`. To do this first the old inner node has to be deleted from `sccTrees` and from `vertexToRoot` which takes $O(1)$ time for both. Then a new leaf node has to be created and added to both `sccTrees` and `vertexToRoot`, adding to a hashmap takes $O(1)$ time. So the operation of replacing an inner node by a leaf node in an `DynamicSCCDecomposition` takes $O(1)$ time.

Operation (Check if an edge is an edge between trees)**.** Checking if an edge is an edge between SCC trees can be done by a get on the source in the `edgesBetweenTrees` hashmap and then a contains on the set for the target in $O(1)$ time.

Operation (Removing leaf node trees for a given list of vertices)**.** When deleting multiple vertices but not all from a tree, first all edges in the SCC tree these vertices are in are deleted so those vertices become single leaf node SCC trees. After doing that these single leaf trees and their edges between trees need to be deleted from the `DynamicSCCDecomposition`. The vertices whose leaf node trees have to be removed is received as a list together with the `Game` object that stores the initial graph. First for each vertex in the list remove the single leaf node tree from `sccTrees` and `vertexToRoot`. Removing something from a hashmap takes $O(1)$ time, so all vertices in the list this takes $O(|V|)$ time. Next for each vertex in the list remove the key and value in `edgesBetweenTrees` to remove the outgoing edges between SCCs. Likewise this can be done in $O(|V|)$ time. Then to remove the incoming edges between trees, for each vertex v in the list get the list of incoming edges of v from the `Game`. For each incoming edge of $(w, v) \in E$, check if w is present as key in `edgesBetweenTrees` and if so remove v from the corresponding value. Checking if a key is present, receiving the corresponding value and removing from a set each take $O(1)$ time. This will be done at most $O(|E|)$ times as each edge is only an in edge of 1 vertex, so removing the incoming edges takes $O(|E|)$ time. So the operation of removing the leaf node trees and their edges can be done in $O(|V| + |E|)$ time.

Operation (Removing a leaf or inner node tree)**.** When all vertices of a tree have to be deleted, the tree can be deleted as a whole from `DynamicSCCDecomposition`. First a list of vertices present in the tree has to be generated, which takes $O(1)$ time as later shown in the `Node` data structure. Then for each vertex in the list remove the key and value from `vertexToRoot`. The root node of the tree with its key is removed from `sccTrees` as well. Removing these vertices from the hashmap takes $O(|V|)$ time. Then if the root is not a leaf node the storage place of the edges stored in the tree has to be removed. This is done as follows: make a queue, add the root to the queue, then as long as the queue is not empty, take a node from the queue, iterate over the list of edges and remove them from `edgesStoragePlace`, then iterate over the children, if the child is an inner node add it to the queue, then take the next node from the queue, etc. The operation of removing an item from a hashmap takes $O(1)$ time, each edge is stored in only 1 place so is removed at most once, so removing all edges stored in an SCC tree can be done in $O(|E|)$ time. The last step is to remove the edges between trees for the list of vertices in the tree. This can be done in the same way as in the paragraph above with as list the list of vertices in the tree, in $O(|E|)$ time. So the operation of removing a single SCC tree from the `DynamicSCCDecomposition` takes $O(|V| + |E|)$ time.

Operation (Get a final SCC tree)**.** In Zielonka's algorithm each iteration starts by getting a final SCC. Getting a final SCC tree from a `DynamicSCCDecomposition` takes the following steps: iterate over the values of `sccTrees`, for each tree if the tree is a leaf node and the vertex of this leaf node is not present in `edgesBetweenTrees` then return the leaf node. If the tree is an inner node, then make a queue, add the root of the tree to the queue, then as long as the queue is not empty, take a node from the queue and iterate over the children, if the child is a leaf node and the vertex in that leaf node has outgoing edges in `edgesBetweenTrees` then this tree is not a final SCC tree, so go to the next value in `sccTrees`, if the child is an inner node add it to the queue. When the queue is empty and no leaf nodes with outgoing edges have been found then this tree is a final SCC tree and can be returned. In this process, each leaf node will be looked at at most once, and the only action for the leaf node is to look up a value in the hashmap, so taking $O(|V|)$ time. Iterating over the values of a hashmap takes $O(sizeofhashmap)$ time. There are at most $O(|V|)$ items in the hashmap, with an 0.75 load factor iterating over the hashmap will take at most $O(|V|)$ time. There are less inner nodes in a SCC tree than vertices, so just the action of iterating over the children also takes $O(|V|)$ time. In conclusion getting a final SCC tree from a `DynamicSCCDecomposition` takes $O(|V|)$ time.

