New algorithms and heuristics for solving Variability Parity Games

Degeling, Koen

Award date:
2021

New algorithms and heuristics for solving Variability Parity Games

Author
Koen Degeling

Supervisor
dr. ir. T. A. C. Willemse

Technische Universiteit Eindhoven
November 8, 2021

Abstract

Variability parity games are a recently proposed extension to well-known parity games that allow for verification of software product lines (SPLs). We propose new algorithms for solving variability parity games based on the existing priority promotion and SCC decomposition, and provide new heuristics for VPGs. We implemented these proposed algorithms, as well as an algorithm based on Jurdziński's small progress measures We compare existing algorithms for solving VPGs and the impact of different pre-processing steps and propose a method to generate random variability parity games.

Contents

Chapter 1

Introduction

Formal model verification techniques are used to verify whether a model adheres to a set of formal requirements. This verification is used to improve the quality of the modelled system, often software. Verifying the requirements can help understand and prove the correctness of the system, something which is getting more important as a lot of critical software is getting significantly more complex.

In verifying software, labelled transition systems are often used to model the *behaviour* of a system. Using temporal logics, such as LTL, CTL and μ-calculus, we can formally describe the requirements of the system. Using labelled transition systems, we can check whether a requirement holds for our model. The problem of finding whether such a requirement holds for a given model has been shown to be reducible (in polynomial time) to solving a parity game, and vice versa.

The topic of parity games is well-established in research [2, 19, 18]. Multiple algorithms exist – and are still being proposed – for solving parity games. It has been shown that parity games are in the complexity class of UP and co-UP and it is still an open question whether they can be solved in polynomial time [18]. Recent algorithms have been shown to solve parity games in quasi-polynomial time[3, 13].

When developing software, we often have multiple similar products we want to verify. Software product lines (SPLs) refer to methods and techniques to describe and reason about multiple software products that are very similar and share most of their behaviour and features. *Featured transition systems* [5] have been proposed to describe these SPLs and are a generalisation of labelled transitions systems. Verifying whether a set of requirements hold for multiple products can be done by solving *variability parity games*, a generalisation of a parity game, as shown in [1].

Variability parity games (VPGs) have been first described in [20] and can be used to verify products in a SPL with a "family based" approach, in which we verify multiple products simultaneously, instead of each individual product in the "product based" approach. Using VPGs we can verify multiple products whilst exploiting the commonalities between them. A few algorithms already exist and have been proposed for solving VPGs and it has been shown to be more efficient than solving the different parity games separately.

In this thesis we will propose new algorithms for solving VPGs by extending priority promotion [2], an existing algorithm for parity games to the setting of VPGs and we

show its correctness. Furthermore, we propose the concept of *SCC-families*, which describe the strongly connected components in a VPG, and implement an SCC decomposition algorithm with tight integration into Zielonka's algorithm, as proposed in [1]. Furthermore, we consider a recently proposed algorithm based on Jurdzinski's small progress measures [12] and provide its implementation details.

Pre-processing parity games is a method to adapt or partially solve parts of a parity game with the intention to reduce its complexity. We will propose some adaptations of the pre-processing steps from [14] and measure their effectiveness when solving VPGs. The algorithms and pre-processing steps we propose have been implemented using C++.

Lastly we compare the existing and new algorithms and pre-processing steps against each other on the existing dataset of VPGs, as well as the newly generated VPGs, to compare the solving times of the different algorithms. We found that the recursive algorithm has the best performance overall. We also found that all three algorithms that depend on the attractor-set calculation (Priority Promotion, Zielonka, Zielonka with SCC decomposition) spend most of their solving time on computing the attractor-set.

The thesis is structured as follows. In Chapter 2 we expand on the related work and the context of this thesis. Next, in Chapter 3 we introduce the preliminary concept of variability parity games. In Chapter 4 we introduce SCC-families and apply them to Zielonka's algorithm with SCC decomposition. Next, in Chapter 5 we briefly introduce the concepts of small progress measures and provide implementation details for the VPG adaptation. We introduce Priority Promotion and show how to adapt its concepts to apply to VPGs and provide an algorithm for Priority Promotion on VPGs in Chapter 6 and in Chapter 7 we adapt self-loop elimination for VPGs. Lastly, in Chapter 8 we discuss our experimental results.

Chapter 2

Related work

In the previous section we already mentioned some previous work on variability parity games and algorithms to solve them, and work related to parity games. We will now discuss some of the most relevant contributions related to the topic.

Multiple algorithms for solving parity games exist, for instance the priority promotion algorithm, as proposed in [2] in 2016, to solve parity games with a time complexity of $O(|E| \cdot (3\frac{|V|-2}{d-2})^{d-1})$ where $|E|$ is the number of edges, $|V|$ is the number of vertices and d is the number of priorities in the parity game. In 1998 Zielonka's algorithm [23] was proposed to solve parity games with $O(|V|^d)$ worst case time complexity, and in 2000 Jurdziński small progress measures algorithm [19] has been showed to solve parity games in roughly $O(|E| \cdot (|V|/d)^d)$. Although Zielonka's algorithm has the worst complexity of the aforementioned algorithms, it often outperforms the other algorithms in practice. Recent algorithms have been shown to solve parity games in quasi-polynomial time [17, 4].

A Software Product Line (SPL) contains multiple products with requirements we want to verify for each one. SNIP [6] is one of the first implementations of an SPL model checker, based on the model checker SPIN [16], and allows for checking LTL formulas for multiple products. ProVeLines [7] contains multiple tools for verifying SPLs and is an extension of SNIP. It allows for verifying discrete and real-timed models, checking the equivalence between SPLs and checking different types of features.

Besides the SPL specification used in this thesis, different specifications and verification techniques have been put forward, such as *variability abstractions* [11]. Where instead of featured transition systems and μ-calculus so-called *feature models* are used, which described the set of valid configurations.

Chapter 3

Preliminaries

We now introduce variability parity games, attractor sets and Zielonka's algorithm.

A variability parity game (VPG) is a generalisation of a parity game, which is played by players *even* and *odd*, denoted by 0 and 1 respectively. The game is played on a finite directed graph, with the vertices of the graph labelled with a *priority*, which is a natural number, and an *owner*, which is either the even or odd player. An edge in a VPG has a set of configurations associated with it for which it is enabled, which we call its *edge guard*. Each VPG has a set of configurations and is played/solved for one of them.

Definition 3.0.1. (Variability Parity Game). A variability parity game is a tuple $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ where,

- V is a finite set of vertices,
- $E \subseteq V \times V$ is the total edge relation,
- $\Omega : V \to \mathbb{N}$ is the priority function, assigning a priority to each vertex,
- $\mathcal{P} : V \to \{0, 1\}$ is the owner function that assigns an owner to each vertex,
- $\mathfrak{C}$ is a non-empty set of configurations,
- The guard function $\theta : E \to 2^{\mathfrak{C}} \setminus \{\emptyset\}$ is a total function mapping every edge to a set of configurations for which that edge is enabled. Furthermore, we require that for all $v \in V \bigcup_{(v,w)\in E} \theta(v, w) = \mathfrak{C}$.

We can partition V into vertices owned by player 0 and player 1. Let $\alpha \in \{0, 1\}$ be an arbitrary player, and $\bar{\alpha}$ their opponent. We define the set of vertices owned by player α as $V_\alpha = \{v \in V \mid \mathcal{P}(v) = \alpha\}$. We denote the *successors* and *predecessors* of a vertex v as vE and Ev respectively, where $vE = \{w \in V \mid (v, w) \in E\}$ and $Ev = \{w \in V \mid (w, v) \in E\}$.

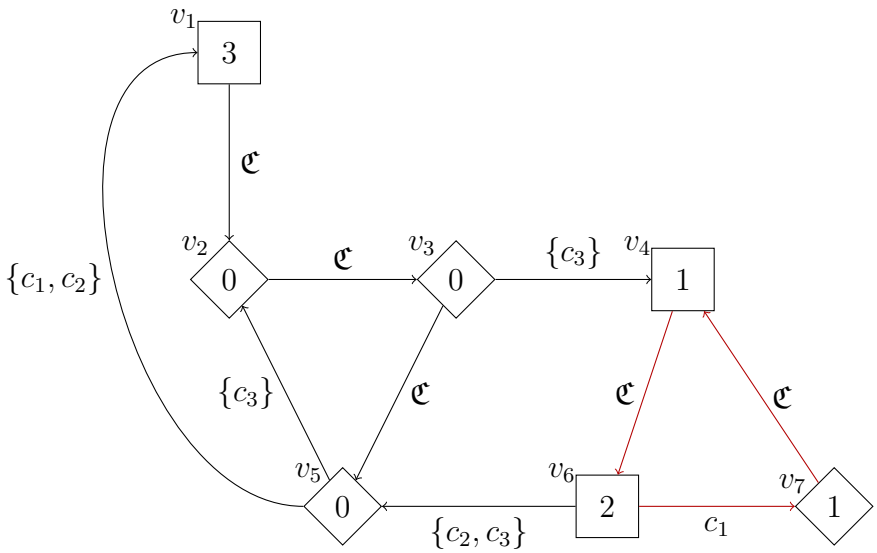


Figure 3.1: A VPG, with in red the infinite play $\pi = (v_4, v_6, v_7)^\omega$ with $\text{pr}^{\downarrow}(\pi) = 1$.

When displaying a VPG we use the following conventions, as can be seen in Figure 3.1:

- A vertex owned by the odd player is denoted by a box, such as vertex v_1,
- A vertex owned by the even player is denoted by a diamond, such as vertex v_2,
- The priority of a vertex is denoted by the number within the box or diamond,
- Direct edges are denoted using arrows between vertices,
- Edges are decorated with a set of configurations, which is the set $\theta(e)$ for an edge $e \in E$, for example for the edge from v_6 to v_5 in Figure 3.1 we have that $\theta((v_6, v_5)) = \{c_2, c_3\}$.

A *play* of a VPG starts by placing a token on a vertex $v \in V$ in the VPG for a configuration $c \in \mathfrak{C}$. The token can then be moved by the owner of the vertex along one of the outgoing edges, with the requirement that c is enabled for the outgoing edge e, i.e. $c \in \theta(e)$. The owner of the new vertex can then move the token along again according to the same rules. Since we require that the graph is total, and the guard function to have at least one edge (v, w) such that $c \in \theta(v, w)$ for all $v \in V$ and $c \in \mathfrak{C}$, we will always be able to move the token, resulting in an infinite path π. With π_i we denote the ith vertex along the path, with $i \in \mathbb{N}$, and the *prefix* of a path we denote with $\pi^{<i} = \pi_0 \pi_1 \dots \pi_{i-1}$, or $\pi^{\leq i}$, which includes the last vertex π_i.

The winner of a play is determined by the *lowest* priority that occurs infinitely often along its resulting path π, where the priority of a vertex π_i is given by $\Omega(\pi_i)$ for $i \in \mathbb{N}$. We denote the lowest priority occuring infinitely often in π with $\text{pr}^{\downarrow}(\pi)$. The play is

won by player *even* if $\text{pr}^{\downarrow}(\pi)$ is even, and won by player *odd* if $\text{pr}^{\downarrow}(\pi)$ is odd. With $\text{pr}^{\uparrow}(\pi)$ we denote the *highest* parity occuring infinitely often along π. Similarly, we define $\text{pr}^{\downarrow}(\mathcal{G})$ and $\text{pr}^{\uparrow}(\mathcal{G})$ as the lowest and highest priority occuring in the game $\mathcal{G}$. In Figure 3.1 we can see an example of a play with starting vertex v_4 and configuration c_1 consisting of $\pi = (v_4, v_6, v_7)^{\omega}$, where ω denotes infinite repetition.

Note that in our definition the lowest parity decides the winner of a play, this is also called a *min*-game. We could have also defined a *max*-game, where the highest priority decides the winner of a play. Both types of games can easily be turned into its alternative, but for consistency we will be considering *min*-games.

A player's moves are determined by their strategy. As the enabled edges depend on the configuration $c \in \mathfrak{C}$, we define the strategy for each $c \in \mathfrak{C}$. Let α be an arbitrary player and $c \in \mathfrak{C}$ be a configuration, we define the strategy for player α as a partial function $\sigma_\alpha^{c\,*} : V^* \to V$ where the function is defined for paths ending with a vertex which is owned by player α. We say a path π and configuration c *conform* to a given strategy $\sigma_\alpha^{c\,*}$ iff for all prefixes $\pi^{\leq i}$, for which $\sigma_\alpha^{c\,*}$ is defined, we have $\pi_{i+1} = \sigma_\alpha^{c\,*}(\pi^{\leq i})$ for all $i \in \mathbb{N}$. A strategy $\sigma_\alpha^{c\,*}$ is winning from a vertex $v \in V$ iff α is the winner of every play starting in v that conforms to $\sigma_\alpha^{c\,*}$. We say a vertex v is won by player α and configuration c if player α has a winning strategy from vertex v.

VPGs are, like parity games, positionally determined. This means that if a vertex is won by player α, player α has a strategy that does not depend on the history of the vertices that the play has previously visited, i.e. we have a partial function $\sigma_\alpha^c : V \to V$ that determines the next move, which we call the *memoryless* strategy. Furthermore, we note that a vertex is either won by the even or the odd player for a given configuration.

We can now partition the vertex-configuration pairs of VPG $\mathcal{G}$ into two sets W_0^c and W_1^c such that vertex v is in W_α^c if and only if player α has a winning strategy σ_α^c from vertex v for configuration c. Solving a VPG is done by computing these partitions for all $c \in \mathfrak{C}$.

Given a VPG $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ and a configuration $c \in \mathfrak{C}$, by $\mathcal{G}|c$ we denote the projection of $\mathcal{G}$ onto c. This projection is a parity game $\mathcal{G}|c = (V, E', \Omega, \mathcal{P})$ where $E' = \{e \in E \mid c \in \theta(e)\}$, which can be seen as a specific class of VPG that only has one configuration. Furthermore, we can also define a projection for a set of configurations, $\mathcal{C} \subseteq \mathfrak{C}$, where the projection $\mathcal{G}|\mathcal{C}$ is a parity game with $E' = \{e \in E \mid \exists c \in \mathcal{C} : c \in \theta(e)\}$.

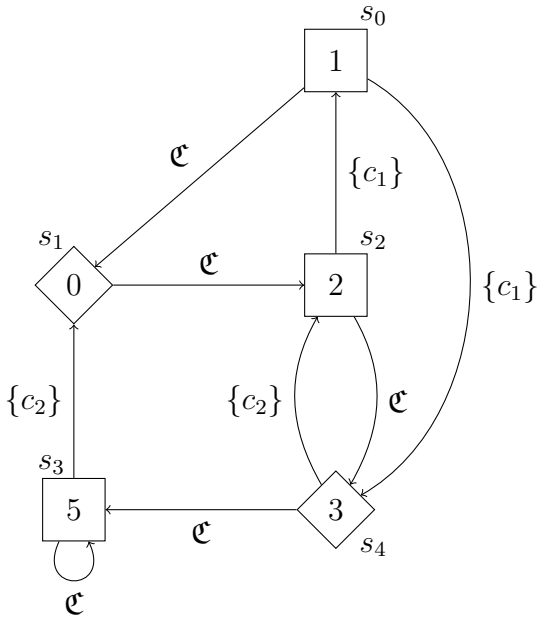
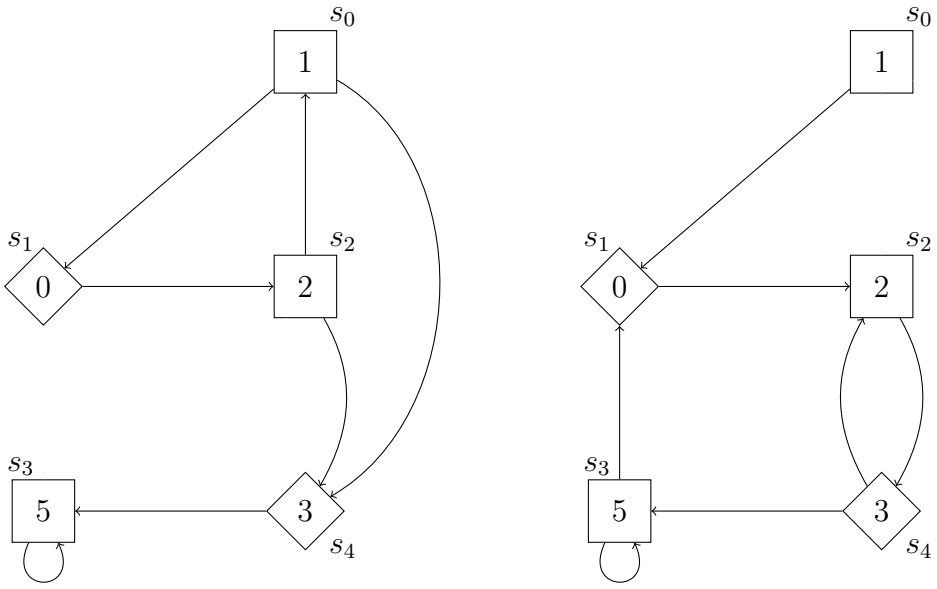


Figure 3.2: Example of a VPG with $\mathfrak{C} = \{c_1, c_2\}$.

Example 3.0.1. Consider the VPG $\mathcal{G}$ from Figure 3.2. In Figure 3.3 we can see the projections $\mathcal{G}|c_1$ and $\mathcal{G}|c_2$. For the projection on c_1 we have the winning sets $W_o^{c_1} = \emptyset$ and $W_1^{c_1} = \{s_0, s_1, s_2, s_3, s_4\}$, since player 0 does not have a winning strategy from any vertex in Figure 3.3a. For the projection on c_2 in Figure 3.3b we have the winning sets $W_0^{c_2} = \{s_0, s_1, s_2, s_4\}$ and $W_1^{c_2} = \{s_3\}$, since player 1 now only has a winning strategy for vertex s_3.



(a) Resulting parity game of projection $\{c_1\}$. (b) Resulting parity game of projection $\{c_2\}$.

Figure 3.3: Projections of the VPG from Figure 3.2.

When discussing a *subgame* we call a function $\varrho : V \to 2^{\mathfrak{C}}$ a *restriction* of $\mathcal{G}$, which indicates the configurations which are under consideration for a vertex. Let $\psi, \varrho : V \to 2^{\mathfrak{C}}$ be two restrictions, we define their intersection, using lambda calculus, as $(\psi \cap \varrho) = \lambda v \in V.\psi(v) \cap \varrho(v)$, their subtraction $(\psi \setminus \varrho) = \lambda v \in V.\psi(v) \setminus \varrho(v)$ and their union $(\psi \cup \varrho) = \lambda v \in V.\psi(v) \cup \varrho(v)$. We say ϱ is a sub-mapping of ψ ($\varrho \subseteq \psi$) iff for all $v \in V$ we have $\varrho(v) \subseteq \psi(v)$, similarly, we can have a *strict* sub-mapping $\varrho \subset \psi$. We say that a vertex v for configuration $c \in \mathfrak{C}$ is won by player α in the game $\mathcal{G}$ restricted to ϱ iff $c \in \varrho(v)$ and the winning strategy for α only passes through vertices w such that $c \in \varrho(w)$. With $\texttt{vert}(\varrho)$ we denote the *vertices* included in the restriction ϱ, i.e. the set $\{v \in V \mid \varrho(v) \neq \emptyset\}$. With $\text{pr}^{\downarrow}(\mathcal{G}, \varrho)$ we denote the lowest priority in the game $\mathcal{G}$ restricted to ϱ, i.e. $\min\{\Omega(v) \mid v \in \texttt{vert}(\varrho)\}$.

3.1 Attractor

The attractor set is an important concept that is often used when solving parity games. It is the set of vertices from which a player can force a play into a set of vertices $U \subseteq V$. In the case of variability parity games, instead of a set of vertices, the attractor is a restriction, giving for each vertex in the restriction the set of configurations for which the vertex is part of the attractor. Formally we define this attractor, which we will call the *featured attractor*, as proposed in [1], denoted α-FAttr, as follows:

Definition 3.1.1. (Featured Attractor). Given a VPG $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \theta)$, and a sub-mapping U^λ of a restriction ϱ, we define $\alpha\text{-FAttr}(\varrho, U^\lambda)(v) = \bigcup_{i \geq 0} \alpha\text{-FAttr}_i(U^\lambda)(v)$ where $\alpha\text{-FAttr}_i(U^\lambda)(v)$ is inductively defined as follows:

$$
\begin{aligned}
&\alpha\text{-FAttr}_0(U^\lambda)(v) = U^\lambda(v) \\
&\alpha\text{-FAttr}_{i+1}(U^\lambda)(v) = \alpha\text{-FAttr}_i(U^\lambda)(v) \cup \\
&\qquad \{c \in \varrho(v) \mid v \in V_\alpha \wedge \exists w \in vE : c \in \theta(v,w) \cap \varrho(w) \cap \alpha\text{-FAttr}_i(U^\lambda)(w)\} \cup \\
&\qquad\quad \{c \in \varrho(v) \mid v \in V_{\bar{\alpha}} \wedge \forall w \in vE : c \in (\mathfrak{C} \setminus (\theta(v,w) \cap \varrho(w))) \cup \alpha\text{-FAttr}(U^\lambda)(w)\}
\end{aligned}
$$

We say a restriction ϱ is α-maximal in $\mathcal{G}$ with $\alpha \in \{0, 1\}$ iff $\varrho = \alpha\texttt{-FAttr}(\varrho, U^\lambda)$. Next, we say that $\mathcal{G}$ is total with respect to a restriction ϱ iff for all $v \in V$ and all $c \in \varrho(v)$, there exists a vertex $w \in vE$ such that $c \in \theta(v,w) \cap \varrho(w)$. Lastly, we observe that for a VPG $\mathcal{G}$ that is total with respect to ϱ, and an α-maximal restriction $U^\lambda \subseteq \varrho$ the game $\mathcal{G}$ is total with respect to $\varrho \setminus U^\lambda$. We have the following two Lemma's from [1].

Lemma 1. Let $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ be a VPG, $\varrho : V \to 2^{\mathfrak{C}}$ a restriction and α an arbitrary player. Then for all sub-mappings U^λ of ϱ, $\alpha\texttt{-FAttr}(\varrho, U^\lambda)$ is also a sub-mapping of ϱ. □

Lemma 2. Let $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ be a VPG and $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ. Let U^λ be an α-maximal restriction such that U^λ is a sub-mapping of ϱ. Then $\mathcal{G}$ is total with respect to $\varrho \setminus U^\lambda$. □

3.2 Zielonka's algorithm

In the previous section we introduced the featured attractor for VPGs, as proposed in [1]. In this section we will describe how we can solve VPGs using this featured attractor in Zielonka's recursive algorithm. We will describe the concepts behind the algorithm, give a rough description of the algorithm and then describe the full algorithm as proposed in [1].

We can recursively solve VPGs using the featured attractor. Let $\mathcal{G}$ be a VPG and $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ. We first take the set of vertices $v \in V$ and their configurations $c \in \varrho(v)$ that have the minimum priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$ in our game $\mathcal{G}$ and compute their attractor, for the player with the same parity as the minimum priority: $\alpha \equiv_2 \mathrm{pr}^{\downarrow}(\mathcal{G})$. This gives us an α-maximal sub-mapping A^λ of ϱ of all the vertices and configurations for which player α can force a play into a vertex with minimum priority in the game. The game $\mathcal{G}$ restricted to $\varrho \setminus A^\lambda$, which we will call the subgame, can then be solved recursively. We now have two cases:

1. The subgame is entirely won by player α. Trivially the entire game is also won by player α.
2. Part of the subgame is won by player $\bar{\alpha}$. Our attracted set A^λ is not necessarily won by player α, as the opponent might be able to force a play from a vertex $v \in \mathtt{vert}(A^\lambda)$ for a configuration $c \in A^\lambda(v)$ into a part of the subgame that is won by player $\bar{\alpha}$. Therefore, we remove the set of vertices from which the opponent can force the play into one of its winning vertices, and again solve the resulting subgame. Since we are guaranteed that the winning vertices of the opponent are won by player $\bar{\alpha}$, we are ensured the returned solution is correct.

The full procedure for solving VPGs can be seen in Algorithm 1. For convenience, we use the following shorthand notation to represent the empty restriction $\emptyset^\lambda = \lambda v \in V.\emptyset$.

In the next chapter we will introduce strongly connected components for VPGs and show how we can integrate them into Zielonka's recursive algorithm to solve VPGs.

Algorithm 1: Zielonka's recursive algorithm for VPGs.

```
1  Zielonka (𝒢 = (V, E, Ω, 𝒫, θ), ϱ)
2      (W_0^λ, W_1^λ) ← (∅^λ, ∅^λ)
3      if ϱ ≠ ∅^λ then
4          p ← pr^↓(𝒢, ϱ)
5          α ← p  mod 2
6          U^λ ← λv ∈ V.{c | c ∈ ϱ(v) ∧ Ω(v) = p}
7          A^λ ← α-FAttr(ϱ, U^λ)
8          (W_0^λ', W_1^λ') ← Zielonka(𝒢, ϱ \ A^λ)
9          if W_ᾱ^λ = ∅^λ then
10             (W_α^λ, W_ᾱ^λ) ← (A^λ ∪ W_α^λ', ∅^λ)
11         else
12             B^λ ← ᾱ-FAttr(ϱ, W_ᾱ^λ')
13             (W_0^λ', W_1^λ') ← Zielonka(𝒢, ϱ \ B^λ)
14             (W_α^λ, W_ᾱ^λ) ← (W_α^λ', B^λ ∪ W_ᾱ^λ')
15         end
16     end
17 end
18 return (W_0^λ, W_1^λ)
```

Chapter 4

Solving VPGs by decomposition into Strongly Connected Components

Parity games can be solved by solving the strongly connected components (SCCs) as shown in [14]. In Section 4.1 we will briefly introduce SCC decomposition in normal parity games and informally sketch how an SCC decomposition can be used when solving parity games. Next we will introduce strongly connected components in a "family-based" setting and describe how we can use SCC decomposition to solve VPGs. Lastly, we introduce a recursive algorithm that uses SCC decomposition to solve VPGs, adapted from the original algorithm for parity games from [15].

4.1 SCC Decomposition

First we will define the concept of strongly connected components on a total directed graph $G = (V, E)$.

Definition 4.1.1. (Strongly Connected Components). Let $G = (V, E)$ be a total directed graph. A set of vertices $S \subseteq V$ is called *strongly connected* if for all $v, w \in S$ there exists a path from v to w in G. The *strongly connected components* of a graph G are all the *maximal* sets of vertices of G that are strongly connected.