F.4 Operations on the ListSetPair data structure

This section will give a description how certain operations are executed on the `ListSetPair` data structure. Next to this a complexity analysis of these operations is given as well.

Operation (Check if an item is present)**.** Checking if an item is present can be done by checking if it is present in the `set`. This takes $O(1)$ time.

Operation (Add an item if it is not present)**.** When adding an item, first the `set` can be checked if the item is already present. If not add the item both to the `set` and `list`. Checking if an item is present in a set, adding an item to a set and adding an item to a list each take $O(1)$ time. So Add an item if it is not present takes $O(1)$ time.

Operation (Iterate over all items)**.** When a function needs to iterate over all items in the data structure, it can just iterate over the `list` in the `ListSetPair`, resulting in $O(1)$ time per item.

Operation (Remove an item)**.** To remove an item from a `ListSetPair` the item has to be removed from both the `set` and `list`. Removing an item from a set takes $O(1)$ time, and removing an item from a list will take $O(|list|)$ time, so to remove an item from a `ListSetPair` takes $O(|list|)$ time.

F.5 Operations on the DAG data structure

This section will give a description how certain operations are executed on the `DAG` data structure. Next to this a complexity analysis of these operations is given as well.

Operation (Getting a list of in edges of a node)**.** For each `Node` a list of in edges is stored in the `incomingEdges` hashmap. So getting the list of in edges of a `Node` is done by retrieving this list from the `incomingEdges` hashmap which takes $O(1)$ time.

Operation (Getting a list of out edges of a node)**.** For each `Node` a list of out edges is stored in the `outgoingEdges` hashmap. So getting the list of out edges of a `Node` is done by retrieving this list from the `outgoingEdges` hashmap which takes $O(1)$ time.

Operation (Getting or decrementing the number of in edges of a node)**.** The `incomingEdgesCount` hashmap contains for each `Node` the number of in edges. Using this hashmap the number of in edges for a `Node` can be decremented or checked in $O(1)$ time.

Operation (Getting or decrementing the number of out edges of a node)**.** The `outgoingEdgesCount` hashmap contains for each `Node` the number of out edges. Using this hashmap the number of out edges for a `Node` can be decremented or checked in $O(1)$ time.

F.6 Operations on the Unreachables data structure

This section will give a description how certain operations are executed on the `unreachables` data structure. Next to this a complexity analysis of these operations is given as well.

Operation (Check if a node is present)**.** Checking if an `Node` is present in an `Unreachables` data structure can be done by checking if it is present in the `UnreachableVertices ListSetPair`. This takes $O(1)$ time.

Operation (Check if an edge is present)**.** Checking if an `NodeEdge` is present in an `Unreachables` data structure can be done by checking if it is present in the `incidentEdges ListSetPair`. This takes $O(1)$ time.

Operation (Add a node if it is not present)**.** Adding a `Node` to an `Unreachables` data structure is done by adding it to the `unreachableVertices ListSetPair`. This takes $O(1)$ time.

Operation (Add an edge if it is not present)**.** Adding a `NodeEdge` to an `Unreachables` data structure is done by adding it to the `incidentEdges ListSetPair`. This takes $O(1)$ time.

Operation (Remove a node)**.** Removing a `Node` to an `Unreachables` data structure is done by removing it from the `unreachableVertices ListSetPair`. This takes $O(1)$ time.

Operation (Remove an edge)**.** Removing a `NodeEdge` to an `Unreachables` data structure is done by removing it from the `incidentEdges ListSetPair`. This takes $O(1)$ time.