Let $\mathcal{S} = S_0, \dots S_n$ be the set of strongly connected components in G. We have the topological ordering $\rightarrow$ where we say that $S_i \rightarrow S_j$ for $i, j \in \mathbb{N}$ iff there is a vertex $w \in S_j$ and $v \in S_i$ such that $v \rightarrow w$ and $S_i \neq S_j$. The smallest elements in this ordering are called *terminal* SCCs. Let S_i for some $i \in \mathbb{N}$ be a terminal SCC, by definition there does not exist a component S_k with $k \in \ltimes$ and $i \neq k$ such that $S_i \rightarrow S_k$. Therefore, we cannot leave such a terminal component, which will be an important property when solving VPGs, since any play that enters the terminal SCC is guaranteed to remain within the vertices of the SCC. We can solve the VPG for the vertices of the terminal SCC without having to take into account the entire game. An example of a SCC decomposition can be seen in Figure 4.1, where we have terminal SCC S_2.

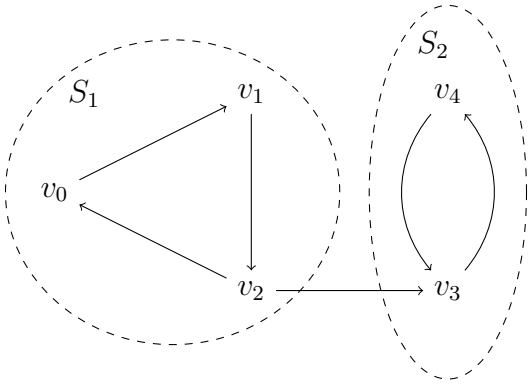


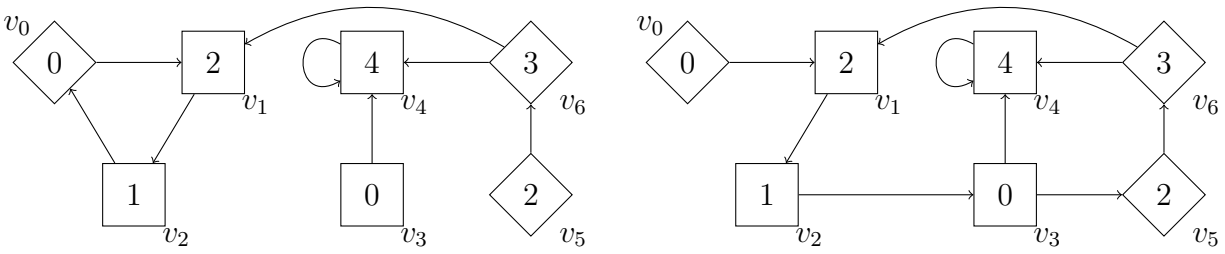
Figure 4.1: SCC decomposition of a graph consisting of two strongly connected components S_1 and S_2.

4.2 Family-based Strongly Connected Components

Before we define the strongly connected components of a VPG, first we will consider the SCCs of a parity game $\hat{\mathcal{G}}$. Let $\hat{\mathcal{G}} = (V, E, \Omega, \mathcal{P})$ be a parity game. We can compute its SCCs in the graph $G = (V, E)$.

Definition 4.2.1. (SCC-Equivalence). Let $\hat{\mathcal{G}} = (V, E, \Omega, \mathcal{P})$ be a parity game and $\equiv \subseteq V \times V$ a relation, where $v \equiv v'$ for $v, v' \in V$ iff there is an SCC S of (V, E) such that $v, v' \in S$.

Lemma 3. The relation $\equiv$ from Definition 4.2.1 is an equivalence relation on V.



(a) Projection of $\mathcal{G}|c_1$

(b) Projection of $\mathcal{G}|c_2$

Figure 4.2: Projections of the VPG $\mathcal{G}$ from Figure 4.3.

Example 4.2.1. Consider the parity games $\mathcal{G}|c_1$ and $\mathcal{G}|c_2$ from Figure 4.2. For Figure 4.2a we have the equivalence classes: $S_0 = \{v_0, v_1, v_2\}, S_1 = \{v_3\}, S_2 = \{v_4\}, S_3 = \{v_5\}, S_4 = \{v_6\}$. And for Figure 4.2b we have the equivalence classes: $S_0 = \{v_1, v_2, v_3, v_5, v_6\}, S_1 = \{v_0\}, S_2 = \{v_4\}$.

We can now define the strongly connected components of a VPG $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$. Let $\approx_{\mathcal{C}} \subseteq V \times V$ be a relation with $\mathcal{C} \in 2^{\mathfrak{C}}$. We define the *SCC-family* as follows:

Definition 4.2.2. (SCC-Family). Given a VPG $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ and $\mathcal{C} \in 2^{\mathfrak{C}}$. Let $\approx_{\mathcal{C}} \subseteq V \times V$ be defined such that:

$v \approx_{\mathcal{C}} v'$ iff for all $c \in \mathcal{C}$ we have $v \equiv v'$ in $\mathcal{G}|c$.

Lemma 4. The relation $\approx_{\mathcal{C}}$ from Definition 4.2.2 is an equivalence relation on V.

Proof. In order to prove that $\approx_{\mathcal{C}}$ is an equivalence relation, we will show that it is reflexive, symmetric and transitive.

Reflexivity By reflexivity of $\equiv$ we have that $v \equiv v$ in $\mathcal{G}|c$ for all $c \in \mathfrak{C}$. It follows that $v \approx_{\mathcal{C}} v$.
Symmetry Assume $v \approx_{\mathcal{C}} w$. By definition of $\approx_{\mathcal{C}}$ we have $v \equiv w$ in $\mathcal{G}|c$ for all $c \in \mathcal{C}$. By symmetry of $\equiv$ we have that $w \equiv v$ in $\mathcal{G}|c$ for all c. We conclude that $w \approx_{\mathcal{C}} v$.
Transitivity Assume $v \approx_{\mathcal{C}} w$ and $w \approx_{\mathcal{C}} u$. We have $v \equiv w$ and $w \equiv u$ in $\mathcal{G}|c$ for all $c \in \mathcal{C}$. By transitivity of $\equiv$ we have that $v \equiv u$ in $\mathcal{G}|c$ for all $c \in \mathcal{C}$. Hence $v \approx_{\mathcal{C}} u$.
□

Let $[v]_{\approx_{\mathcal{C}}}$ be the equivalence class of $v \in V$ with $\mathcal{C} \in 2^{\mathfrak{C}}$, where $[v]_{\approx_{\mathcal{C}}} = \{w \in V \mid w \approx_{\mathcal{C}} v\}$.

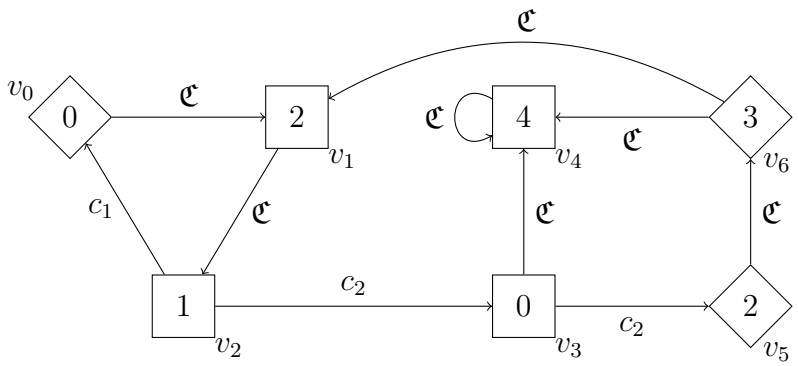


Figure 4.3: Strongly Connected Components in a VPG $\mathcal{G}$ with $\mathfrak{C} = \{c_1, c_2\}$.

Example 4.2.2. Consider the VPG $\mathcal{G}$ from Figure 4.3. We have the following set of equivalence classes for $\{c_1\}$ and $\{c_2\}$: $s_0 = \{v_0, v_1, v_2\}_{c_1}, s_1 = \{v_3\}_{c_1}, s_2\{v_5\}_{c_1}$ and $s_3 = \{v_6\}_{c_1}$ and $s_4 = \{v_1, v_2, v_3, v_5, v_6\}_{c_2}$ and $s_5 = \{v_0\}_{c_2}$ and for $\{c_1, c_2\}$: $s_6 = \{v_4\}_{\{c_1,c_2\}}$, which are also the strongly connected components of the individual projections of $\mathcal{G}$.

4.3 Approximated decomposition

In the worst case this SCC-family decomposition yields a different SCC for each configuration and vertex, meaning we can no longer exploit the commonality in the VPG.

The time complexity for such a decomposition (using Tarjan's algorithm [22]) would be $\mathcal{O}(|\mathfrak{C}| \times (|V| + |E|))$. In Figure 4.4 we can see a VPG consisting of 4 vertices and 3 configurations, where all strongly connected components for the different configurations are distinct. In fact, for any VPG with n configurations and $n + 1$ vertices, where $n \geq 2$, we can construct such a worst case scenario where for all configurations all the connected components are distinct. Hence, it might be more interesting to consider an *approximation* of this family-based SCC decomposition.

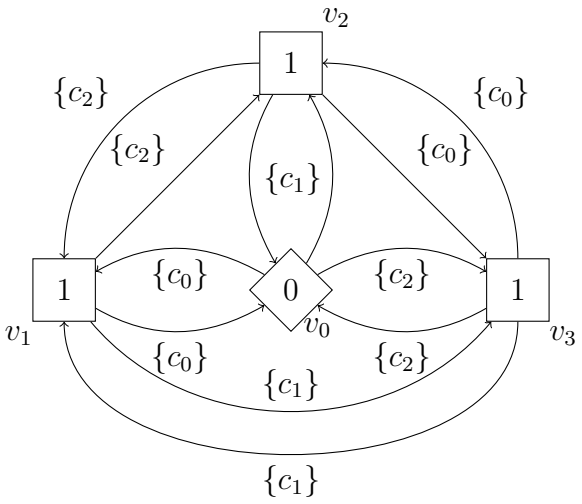


Figure 4.4: Example of a VPG where all strongly connected components in the SCC-family are distinct.

Example 4.3.1. Given the VPG from Figure 4.4 we have the following sets of strongly connected components for the different configurations: $\{v_0, v_1\}, \{v_2, v_3\}$ for configuration c_0, in case of c_1 we have $\{v_0, v_2\}, \{v_1, v_3\}$ and for c_2: $\{v_0, v_3\}, \{v_1, v_2\}$.

Instead of computing this SCC-family exactly, we propose the concept of an *approximation* of an SCC-family.

Definition 4.3.1. (SCC-Family Approximation). Given a VPG $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$, $\mathcal{C} \in \mathfrak{C}$ and $\tilde{\mathcal{C}} \in \mathfrak{C}$ such that $\mathcal{C} \subset \tilde{\mathcal{C}}$. We say that $\tilde{\mathcal{C}}$ (over)approximates $\mathcal{C}$ iff for all $v, v' \in V$ we have that $v \equiv v'$ in $\mathcal{G}|\tilde{\mathcal{C}}$.

Instead of computing the SCC decomposition exactly, we only compute the approximated SCC decomposition for some $\tilde{\mathcal{C}} \in 2^{\mathfrak{C}}$.

Note that the coursest SCC approximation we can compute is that of the game $\mathcal{G}|\mathfrak{C}$. This approximation disregards any of the edge guards and computes the connected components in the entire VPG. Although it is not exploiting any information about the different configurations, it is computationally inexpensive and easy to compute (using for instance Tarjan's algorithm [22] running in $\mathcal{O}(|V| + |E|)$).

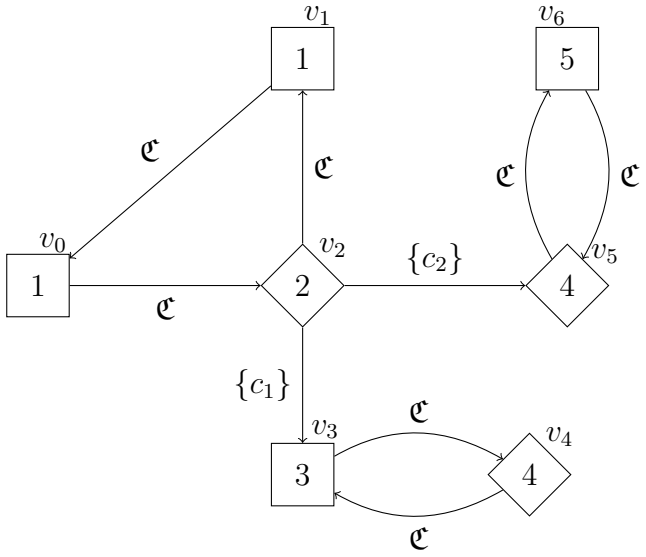


Figure 4.5: VPG with $\mathfrak{C} = \{c_1, c_2\}$ and strongly connected components $\{v_0, v_1, v_2\}$, $\{v_3, v_4\}$ and $\{v_5, v_6\}$.

Using this coursest approximation, we can decompose a VPG into its strongly connected components. Algorithm 2 on the following page describes such an algorithm for solving VPGs with tight SCC integration. Note that the algorithm is very similar to the recursive algorithm we described in Chapter 3.2. On line 8 we compute the SCC decomposition of the VPG, where we only include vertices if $\varrho(v) \neq \emptyset$ and includes edges if $\varrho(v) \cap \theta(v, w) \cap \varrho(w) \neq \emptyset$. Next we compute the restriction $\zeta = \lambda v \in V.\{c \mid c \in \varrho(v) \wedge v \in T\}$, where $\zeta(v) = \emptyset$ if $v \notin T$ and $\zeta(v) = \varrho(v)$ otherwise. Because of this, the restriction $\zeta(v)$ is closed under reachability: for each vertex $v \in \texttt{vert}(\zeta)$ there does not exist a vertex $w \in V \wedge w \notin \texttt{vert}(\zeta)$ such that $(v, w) \in E$ and $\varrho(v) \cap \theta(v, w) \cap \theta(w) = \emptyset$. Because of being closed under reachability, if player α has a winning strategy for vertex v and a configuration $c \in \mathfrak{C}$ in the game $\mathcal{G}$ restricted to ζ, the player will also have this strategy in the entire game $\mathcal{G}$.

Next we solve the game $\mathcal{G}$ restricted to ζ similarly as we would in Zielonka's algorithm: removing all vertices with minimum priority and recursively solving the smaller game. Lastly, any vertices and configurations which were not in any of the terminal components are solved on line 26.

Example 4.3.2. Let $\mathcal{G}$ be the VPG as shown in Figure 4.5. The game $\mathcal{G}$ has three strongly connected components: $\{v_0, v_1, v_2\}$ and the terminal components $\{v_3, v_4\}$ and $\{v_5, v_6\}$. The vertices in $\{v_3, v_4\}$ are won by for configurations $\mathfrak{C}$ the odd player (by the lowest priority 3) and by extension of the featured attractor the vertices $\{v_0, v_1, v_2\}$ are also won by the odd player for configurations $\{c_1\}$. Similarly, the vertices $\{v_5, v_6\}$ are won by the even player for configurations $\mathfrak{C}$ (by lowest priority 4) and by extension of the featured attractor the vertices $\{v_0, v_1, v_2\}$ are also won by the even player for configurations $\{c_2\}$. As all vertices and configurations have been solved, the algorithm can stop.

In Example 4.3.2 on the preceding page we informally give an example how Algorithm 2 solves a VPG using SCC decomposition. In the example the recursive algorithm with SCC integration requires two recursive calls (both on line 15) and one SCC decomposition to solve the game. If we compare this to Zielonka's recursive algorithm from page 10 – which requires 14 recursions in total – we conclude that for certain VPGs it can be beneficial to do this SCC decomposition, instead of the original Zielonka's algorithm.

Algorithm 2: Zielonka's recursive algorithm with SCC decomposition integration.

```
1  Zielonka-SCC(𝒢 = (V, E, Ω, 𝒫, θ), ϱ)
2      (W_0^λ, W_1^λ) ← (∅^λ, ∅^λ)
3      if ϱ = ∅^λ then
4          return (W_0^λ, W_1^λ)
5      else
6          V' ← {v ∈ V | ϱ(v) ≠ ∅}
7          E' ← {(v, w) ∈ E | ϱ(v) ∩ θ(v, w) ∩ ϱ(w) ≠ ∅}
8          𝒮 ← SCC_Decomposition(V', E')
9          foreach terminal SCC T ∈ 𝒮 do
10             ζ ← λv ∈ V.{c | c ∈ ϱ(v) ∧ v ∈ T}
11             p ← pr↓(𝒢, ζ)
12             α ← p mod 2
13             U^λ ← λv ∈ V.{c | c ∈ ζ(v) ∧ Ω(v) = p}
14             A^λ ← α-FAttr(ζ, U^λ)
15             (W_0^λ', W_1^λ') ← Zielonka-SCC(𝒢, ζ \ A^λ)
16             if W_ᾱ^λ' = ∅^λ then
17                 (W_α^λ'', W_ᾱ^λ'') ← (A^λ ∪ W_α^λ', ∅^λ)
18             else
19                 B^λ ← ᾱ-FAttr(ζ, W_ᾱ^λ')
20                 (W_0^λ', W_1^λ') ← Zielonka-SCC(𝒢, ζ \ B^λ)
21                 (W_α^λ'', W_ᾱ^λ'') ← (W_α^λ', B^λ ∪ W_ᾱ^λ')
22             end
23             (W_0^λ, W_1^λ) ← (W_0^λ ∪ α-FAttr(ϱ, W_0^λ''), W_1^λ ∪ α-FAttr(ϱ, W_1^λ''))
24             ϱ ← ϱ \ (α-FAttr(ϱ, W_0^λ'') ∪ α-FAttr(ϱ, W_1^λ''))
25         end
26         (W_0^λ', W_1^λ') ← Zielonka-SCC(𝒢, ϱ \ (W_0^λ ∪ W_1^λ))
27         return (W_0^λ ∪ W_0^λ', W_1^λ ∪ W_1^λ')
28     end
29 end
```

4.4 Correctness

We will now show the correctness of Algorithm 2 on the preceding page. To prove its correctness, we will use a similar approach to the proof of Zielonka's algorithm for VPGs from [1]. First, we observe that a terminal component $T \in \mathcal{S}$ of the (total by Lemma 2) graph (V', E') where $V' = \texttt{vert}(\varrho)$ and $E' = \{(v, w) \in E \mid \varrho(v) \cap \theta(v, w) \cap \varrho(w) \neq \emptyset\}$ is closed under reachability, i.e. for all $v \in T$ there does not exist $w \in V \setminus T$ such that $(v, w) \in E$ and that any play that enters T never leaves T.

Theorem 1. Let $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \theta)$ be a variability parity game and $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ. `Zielonka-SCC`$(\mathcal{G}, \varrho)$ returns mappings $W_0^\lambda, W_1^\lambda : V \to 2^{\mathfrak{C}}$ such that for all $v \in V$, $W_0^\lambda(v) \cup W_1^\lambda(v) = \varrho(v)$ and for each $c \in W_\alpha^\lambda(v)$ player α has a winning strategy from v for configuration c.

Proof. Let $|\varrho| = \sum_{v \in V} |\varrho(v)|$. We will prove the theorem by induction on $|\varrho|$.

Base Case: We have $\varrho = \emptyset^\lambda$. Therefore, by lines 2,3 and 4 of Algorithm 2, `Zielonka-SCC`$(\mathcal{G}, \varrho)$ returns $W_0^\lambda = \emptyset^\lambda$ and $W_1^\lambda = \emptyset^\lambda$ which trivially satisfy the statement.

Inductive Step: For our induction hypothesis, we assume that the theorem holds for all ϱ' such that $|\varrho'| < |\varrho|$. We have that $\varrho \neq \emptyset^\lambda$ and by our assumption $\mathcal{G}$ is total with respect to ϱ, and therefore both V' and E' are non-empty. As the graph (V', E') is non-empty, so is its decomposition, and therefore we have at least one smallest element in the decomposition $\mathcal{S}$, which is terminal. First, note that $T \in \mathcal{S}$ from line 9 is a terminal component of (V', E') (lines 6 and 7) and therefore closed under reachability in $\mathcal{G}$ restricted to ϱ. Next, ζ is the restriction where we only include vertex $v \in V$ and their configuration $\varrho(v)$ if $v \in T$, as per line 10. Next, we observe that $\mathcal{G}$ is total with respect to restriction ζ, since $\mathcal{G}$ is total with respect to ϱ and for all vertices in $v \in \texttt{vert}(\zeta)$ and their configurations $\zeta(v)$ we have that there exists an edge (v, w) such that $w \in \texttt{vert}(\zeta)$ (otherwise ϱ would not be a total restriction or T would not be a terminal SCC of (V', E')). Now let $U^\lambda \subseteq \zeta$ be the restriction containing the configurations $\zeta(v)$ only if v has minimum priority in the game, and $\emptyset$ otherwise (line 13). Next by Lemma 1 we note that $A^\lambda \subseteq \zeta$ and since A^λ is an α-maximal restriction, by Lemma 2 $\mathcal{G}$ restricted to $\zeta \setminus A^\lambda$ is also total. By our induction hypothesis $W_0^{\lambda'}, W_1^{\lambda'}$ on line 15 are mappings that satisfy the statement. Now we have two possible cases:

Case 1) The opponent does not have a winning strategy for any of the vertices - and their configurations - in $\zeta \setminus A^\lambda$. Since player α has a winning strategy for all vertices $v \in \texttt{vert}(\zeta \setminus A^\lambda)$ and all their configurations $c \in (\zeta \setminus A^\lambda)(v)$, they also have a winning strategy for all vertices $v \in \texttt{vert}(A^\lambda)$ and configurations in $A^\lambda(v)$, since for all configurations the play either stays in vertices of $\texttt{vert}(A^\lambda)$ forever, or they eventually leave $\texttt{vert}(A^\lambda)$ to

a vertex in $\texttt{vert}(\zeta \setminus A^\lambda)$ which is also won by player α. Hence the mappings $A^\lambda \cup W_\alpha^{\lambda\prime}$ and $\emptyset^\lambda$ are mappings such that for each player and vertex v we have that $W_\alpha^\lambda(v)$ contains the set of configurations such that player α has a winning strategy from v.

Case 2) The opponent has a winning strategy for some of the vertices and configurations in $\zeta \setminus A^\lambda$. Since some of the vertices and configurations in A^λ might be won by the opponent, we recompute the winning sets, this time removing all vertices in $\bar{\alpha}\text{-FAttr}(\varrho, W_{\bar{\alpha}}^{\lambda\prime})$, since player $\bar{\alpha}$ has a winning strategy for all configurations and vertices in $W_{\bar{\alpha}}^{\lambda\prime}$. Again, by our induction hypothesis and Lemma 2, $\zeta \setminus B^\lambda$ is a total restriction and the mappings $W_\alpha^{\lambda\prime}, W_{\bar{\alpha}}^{\lambda\prime}$ contain for each player and vertex in $\texttt{vert}(\zeta)$ the set of configurations for which the player has a winning strategy from v. For any vertex and configuration in B^λ that was not in $W_{\bar{\alpha}}^{\lambda\prime}$ we have a winning strategy: forcing the play towards vertices in $W_{\bar{\alpha}}^{\lambda\prime}$. Hence the mapping $B^\lambda \cup W_{\bar{\alpha}}^{\lambda\prime}$ contains for each vertex $v \in \texttt{vert}(\zeta)$ the set of configurations for which player $\bar{\alpha}$ has a winning strategy, and the same holds for $W_\alpha^{\lambda\prime}$ and player α.

In both cases we get mappings W_0^λ and W_1^λ and since the restriction ζ is closed under reachability, the player also has a winning strategy in the game $\mathcal{G}$ restricted to ϱ, since no play will ever leave the restriction ζ. Because of this, any vertices and configurations in the featured attractor set of $W_\alpha^{\lambda\prime\prime}$, which is computed on line 23, is also won by player α. Furthermore, if a vertex v and configuration c was not in any of the restrictions ζ, it is either in the attractor set of $W_0^{\lambda\prime\prime}$ or $W_1^{\lambda\prime\prime}$ in which case player even, respectively player odd, has a winning strategy, or it is in the remainder of the game $\mathcal{G}$ restricted to $\varrho \setminus (W_0^\lambda \cup W_1^\lambda)$, which is total by Lemma 2. By our induction hypothesis, the remainder of the game solved on line 26.

We can conclude that $\texttt{Zielonka-SCC}(\mathcal{G}, \varrho)$ returns mappings W_0^λ, W_1^λ for player even and odd such that $W_\alpha^\lambda(v)$ contains the set of configurations for which player α has a winning strategy from vertex v. □

4.5 Conclusions

We have introduced the concept of an SCC-family, which describes the strongly connected components in a VPG. We have given an approach to solve VPGs using this family-based decomposition into SCC-families and how to solve these decomposed games and its correctness. Furthermore, we have defined the *approximation* of an SCC-family and how to compute them. In the next section we will introduce a small progress measures algorithm for variability parity games.

Chapter 5

Small Progress Measures

In this section we will introduce a second algorithm for solving variability parity games based on the small progress measures (SPM) algorithm as first proposed in [19]. The algorithm discussed here is based on the SPM algorithm for variability parity games as put forward in [12]. First we will briefly introduce the required notation and the (non-deterministic) small progress measures algorithm from [12]. Next we describe the required changes required to implement the algorithm. For easier reading we will introduce an adapted version which is more in line with the notation throughout the rest of this thesis.

5.1 Jurdziński's Small Progress Measures

First, let $m \in \mathbb{N}^d$ be a d-tuple of non-negative integers, with d the maximum priority in a parity game $\mathcal{G}$. When comparing d-tuples $(<, \leq, =, \neq, \geq, >)$ we use the lexicographic ordering. Furthermore, let $<_i, \leq_i, =_i, \neq_i, \geq_i, >_i$ for some $i \in \mathbb{N}$ with $i < d$ be the lexicographic ordering applied to the first $i+1$ integers in the tuple m. For instance, we have $(2,3,0,1) > (2,3,0,0)$ and $(2,3,0,1) =_2 (2,3,0,0)$, as $(2,3,0) = (2,3,0)$.