Operation (Iterate over the nodes)**.** Iterating over all `Node` objects in an `Unreachables` data structure is done by iterating over the `list` of the `unreachableVertices ListSetPair`. This takes $O(1)$ time per item.

Operation (Iterate over the edges)**.** Iterating over all `NodeEdges` objects in an `Unreachables` data structure is done by iterating over the `list` of the `incidentEdges ListSetPair`. This takes $O(1)$ time per item.

Appendix G

Random games runtime differences experiment standard deviation and median

In section 7.4.2 the average results of each set 25 random games with the same settings was given. This appendix contains the standard deviation and median of each set of 25 random games

Random games								
max *#out-edges*	$\lvert V \rvert$	avg #trivial SCC	avg #non trivial SCC	avg size longest tree	avg #tar-jan calls	avg Zielonka+ tarjan runtime	avg Zielonka+ partial runtime	avg Zielonka+ dynamic runtime
1	1K	14,80686778	1,414213562	0	0	0,001s	0,001s	0,002s
2	1K	30,231	0,917	8,326	2,723	0,002s	0,002s	0,021s
5	1K	9,559	0,2	14,95	2,208	0,001s	0,001s	0,036s
20	1K	0,277	0	13,505	0,866	0,001s	0,001s	0,03s
1	5K	51,465	1,15	0	0	0,001s	0,001s	0,001s
2	5K	62,739	0,764	25,387	2,447	0,005s	0,006s	0,229s
5	5K	22,298	0,332	47,513	1,908	0,009s	0,008s	0,745s
20	5K	0,374	0	25,15	0,841	0,027s	0,026s	4,803s
1	10K	76,079	1,696	0	0	0,002s	0,002s	0,003s
2	10K	69,008	0,557	32,313	2,533	0,013s	0,013s	0,794s
5	10K	26,742	0,277	50,628	1,508	0,048s	0,049s	11,19s
20	10K	0,712	0	48,79	0,833	0,044s	0,044s	6,098s
1	50K	161,825	2,531	0	0	0,013s	0,01s	0,01s
2	50K	206,353	0,436	113,496	2,135	0,198s	0,194s	45,011s
5	50K	60,939	0,332	211,103	2,521	0,495s	0,491s	9m1s
20	50K	1,08	0,2	108,007	0,781	0,53s	0,526s	12m

Table G.1: The standard deviation of the results of the runtime experiment for the random games.

Random games								
max *#out-edges*	$\lvert V \rvert$	avg #trivial SCC	avg #non trivial SCC	avg size longest tree	avg #tar-jan calls	avg Zielonka+ tarjan runtime	avg Zielonka+ partial runtime	avg Zielonka+ dynamic runtime
1	1K	971	3	1	2	0,002s	0,002s	0,002s
2	1K	427	1	72	5	0,002s	0,002s	0,026s
5	1K	65	1	275	7	0,006s	0,006s	0,236s
20	1K	0	1	613	6	0,01s	0,01s	0,512s
1	5K	4,9K	4	1	2	0,006s	0,007s	0,008s
2	5K	2,1K	1	384	7	0,014s	0,015s	0,503s
5	5K	290	1	1,4K	7	0,043s	0,045s	6,133s
20	5K	0	1	3,1K	6	0,088s	0,102s	17,867s
1	10K	9,9K	4	1	2	0,013s	0,016s	0,021s
2	10K	4,2K	1	777	7	0,038s	0,04s	2,11s
5	10K	591	1	2,9K	6	0,127s	0,128s	33,604s
20	10K	0	1	6,1K	6	0,33s	0,335s	1m57s
1	50K	50K	4	1	2	0,131s	0,182s	0,253s
2	50K	21K	1	4K	5	0,263s	0,266s	6m48s
5	50K	3K	1	15K	7	1,365s	1,33s	35m
20	50K	1	1	31K	6	2,276s	2,294s	1h20m

Table G.2: The median of the results of the runtime experiment for the random games.