Definition 5.1.1. (Parity progress measure). Let $\mathcal{G} = (V, E, \Omega, \mathcal{P})$ be a parity game. A function $\varrho : V \to \mathbb{N}^d$ is a parity progress measure for $\mathcal{G}$ if for all $(v, w) \in E$ we have $\varrho(v) \geqslant_{\Omega(v)} \varrho(w)$ where $\geqslant_{\Omega(v)}$ is defined as:

$$\begin{cases} \varrho(v) \geq_{\Omega(v)} \varrho(w) & \text{if } \Omega(v) \text{ is even} \\ \varrho(v) >_{\Omega(v)} \varrho(w) & \text{if } \Omega(v) \text{ is odd} \end{cases}$$

Next, we define $M_{\mathcal{G}}$ to be the following finite subset of $\mathbb{N}^d$:

$$M_{\mathcal{G}} = [0] \times [V^1] \times [0] \times [V^3] \times \cdots \times [V^{d-1}] \text{ with } V^p = |\Omega^{-1}(p)| \text{ and } [n] = \{0, \ldots, n\}.$$

if d is even. If d is odd, we have $[V^{d-2}] \times [0]$ at the end. So $M_{\mathcal{G}}$ is a finite set of

d-tuples such that we have zeros on the even positions, and non-negative integers on the odd positions i which are bounded by the amount of vertices with priority $d - i$.

We define the set $M_{\mathcal{G}}^{\top} = M_{\mathcal{G}} \cup \{\top\}$, where $\top$ is the largest element in $M_{\mathcal{G}}^{\top}$, i.e. for all $m \in M_{\mathcal{G}}$ and $i \in \mathbb{N}$ we have that $m <_i \top$.

Next we define the *Prog* operation:

$$Prog(\varrho, v, w) = \begin{cases} \min\{m \in M_{\mathcal{G}}^{\top} \mid m \geq_{\Omega(v)} \varrho(w)\} & \text{if } \Omega(v) \text{ is even} \\ \min\{m \in M_{\mathcal{G}}^{\top} \mid m >_{\Omega(v)} \varrho(w) \text{ or } m = \varrho(w) = \top\} & \text{if } \Omega(v) \text{ is odd} \end{cases}$$

Lastly, we define the *game parity progress measure*.

Definition 5.1.2. (Game parity progress measure). A function $\varrho : V \to M_{\mathcal{G}}^{\top}$ is a game parity progress measure if for all $v \in V$, we have:

- if $v \in V_0$ then $\varrho(v) \geq_{\Omega(v)} \text{Prog}(\varrho, v, w)$ for some $(v, w) \in E$, and
- if $v \in V_1$ then $\varrho(v) \geq_{\Omega(v)} \text{Prog}(\varrho, v, w)$ for all $(v, w) \in E$

Furthermore, with $||\varrho||$ we denote the set $\{v \in V \mid \varrho(v) \neq \top\}$.

We then have the following proposition from [19].

Proposition 1. Player even has a winning strategy for a vertex $v \in V$ iff $\varrho(v) \neq \top$, where ϱ is the smallest game parity progress measure.

The proof of Proposition 5.1 is omitted here, but can be found in [19]. Next we will describe how we can compute the smallest game parity progress measure.

First, we define an ordering $\sqsubseteq$, and a family of $\text{Lift}(\cdot, v)$ operators for all $v \in V$, on the set of functions $V \to M_{\mathcal{G}}^{\top}$. Given functions $\varrho, \mu : V \to M_{\mathcal{G}}^{\top}$, we define $\mu \sqsubseteq \varrho$ to hold if $\mu(v) \leq \varrho(v)$ for all $v \in V$. This ordering relation $\sqsubseteq$ gives a complete lattice structure on the set of functions $V \to M_{\mathcal{G}}^{\top}$. We write $\mu \sqsubset \varrho$ if $\mu \sqsubseteq \varrho$ and $\mu \neq \varrho$. We can now define the $\text{Lift}(\varrho, v)$ operator for $v \in V$ as follows:

$$\text{Lift}(\varrho, v)(u) = \begin{cases} \varrho(u) & \text{if } u \neq v \\ \max\{\varrho(v), \min_{(v,w)\in E} \text{Prog}(\varrho, v, w)\} & \text{if } u = v \in V_0 \\ \max\{\varrho(v), \max_{(v,w)\in E} \text{Prog}(\varrho, v, w)\} & \text{if } u = v \in V_1 \end{cases}$$

From this definition, it trivially follows that the $\text{Lift}(\cdot, v)$ operator is monotone, as it is always assigned a value of *at least* $\varrho(u)$. As we have a complete lattice and a

monotone operator on that lattice, Lift$(\cdot, v)$, by the Knaster-Tarski theorem, there must exist a least fixpoint.

In order to solve VPGs using parity progress measures, the authors from [12] define the parity progress measure for each vertex $v \in V$ and set of configurations $c \in 2^{\mathfrak{C}}$ for the VPG $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \theta)$. By taking into account the edge guards of the VPG, we can update the parity progress measure for each vertex in the VPG accordingly.

In Algorithm 3 we describe how to compute this smallest game parity progress measure for VPGs, which relies on the FPAttr, defined in Algorithm 4. As VPGs can be played for multiple configurations, the algorithm tries to compute the game parity progress measure (denoted with U) in the algorithm for each vertex and set of configurations $c \in 2^{\mathfrak{C}}$. Algorithm 3 and 4 are taken from [12] and have been adapted to the VPG notation used throughout this thesis. With $\underline{0}$ we denote the smallest game parity progress measure $(0, 0, 0, 0, \ldots)$.

Algorithm 3: Fixed point computation of the small progress measures algorithm for VPGs.

```
1  Fpattr*(G = (V, E, Ω, P, θ))
2      forall v ∈ V do
3          U(v)(𝔠) ← 0
4      end
5      repeat
6          U_old ← U
7          U ← max(U, FPattr(U))
8      until U = U_old;
9      return U
10 end
```

Note that Algorithm 3 tries to compute the (smallest) fixed point of the game parity progress measure by trying to lift the progress measures until we reach a stable point. The computation of the minimum and maximum of the function $U : V \to 2^{\mathfrak{C}} \to M_{\mathcal{G}}^{\top}$ is omitted here, but is similar to Algorithm 6 on page 24 and can be found in [12]. The same holds for the reduce function on line 18 of Algorithm 4 and is required to make sure the domain of $U(v)$ is a partition of the configurations $\mathfrak{C}$. In Section 5.2 we will discuss a more efficient approach to computing the new function U.

Algorithms 3 and 4 compute the game parity progress measure for all vertices depending on the old progress measure. This is not a very efficient way to compute the progress measure in practice, as most changes will be localised to a group of vertices, where most of the progress measures will stay the same. In the next section we will describe the set of changes required to implement the small progress measures algorithm for VPGs efficiently.

Algorithm 4: Algorithm to lift the parity progress measures for all vertices and configurations in the VPG $\mathcal{G}$.

```
1  Fpattr(G = (V, E, Ω, P, 𝔈, θ), U)
2      forall v ∈ V do
3          U'(v)(𝔈) ← 0̲
4          forall w ∈ vE do
5              foreach φ ∈ dom(U(w)) do
6                  Ψ ← θ(v, w) ∩ φ
7                  if Ψ ≠ ∅ then
8                      W(Ψ) ← min{m ∈ M_G^⊤ | m ⩾_Ω(v) U(w)(φ)}
9                  end
10             end
11             if P(v) = 0 then
12                 U'(v) ← MIN(U'(v), W)
13             else
14                 U'(v) ← MAX(U'(v), W)
15             end
16         end
17     end
18     U'(v) ← REDUCE(U'(v))
19     return U'
20 end
```

5.2 Adapted algorithm

In a VPG a player can have different strategies depending on the configuration for which the game is played, therefore we also have to keep track of the progress measures per configuration for all the vertices.

To keep track of these progress measures, in [12] the authors define a partial mapping $U : 2^{\mathfrak{C}} \to M_{\mathcal{G}}^{\top}$ for each vertex $v \in V$. The domain of U partitions $\mathfrak{C}$ and describes for each set $c \in \texttt{dom}(U)$ the parity progress measure associated with it, where a progress measure $m \in M_{\mathcal{G}}^{\top}$ is the same as that defined in the original algorithm.

In our version, we inverse this mapping such that each progress measure $m \in M_{\mathcal{G}}^{\top}$ in the (new) domain of U points to the set of configurations $c \in 2^{\mathfrak{C}}$ for which m is the current progress measure. This makes it easier to ensure that each progress measure m is only mapped to one set of configurations, by using a datastructure, such as a hashmap, which only allows for unique values in the domain. In Algorithm 6 we also ensure that this property holds.

In Algorithm 5 we describe our adapted version of the small progress measures algorithm. Note that instead of computing the new progress measures for all the vertices

like in the original algorithm as described in [12], we keep a queue Q of vertices which can potentially be lifted to a higher progress measure. Because a progress measure depends on the progress measures of its neighbours, we only have to recompute it for a vertex v if one of its neighbours has had its progress measure lifted. Hence, when we compute the progress measure for a vertex w and it has changed compared to the previous value, we add all vertices $v \in V$, such that $(v, w) \in E$, to the queue Q to be updated again. For the ordering of the queue we use a first in, first out (FIFO) ordering. With $(m \mapsto c)$ with $m \in M_{\mathcal{G}^\top}$ and $c \in 2^{\mathfrak{C}}$ we denote the partial function s.t. $(m \mapsto c)(m) = c$.

Algorithm 5: Adapted small progress measures algorithm for solving variability parity games.

```
1  Prog(G = (V, E, Ω, P, 𝔈, θ))
2      U ← λv ∈ V.(0̲ ↦ 𝔈)
3      Q ← V
4      while Q ≠ ∅ do
5          v ← Q.pop()
6          W(0̲) ← 𝔈
7          forall w ∈ vE do
8              foreach m_w ∈ dom(U(w)) do
9                  Ψ ← θ(v, w) ∩ U(w)(m_w)
10                 if Ψ ≠ ∅ then
11                     m_v ← min{n ∈ M_G^⊤ | n ⩾_{Ω(v)} m_w}
12                     X(m_v) ← Ψ
13                 end
14             end
15             if P(v) = 0 then
16                 W ← MIN(W, X)
17             else
18                 W ← MAX(W, X)
19             end
20         end
21         J ← U(v)
22         U(v) ← MAX(U(v), W)
23         if J ≠ U(v) then
24             Q ← Q ∪ {t | t ∈ Ev}
25         end
26     end
27 end
```

In order to compute the maximum and minimum of two different functions $U : M_{\mathcal{G}}^{\top} \to 2^{\mathfrak{C}}$ and $S : M_{\mathcal{G}}^{\top} \to 2^{\mathfrak{C}}$ as described on lines 16 and 18 of Algorithm 5 we need to compare all the progress measures we defined for the different sets of configurations $c \in \mathtt{rng}(U)$ and $c' \in \mathtt{rng}(S)$ and set the new progress measure of $c \cap c'$ to the

maximum, respectively minimum, of $U^{-1}(c)$ and $S^{-1}(c')$. This procedure for the MIN operation is described in Algorithm 6, the procedure of computing the MAX has been left out as it is very similar.

Algorithm 6: Algorithm to compute the minimum of two functions $U, S : M_{\mathcal{G}}^{\top} \to 2^{\mathfrak{C}}$.

```
1  MIN (U : M_G^⊤ → 2^𝔠, S : M_G^⊤ → 2^𝔠)
2      foreach c ∈ rng(U) do
3          m_w ← U^{-1}(c)
4          W(m_w) ← W(m_w) ∪ c
5          foreach c' ∈ rng(S) do
6              m_s ← S^{-1}(c')
7              if c' ∩ c ≠ ∅ then
8                  if m_s < W^{-1}(c) then
9                      W(m_s) ← c ∩ c'
10                 end
11             end
12         end
13     end
14     return W
15 end
```

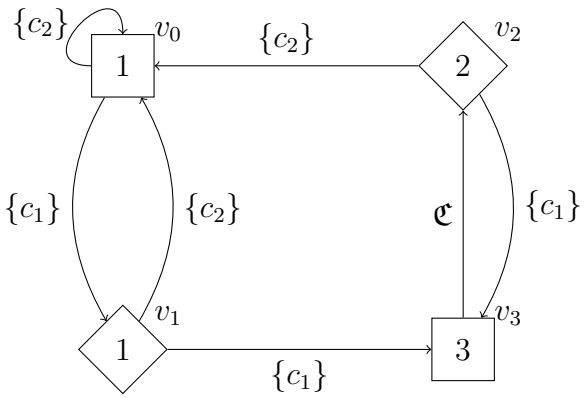


Figure 5.1: A small Variability Parity Game with $\mathfrak{C} = \{c_1, c_2\}$. The smallest fixed point of the parity progress measures can be found in Table 5.1 on the next page.

5.3 Example

In Figure 5.1 we can see an example of a variability parity game, with $\mathfrak{C} = \{c_1, c_2\}$. We will now go over a few iterations of the small progress measures algorithm from Algorithm 5.

Initially our queue Q will contain all vertices in the parity game, i.e. $[v_0, v_1, v_2, v_3]$. The initial ordering of this queue is not particularly important, as long as each vertex is considered at least once. After one iteration we have that the game parity progress measure of vertex v_0 is lifted from $U(v_0)(\mathfrak{C}) \mapsto (0,0,0,0)$ to $U(v_0)(\mathfrak{C}) \mapsto (1,0,0,0)$ and the queue is updated to $[v_1, v_2, v_3, v_0]$. Vertex v_1 is then also lifted to $U(v_1)(\mathfrak{C}) \mapsto (1,0,0,0)$. In case of vertex v_2, we have that $U(v_2)(\{c_2\}) \mapsto (1,0,0,0)$ and $U(v_2)(\{c_1\}) \mapsto (0,0,0,0)$, since the progress measure of $U(v_3)(\{c_1\}) \mapsto (0,0,0,0)$.

After several iterations, we get to the stable state as described in Table 5.1. As we can see, the vertices v_0, v_1, v_2, v_3 for configuration $\{b\}$ are won by player odd, as their progress measure is set to $\top$. The remainder of the vertices and their configurations are therefore won by player even.

v_0	$\{c_1\}$	$(0,2,0,0)$	v_2	$\{c_1\}$	$(0,0,0,0)$
	$\{c_2\}$	$\top$		$\{c_2\}$	$\top$
v_1	$\{c_1\}$	$(0,1,0,0)$	v_3	$\{c_1\}$	$(0,0,0,1)$
	$\{c_2\}$	$\top$		$\{c_2\}$	$\top$

Table 5.1: Solution for VPG from Figure 5.1 on the previous page

Chapter 6

Priority Promotion

In this chapter we will first informally describe the original priority promotion algorithm for parity games, as first put forward in [2]. Afterwards we will describe the adaptations needed to perform priority promotion on variability parity games and provide an algorithm and prove its correctness.

6.1 Original Priority Promotion

The main concept of the priority promotion algorithm is that of an α-dominion, which is the set of vertices from which $\bar{\alpha}$ can not escape and is won by player α.

Definition 6.1.1. (α-dominion). Let $\mathcal{G} = (V, E, \Omega, \mathcal{P})$ be a parity game. Furthermore, let $U \subseteq V$ be a set of vertices in the parity game $\mathcal{G}$. We say that U is an α-dominion for $\alpha \in \{0, 1\}$ if there exists a strategy σ_α for player α such that for all opponent strategies $\sigma_{\bar{\alpha}}$ and positions $v \in U$ the induced play π has $\text{pr}^{\downarrow}(\pi) \equiv_2 \alpha$, and $\pi_i \in U$ for all $i \geq 0$.

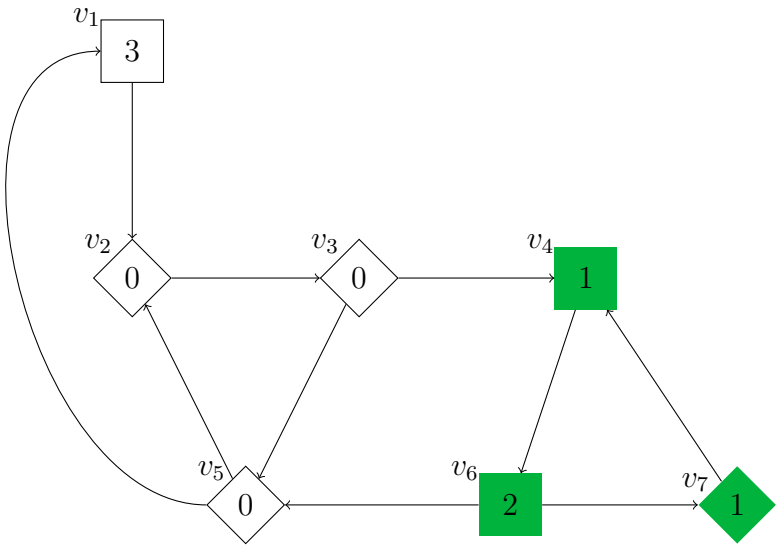


Figure 6.1: A parity game with in green the 1-dominion $D = \{v_4, v_6, v_7\}$.

Example 6.1.1. Consider the parity game from Figure 6.1 and let $U = \{v_4, v_6, v_7\}$. U is a 1-dominion, since the odd player has a strategy for which all positions and induced plays in U are won by player odd.

Next, we define the escape set for a set of vertices U, which is the set of vertices $v \in U$ through which player α can escape U.

Definition 6.1.2. (α-escape). Let $\mathcal{G} = (V, E, \Omega, \mathcal{P})$ be a parity game, and $U \subseteq V$ a set of vertices. We define the escape set of U as follows:

$$\begin{aligned}\mathtt{esc}^{\alpha}_{\mathcal{G}}(U) =& \{v \in V \mid \mathcal{P}(v) = \alpha \wedge \exists w \in vE : w \notin U\} \\ &\cup \{v \in V \mid \mathcal{P}(v) = \bar{\alpha} \wedge \forall w \in vE : w \notin U\}\end{aligned}$$

We say that a set of vertices $U \subseteq V$ is open, respectively closed, iff $\mathtt{esc}^{\alpha}_{\mathcal{G}}(U) \neq \emptyset$ and $\mathtt{esc}^{\alpha}_{\mathcal{G}}(U) = \emptyset$ respectively. A more general concept with respect to a dominion, is that of a quasi α-dominion, which is a set of vertices which is won by player α if the play does not leave the quasi dominion.

Definition 6.1.3. (Quasi α-dominion). Let $\mathcal{G} = (V, E, \Omega, \mathcal{P})$ be a parity game and α an arbitrary player. A non-empty set of positions $Q \subseteq V$ is a *quasi α-dominion* in $\mathcal{G}$ if there exists an α-strategy σ_α such that, for all $\bar{\alpha}$-strategies $\sigma_{\bar{\alpha}}$ and positions $v \in Q$, the induced play π satisfies $\mathrm{pr}^{\downarrow}(\pi) \equiv_2 \alpha$, if $\pi_i \in Q$ for all $i \in \mathbb{N}$, otherwise let $\pi^{\leq i}$ be the prefix such that $\pi_j \in Q$ for all $j \leq i$ and $\pi_{i+1} \notin Q$, then $\pi_i \in \mathrm{esc}^{\bar{\alpha}}_{\mathcal{G}}(Q)$.

Next, we introduce the α-region, which is a set of vertices in our parity game which the opponent $\bar{\alpha}$ can only leave through vertices of priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$.

Definition 6.1.4. (Region). A quasi α-dominion R is an α-region if $\mathrm{pr}^{\downarrow}(\mathcal{G}) \equiv_2 \alpha$ and all positions in $\mathtt{esc}^{\bar{\alpha}}(R)$ have priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$, which we call the *priority* of the region.

The above definition ensures that if an opponent $\bar{\alpha}$ can escape from an α-region, it must visit a position in the region which has lowest priority in the game and has parity α. Definition 6.1.4 allows for two operations crucial to priority promotion: *region merging* and *region extension*, taken from [2]. We say a set of vertices $R \subseteq V$ is α-maximal in $\mathcal{G}$ if $R = \alpha\text{-}\mathtt{Attr}_{\mathcal{G}}(R)$, where $\alpha\text{-}\mathtt{Attr}$ is the original attractor set definition, which does not take into account configurations.

Proposition 2. (Region Merging). Let $\mathcal{G}$ be a parity game, $R \subseteq V$ an α-region, and $D \subseteq V$ an α-dominion in the subgame $\mathcal{G}'$ where we remove vertices in R. Then, we have that $R^\star = R \cup D$ is an α-region in $\mathcal{G}$. Moreover, if both R and D are α-maximal in $\mathcal{G}$ and $\mathcal{G}'$ then $R^\star$ is α-maximal in $\mathcal{G}$ as well.

Proposition 3. (Region Extension). Let $\mathcal{G}$ be a parity game and $R^\star \subseteq V$ an α-region in $\mathcal{G}$. Then, $R = \alpha\text{-}\mathrm{Attr}_{\mathcal{G}}(R^\star)$ is an α-maximal region in $\mathcal{G}$.

With Proposition 2 and 3 we now have all the required ingredients for explaining the priority promotion mechanism.

First, we define the *region function* $r : V \to \mathbb{N}$, which is a function that maps each vertex $v \in V$ to the priority of the region it belongs to, where the *priority* of a region is the lowest priority of a vertex that is in the region. With $\mathcal{G}_{r \approx p}$ for some $p \in \mathtt{rng}(r)$ and region function r, we denote the subgame where we only include vertices from $\{v \in V \mid r(v) \approx p\}$, where $\approx \in \{=, \leq, \geq, <, >\}$. Lastly, we require for each $i \in \mathtt{rng}(r)$ that $r^{-1}(i)$ is a region in the subgame $\mathcal{G}_{r \geq i}$

The simplest region we can think of would be the set of all vertices with minimum priority in the game $\mathcal{G}$, i.e. $R = \{v \in V \mid \Omega(v) = \mathrm{pr}^{\downarrow}(\mathcal{G})\}$. From Proposition 3, we have that the attractor set $R^\star = \alpha\text{-attr}(R)$ with $\alpha \equiv_2 \mathrm{pr}^{\downarrow}(\mathcal{G})$, is an α-maximal region. Now we have two possibilities: either $R^\star$ is open in the game $\mathcal{G}$, or $R^\star$ is closed and we have found a dominion.

In case $R^\star$ is open in $\mathcal{G}$, we can continue to look for a region in the subgame $\mathcal{G} \backslash R^\star$. We can again construct a region by computing the attractor set of the set of vertices $R^{\star'} = \{v \in V \backslash R^\star \mid \Omega(v) = \mathrm{pr}^{\downarrow}(\mathcal{G} \backslash R^\star)\}$. We can keep constructing and removing α-maximal regions in this way until we eventually encounter a closed region.

Let $\mathcal{G}^\star$ be the current subgame we are considering. In case the region $R^\star$ is not closed in the entire game $\mathcal{G}$ but is closed in the subgame $\mathcal{G}^\star$, we can *promote* the region $R^\star$. Since $\mathtt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star) \neq \emptyset$, we have that there is at least one vertex $v \in R^\star$ from which we can escape to a vertex w in $\mathcal{G}$. Since this vertex w is not present in the subgame $\mathcal{G}^\star$, it must belong to a region of lower priority. Furthermore, this region is also of the same parity as region $R^\star$, as otherwise one of the previous regions we removed from the game was not α-maximal (since we could have added v which would have been attracted from vertex w). Since the region $R^\star$ is closed in $\mathcal{G}^\star$, this also means it is an α-dominion in $\mathcal{G}^\star$. Therefore we can *promote* $R^\star$ to the priority of this lower region and merge these regions by Proposition 2.

Let $r^{\underline{0}}$ be the original region function that for each vertex $v \in V$ sets $r^{\underline{0}}(v) = \Omega(v)$. Since each region of priority p consist of the set of vertices with priority p, it trivially holds that all plays that stay in the region are won by player $p \mod 2$ and that all vertices in the escape set of the region have priority p. Next, we define the update operator $\uplus$ for a region function r and a partial region function r' as follows:

$$(r \uplus r')(v) = \begin{cases} r'(v) & \text{if } v \in \mathtt{dom}(r') \\ r(v) & \text{otherwise} \end{cases} \qquad \text{for all } v \in V$$

The original priority promotion algorithm relies heavily on the search operation for finding dominions. In the variability parity game setting we will take the same approach. However, instead of computing dominions for a set of vertices in the graph, we will adapt it to compute the dominions for a *restriction* $U^\lambda : V \to 2^{\mathfrak{C}}$ of the VPG

graph. In the next section we will adapt the definitions used in priority promotion to apply to VPGs.

6.2 Adaptations for priority promotion in VPGs

Recall from Chapter 3 that we have a variability parity game $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$. We will now introduce some notation for the remainder of the section.

When describing a restriction $U^\lambda : V \to 2^{\mathfrak{C}}$ we use the following notation: with $U^\lambda = (v_1 \mapsto c_1, \ldots, v_i \mapsto c_i)$ for $v_j \in W \subseteq V$, and $c_j \in 2^{\mathfrak{C}}$ we denote the restriction where $U^\lambda(v_j) = c_j$ for all $v_j \in W$ and $U^\lambda(w_j) = \emptyset$ for all $w_j \in V \setminus W$.

Next we define the featured α-escape, which is again a restriction describing the vertices and configurations for which player α can leave the restriction U^λ in a single step. We say that a vertex v can reach vertex w for a configuration $c \in U^\lambda(v)$ with $v, w \in V$ iff $c \in \theta(v, w)$

Definition 6.2.1. (Featured α-escape). Let $U^\lambda : V \to 2^{\mathfrak{C}}$ be a restriction and $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ a VPG restricted to $\varrho : V \to 2^{\mathfrak{C}}$ and $U^\lambda \subseteq \varrho$, such that $\mathcal{G}$ is total with respect to ϱ. We define the featured α-escape as the set of configurations per vertex in U^λ from which player α can leave U^λ.

$$\begin{aligned}\mathtt{esc}^\alpha_{\mathcal{G}}(\varrho, U^\lambda)(v) =& \{c \in U^\lambda(v) \mid \mathcal{P}(v) = \alpha \wedge \exists w \in vE : c \notin U^\lambda(w) \wedge c \in \theta(v, w) \cap \varrho(w)\} \\ &\cup \{c \in U^\lambda(v) \mid \mathcal{P}(v) = \bar{\alpha} \wedge \forall w \in vE : c \in \theta(v, w) \cap \varrho(w) \implies c \notin U^\lambda(w)\}\end{aligned}$$

We leave out the restriction ϱ when it is clear and does not change in the context, where we write $\mathtt{esc}^\alpha(U^\lambda)$ instead of $\mathtt{esc}^\alpha_{\mathcal{G}}(\varrho, U^\lambda)$. We say that a restriction U^λ is open, respectively closed, in $\mathcal{G}$ restricted to ϱ, iff $\mathtt{esc}^\alpha_{\mathcal{G}}(U^\lambda) \neq \emptyset^\lambda$ or $\mathtt{esc}^\alpha_{\mathcal{G}}(U^\lambda) = \emptyset^\lambda$ respectively.

Example 6.2.1. Consider the VPG from Figure 6.2 with $U^\lambda = (a \mapsto \{c_1, c_2\}, e \mapsto \mathfrak{C}, c \mapsto \{c_1\}, i \mapsto \{c_3\})$ and $\varrho = (a \mapsto \mathfrak{C}, e \mapsto \mathfrak{C}, c \mapsto \mathfrak{C}, i \mapsto \mathfrak{C})$. The set of configurations such that player even can force to leave the restriction U^λ is $\mathtt{esc}^0(\varrho, U^\lambda) = (a \mapsto \emptyset, e \mapsto \{c_2\}, c \mapsto \{c_1\}, i \mapsto \emptyset)$. The odd player can stay in U^λ from vertex a, since they can move to vertex e for both configurations c_1 and c_2. However, the odd player *must* leave U^λ from vertex e for configuration c_2, as the only option is to move to vertex c, however $c_2 \notin U^\lambda(c)$. For vertex c we can leave U^λ for configuration c_1, since we can move to vertex e or vertex i, and both $c_1 \notin U^\lambda(e)$ and $c_1 \notin U^\lambda(i)$. Lastly, the even player can not force the odd player to leave U^λ from vertex i, since the odd player can remain in U^λ.

Definition 6.2.2. (Featured α-dominion). Let $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ be a VPG and $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ. Let $U^\lambda : V \to 2^{\mathfrak{C}}$ be a non-empty sub-mapping of ϱ. We say that U^λ is a featured α-dominion iff there

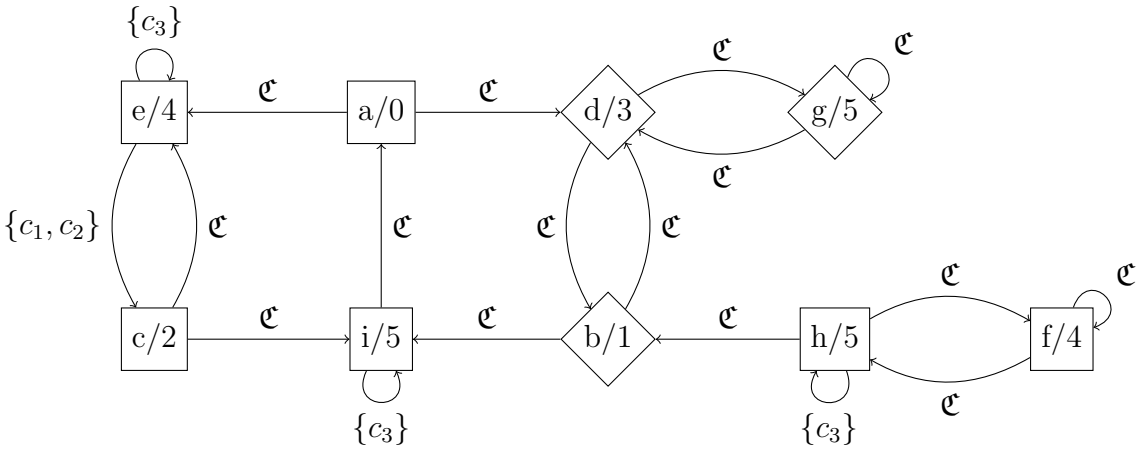


Figure 6.2: Running example of a variability parity game.

exists a strategy for player α such that for all $\bar{\alpha}$-strategies, player α has a winning strategy for all vertices $v \in \texttt{vert}(U^\lambda)$ and configurations $c \in U^\lambda(v)$ in $\mathcal{G}$ restricted to U^λ.

Example 6.2.2. Consider again the VPG from Figure 6.2. Consider the restriction $U^\lambda = (b \mapsto \{c_3\}, d \mapsto \{c_3\}, g \mapsto \{c_3\}, i \mapsto \{c_3\})$ and $\varrho = (\lambda v \in V.\mathfrak{C})$. Since $\texttt{esc}^0(U^\lambda) = \emptyset^\lambda$, and all infinite paths in (U^λ) are won by player odd, U^λ is a featured 1-dominion.

Lastly, we define the adaptations of the *quasi α-dominion* and *α-region*.

Definition 6.2.3. (Featured quasi α-dominion). Let $\mathcal{G} = (V, E, \Omega, \mathcal{P}, \mathfrak{C}, \theta)$ be a VPG and $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ. Let $Q^\lambda : V \to 2^{\mathfrak{C}}$ be a non-empty sub-mapping of ϱ, Q^λ is a quasi α-dominion iff there exists a strategy for player α such that for all $\bar{\alpha}$-strategies player α has a winning strategy for all vertices $v \in \texttt{vert}(Q^\lambda)$ and configurations $c \in Q^\lambda(v)$ in the game $\mathcal{G}$ restricted to ϱ, or the play leaves vertices or configurations of Q^λ.

Furthermore, we define the concept of *open* and *closed* quasi α-dominions. We say a featured quasi dominion Q^λ is open or closed, if $\texttt{esc}^{\bar{\alpha}}(Q^\lambda) \neq \emptyset^\lambda$ respectively $\texttt{esc}^{\bar{\alpha}}(Q^\lambda) = \emptyset^\lambda$. Note that a closed featured quasi α-dominion is an α-dominion.

Definition 6.2.4. (Featured α-region). Again let $\mathcal{G}$ be a VPG and $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ. Let $R^\lambda : V \to 2^{\mathfrak{C}}$ be a non-empty sub-mapping of ϱ that is a quasi α-dominion in $\mathcal{G}$ restricted to ϱ, we say R^λ is an α-region if the lowest priority in $\mathcal{G}$ restricted to ϱ is of parity α and all vertices in $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}(R^\lambda))$ have lowest priority $\Omega(v) = \text{pr}^{\downarrow}(\mathcal{G}, \varrho)$.

Example 6.2.3. Consider the VPG from Figure 6.2. Take $R^\lambda = (a \mapsto \mathfrak{C}, i \mapsto \{c_1, c_2\})$. R^λ and $\varrho = (\lambda v \in V.\mathfrak{C})$is an open 0-region, since the plays that remain in R^λ (none) are won by player 0 and $\texttt{esc}^1_{\mathcal{G}}(\varrho, R^\lambda) = (a \mapsto \mathfrak{C})$ and $\Omega(a) = 0 \equiv_2 0$.

Next, we define region *merging* and *extension* for VPGs as follows.

Proposition 4. (Region Merging) Let $\mathcal{G}$ be a VPG, $\varrho : V \to 2^{\mathfrak{C}}$ a restriction such that $\mathcal{G}$ is total with respect to ϱ, $R^\lambda : V \to 2^{\mathfrak{C}}$ an α-maximal region such that R^λ is a sub-mapping of ϱ and $D^\lambda : V \to 2^{\mathfrak{C}}$ an α-dominion in the game $\mathcal{G}$ restricted to $\varrho \setminus R^\lambda$, which is total according to Lemma 2. Then $R^\star = R^\lambda \cup D^\lambda$ is an α-region in $\mathcal{G}$ restricted to ϱ. Moreover, if both R^λ and D^λ are α-maximal in $\mathcal{G}$ and $\mathcal{G}$ restricted to $\varrho \setminus R^\lambda$ respectively, then $R^\star$ is α-maximal as well.

Proof. To show that $R^\star$ is still an α-region, we will have to show that $R^\star$ is (i) a quasi α-dominion and (ii) the minimum priority p in $\mathcal{G}$ is of parity α and for all $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star))$, v has minimum priority p.

Let σ_α be the combined strategy of player α for region R^λ and dominion D^λ, and $\sigma_{\bar{\alpha}}$ be the opponent strategy for R^λ and D^λ. We will show that for all $v \in \texttt{vert}(R^\star)$ the induced play π is won by player α if it remains in $R^\star$, or the vertices in $\texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star))$ have priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$. We consider the following cases:

- π always stays in D^λ. As D^λ is an α-dominion, the play π will be won by player α.
- π always stays in R^λ. As R^λ is an α-region, π will also be won by player α.
- π goes through vertices of R^λ and D^λ infinitely often. As π must leave R^λ infinitely often, it must also pass through a vertex in $\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(\varrho, R^\lambda)$ infinitely often. Since R^λ is an α-region all $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda))$ must have priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$ and are of parity α, which implies that π is won by α.
- π eventually leaves $R^\star$. As D^λ is an α-dominion in $\mathcal{G}$ restricted to $\varrho \setminus R^\lambda$, the play π is only able to leave D^λ through a vertex in the restriction R^λ. Hence $\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star) \subseteq \texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda)$. As all vertices $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda))$ have priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$ and parity α, $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star))$ must have priority $\mathrm{pr}^{\downarrow}(\mathcal{G})$ of parity α as well.

As we have shown that $R^\star$ is a quasi α-dominion and that the lowest priority p is of parity α and all vertices in $\texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(\varrho, R^\star))$ have priority p, we can conclude that $R^\star$ is an α-region.

To prove that $R^\star$ is α-maximal if R^λ and D^λ are α-maximal, suppose that R^λ and D^λ are α-maximal, and $R^\star$ is *not* α-maximal. This means that there exists a vertex $v \notin \texttt{vert}(R^\star)$ from which player α can force a play into $R^\star$. As $R^\star = R^\lambda \cup D^\lambda$, v enters $R^\star$ through a vertex w, where either $w \in \texttt{vert}(R^\lambda)$ or $w \in \texttt{vert}(D^\lambda)$. However, this contradicts the maximality of R^λ or D^λ. Hence by contradiction, we have that $R^\star$ is also α-maximal. □

Proposition 5. (Region Extension) Let $\mathcal{G}$ be a game restricted to ϱ and $R^\lambda \subseteq \varrho$ an α-region in $\mathcal{G}$. Then $R^\star = \alpha\text{-}\texttt{FAttr}(\varrho, R^\lambda)$ is an α-maximal α-region in $\mathcal{G}$ restricted to ϱ.

Proof. Since R^λ is an α-region in $\mathcal{G}$, we have that the minimum priority p in $\mathcal{G}$ must be of parity α. Furthermore, let σ_α be the combined strategy of player α for region R^λ and $\sigma_{\bar{\alpha}}$ be the opponent strategy in $R^\star$. Now, for all opponent strategies and vertices $v \in \texttt{vert}(R^\star)$ the play π we have that either:

- π remains in $R^\star$ forever. As $R^\star = \alpha\text{-FAttr}(\varrho, R^\lambda)$ the play π will eventually reach region $R^\star$. As R^λ is an α-region, the play π will be won by player α.
- π eventually leaves region $R^\star$. As $R^\star = \alpha\texttt{-FAttr}(\varrho, R^\lambda)$ we have that $\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star) \subseteq \texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda)$ since $\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda) \setminus R^\lambda = \emptyset^\lambda$ as otherwise a vertex v would not be in the attracted set $R^\star \setminus R^\lambda$. Since R^λ is an α-region, all vertices $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda))$ have lowest priority p in $\mathcal{G}$ and are of parity α. Therefore, the same holds for all vertices $v \in \texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star))$.

In both cases $R^\star$ is still an α-region. Lastly, it trivially holds that $R^\star$ is α-maximal, since $R^\star = \alpha\texttt{-FAttr}(\varrho, R^\lambda)$. □

This gives us all the concepts required to describe the priority promotion algorithm, which we will describe in the next section.

6.3 A Priority Promotion Algorithm for Variability Parity Games

In the previous section we adapted the concepts required for the priority promotion algorithm to VPGs. In this section we will introduce some auxiliary concepts which are required for the algorithm (but are not interesting outside the priority promotion algorithm components), and lastly the priority promotion algorithms.

First, we generalise region functions r to VPGs as follows: let $\mathcal{G}$ be a VPG restricted to ϱ, such that $\mathcal{G}$ is total with respect to ϱ. Then $r : V \to 2^{\mathfrak{C}} \to \mathbb{N}$ is the region function, where we now require that for all $i \in \texttt{rng}(r)$ we have that the restriction $r^{-1}(i)$ is a region in $\mathcal{G}$ restricted to ϱ and that $\texttt{dom}(r(v))$ is a partition of the set $\varrho(v)$, for all $v \in V$. Furthermore, we say that a region function r is maximal below p for some $p \in \texttt{rng}(r)$ iff for all $q \in \texttt{rng}(r)$ such that $q < p$ the restriction $r^{-1}(q)$ is an α-maximal region, with $\alpha = q \mod 2$. Lastly, we use the notation $\mathcal{G}_{r \leq p}$ to denote the game $\mathcal{G}$ restricted to $\varrho = \left(\bigcup_{i \leq p} r^{-1}(i) \right)$. Hence we denote the escape set of a restriction $U^\lambda \subseteq \varrho$ as $\texttt{esc}^{\alpha}_{\mathcal{G}_{\leq p}}(U^\lambda) = \texttt{esc}^{\alpha}_{\mathcal{G}}(\varrho, U^\lambda)$. The region function r is used to restrict the game $\mathcal{G}$ to the subgame we are using, as well as containing all information about the regions in the game.

When the region is open in the entire game, but closed in the subgame, the opponent can escape to a region with the *best escape priority*. We define $\texttt{bep}_r^\alpha(A^\lambda)$ as the function that returns the highest priority (according to the region function r) of the set of vertices that player α can escape restriction A^λ from, formally: $\texttt{bep}_r^\alpha(A^\lambda) \triangleq \max\{n \in \texttt{rng}(r) \mid \exists (v,w) \in E : A^\lambda(v) \cap \theta(v,w) \neq \emptyset \wedge A^\lambda(v) \cap \theta(v,w) \subseteq r^{-1}(n)(w)\}$.

We again use the notation $r^{\underline{0}}$ to denote the 'original' region function, which for a VPG restricted to ϱ is defined as $r^{\underline{0}}(v)(\varrho(v)) = \Omega(v)$ for all $v \in V$. Again, since each region of parity p consists of all vertices with parity p, each restriction in $r^{\underline{0}}$ is trivially a region. We say a region function is a *partial* region function if there exists a vertex $v \in V$ and configuration $c \in \varrho(v)$ for which $\neg\exists C \in \texttt{dom}(r(v)) : c \in C$. Let $r^{\leq m}$ denote the region function r restricted to the largest domain such that for all $v \in \texttt{dom}(r^{\leq m})$ and $C \in \texttt{dom}(r^{\leq m}(v))$ we have $r^{\leq m}(v)(C) \leq m$. Similarly we can define the restricted functions $r^{<m}, r^{>m}$ and $r^{\geq m}$, which are all partial region functions.

We use the operator $\uplus$ to merge the region functions $r^{\underline{0}}$ and $r^{\leq p^\star}$ on line 14 of Algorithm 7. First, let r be a region function and r' a *partial* region function. For each $v \in \texttt{r}$ let $\mathcal{C}(v) = \texttt{dom}(r'(v)) \cup \{C \in 2^{\mathfrak{C}} \mid \exists C' \in \texttt{dom}(r(v)) : C = C' \setminus \bigcup_{C'' \in \texttt{dom}(r'(v))} C''\}$. We then define the update function $\uplus$, for all $v \in V$ and $C \in \mathcal{C}(v)$ as follows:

$$(r \uplus r')(v)(C) = \begin{cases} r'(v)(C) & \text{if } C \in \texttt{dom}(r'(v)) \\ r(v)(C') & \text{if } C \notin \texttt{dom}(r'(v)) \text{for the unique } \mathcal{C}' \text{ such that } \mathcal{C} \subseteq C' \end{cases}$$

Lastly, we use the notation $r[U^\lambda \mapsto i]$ for a sub-mapping U^λ of ϱ and $i \in \texttt{rng}(\Omega)$ to update the region function r, formally $r[U^\lambda \mapsto i] = r \uplus \big(\lambda v \in \{u \in V \mid U^\lambda(u) \neq \emptyset\}.(U^\lambda(v) \mapsto i)\big)$.

In Algorithm 8 we can see the adapted priority promotion algorithm for VPGs, and Algorithm 7 describes how to find a region in the game $\mathcal{G}$. In Algorithm 8, U^λ is the restriction that contains for each vertex $v \in V$ the set of configurations for which we still need to find a solution. We then find a dominion in the game $\mathcal{G}$ restricted to U^λ and add the dominion to the winning set of player even or odd, until we have solved the entire game. Algorithm 7 uses the region function to check if the region of priority p that we are currently considering is open or closed in the entire game $\mathcal{G}$ and the restricted game $\mathcal{G}_{r^{\geq p}}$ respectively. Using region merging and region extension, we ensures that r is maximal below p until we find a dominion for player α.

Algorithm 7: VPG Dominion-searcher

```
1  searchDominionVPG (G = (V, E, Ω, P, 𝔠, θ), r, p)
2      α ← p mod 2
3      R ← r^{-1}(p)
4      R* ← α-FAttr(G_{r≥p}, R)
5      if esc^ᾱ_G(R*) = ∅^λ then
6          D^λ ← α-FAttr(G_r, R*)
7          return (α, D^λ)
8      else
9          if esc^ᾱ_{G_{r≥p}}(R*) ≠ ∅^λ then
10             r* ← r[R* ↦ p]
11             p* ← min{n ∈ rng(r) | r*^{-1}(n) ≠ ∅^λ ∧ n > p}
12         else
13             p* ← bep^ᾱ_r(R*)
14             r* ← (r^0 ⊎ r^{≤p*}) [R* ↦ p*]
15         end
16         return searchDominionVPG(G, r*, p*)
17     end
18 end
```

Algorithm 8: VPG Priority Promotion

```
1  Priority Promotion G = (V, E, Ω, P, 𝔠, θ)
2      (W_0, W_1) ← (∅^λ, ∅^λ)
3      U^λ ← (λv ∈ V.𝔠)
4      while U^λ ≠ ∅^λ do
5          p ← min{Ω(v) | v ∈ vert(U^λ)}
6          r ← λv ∈ V.(U^λ(v) ↦ Ω(v))
7          (α, D^λ) ← searchDominionVPG(G, r, p)
8          (W_α) ← W_α ∪ D^λ
9          U^λ ← U^λ \ D^λ
10     end
11     return (W_0, W_1)
12 end
```

6.3.1 Early termination of dominion search

The dominion-searcher as described earlier only returns a dominion if it is closed in the entire game $\mathcal{G}$. This requires that $\mathtt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda) = \emptyset^\lambda$, in other words, there is no configuration such that player $\bar{\alpha}$ can force the play to leave the region R^λ. This means that if the region is not closed, we first need to promote or solve the vertices

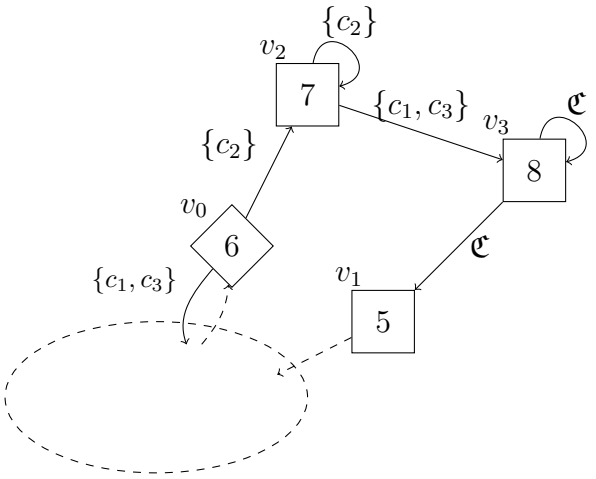


Figure 6.3: Early termination of a region of priority 6 that contains vertices v_0, v_1 for the set of configurations $\{c_1, c_2, c_3\}$.

with higher priorities before we can return the region R^λ.

Consider the situation as illustrated in Figure 6.3. Let $\mathcal{G}$ be total with respect to $\varrho = (v_0 \mapsto \mathfrak{C}, v_1 \mapsto \mathfrak{C}, v_2 \mapsto \mathfrak{C}, v_3 \mapsto \mathfrak{C}, \ldots)$. We have a region $R^\lambda = (v_0 \mapsto \{c_1, c_2, c_3\}, v_1 \mapsto \{c_1, c_2, c_3\}, v_2 \mapsto \emptyset, v_3 \mapsto \emptyset, \ldots)$ of priority 6. The region is not closed, as $\texttt{esc}^1_{\mathcal{G}} = (v_0 \mapsto \{c_2\})$, and we will continue to look for a dominion in the game $\mathcal{G}$ restricted to $\varrho \setminus R^\lambda$. This means we will only be able to solve the aforementioned region R^λ if the vertex of priority 7 with configuration c_2 is removed from the game. However, the region $R^\lambda = (v_0 \mapsto \{c_1, c_3\}, \ldots)$ is closed. Instead of continuing the search, we can compute the smallest set of configurations such that the region is closed in the subgame $\mathcal{G}$. We can do this according to the following set of rules:

$$\text{Let } R^{\text{pr}^\downarrow(R^\lambda)}(v) = \begin{cases} R^\lambda(v) & \text{if } \Omega(v) = \text{pr}^\downarrow(R^\lambda) \\ \emptyset & \text{if } \Omega(v) \neq \text{pr}^\downarrow(R^\lambda) \end{cases}$$

Then we compute the new region as follows:

$$R^{\lambda'} = \alpha\text{-}\texttt{FAttr}\left(\mathcal{G}, R^{\text{pr}^\downarrow(R^\lambda)} \setminus \texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\lambda)\right)$$

As this smaller region is still a valid dominion in the VPG, removing it earlier does not have an impact on the rest of the game.

There are cases where early termination might require more searches; respectively less searches. Consider again for instance the VPG from Figure 6.3. When we terminate our search early, we will have to search the VPG three times before we solved it entirely, namely: the region of priority 6, region of priority 7 and lastly the region of priority 8 consisting of only vertex v_3. Compare this with the amount of searches when not terminating early, in which case we will only need two searches: the first dominion returned will be that of priority 7 and lastly the region of priority 6, with the region of priority 8 promoted to priority 6 during the search.

6.3.2 Escape-set approximation

In Section 6.2 we discussed the α-escape set for a restriction in a VPG. In the priority promotion algorithm the escape set is computed to determine whether a region is open or closed in the entire game $\mathcal{G}$ and subgame $\mathcal{G}_{r\geq p}$. In both cases we are not interested in the exact escape-set, only whether it is empty or not. In Algorithm 9 we describe how to more efficiently find if the escape set in the subgame is open in the subgame $\mathcal{G}_{r\geq p}$. Since R^λ is a region, by definition, the only vertices which we can leave through will be of priority $p = \mathrm{pr}^{\downarrow}(\mathcal{G}_{r\geq p})$ and we can restrict the search to vertices of priority p. Note that we can stop our search as soon as we have found a vertex through which we can leave the region.

When computing the escape-set of R^λ in the entire game $\mathcal{G}$, and we already know that the region is closed in the subgame $\mathcal{G}_r^{\geq p}$, we only have to look if there is a vertex owned by the opponent that can leave the region R^λ. We don't have to consider vertices that belong to the owner of the region R^λ since any vertices owned by player α will not be able to leave the region either, as they would have been attracted in an earlier subgame. The computation of the `bep` and finding whether it is open or closed can then be combined, similarly as described in Algorithm 9 where we keep track of the best escape priority throughout the search.

Algorithm 9: Escape set approximation for a region R^λ of priority p.

```
1  Escape(G = (V, E, Ω, P, 𝔈, θ), R^λ, p)
2      α ← p mod 2
3      P ← {v ∈ vert(R^λ) | Ω(v) = p}
4      for v ∈ P do
5          if P(v) = α then
6              ESC ← ∅
7              for w ∈ vE do
8                  ESC ← ESC ∪ (R^λ(v) ∩ θ(v, w) ∩ ϱ(w))
9                  ESC ← ESC \ (R^λ(v) ∩ θ(v, w) ∩ R^λ(w))
10             end
11             if ESC ≠ ∅ then
12                 return TRUE
13         else if P(v) = ᾱ then
14             for w ∈ vE do
15                 if (R^λ(v) ∩ θ(v, w) ∩ ϱ(w)) \ R^λ(w) ≠ ∅ then
16                     return TRUE
17             end
18         end
19     end
20     return FALSE
21 end
```

Algorithm 9 returns if the escape set of region R^λ of priority p is open or closed in the restricted game $\mathcal{G}_r^{\geq p}$.

6.3.3 Example

In Table 6.1 we can see an example of the dominion search on the game illustrated in Figure 6.2. A downward arrow denotes a region that is open in its respective subgame, an upward arrow denotes a region being promoted, and a star denotes the search finding a closed region (and therefore also a dominion) in the entire game $\mathcal{G}$. The index on the left indicate the priority of the region, the numbers in the second column indicate when a search has been reset. Note that when describing a region in Table 6.1 we usually leave out the elements in the domain which map to the empty set, except to explicitly indicate that that vertex already belongs to a region with higher priority.

The first region returned by the `searchDominionVPG` function is the region $(a \mapsto \mathfrak{C}, i \mapsto \{c_1, c_2\}, b \mapsto \{c_1, c_2\}, d \mapsto \{c_1, c_2\}, g \mapsto \{c_1, c_2\})$ of parity even and priority 0, which is open. Next we find the region of priority 1, which is just $(b \mapsto \{c_3\})$, and compute its maximal α-region in the subgame where we remove the regions of higher priority, which is $(b \mapsto \{c_3\}, h \mapsto \{c_3\}, f \mapsto \{c_3\})$. This search continues until we find the closed region of parity 2 $(f \mapsto \{c_1, c_2\}, h \mapsto \{c_1, c_2\}, e \mapsto \{c_3\})$, which is not closed in the entire game $\mathcal{G}$ but the only opponent move exiting the region is to the region of priority 0 which has the same parity, thus the priority can be promoted. This continues until we find the dominion $D^\lambda = (b \mapsto \{c_3\}, h \mapsto \{c_3\}, f \mapsto \{c_3\}, d \mapsto \{c_3\}, i \mapsto \{c_3\}, g \mapsto \{c_3\})$ of priority 1. As D^λ is a 1-dominion, we also know that every vertex $v \in \texttt{vert}(D^\lambda)$ is won by player 1 for configuration(s) $D^\lambda(v)$. The search for dominions then continues in the game $\mathcal{G}$ restricted to $\varrho \setminus D^\lambda$.

6.4 Correctness

Next we will prove the correctness of our `searchDominionVPG` function. In order to do this, let us first introduce the *state space* of our search, which is adapted for VPGs from [2].

Definition 6.4.1. (State Space) We define the state space of our *dominion search* as the tuple $\mathcal{S}_\mathcal{G} = \langle S_\mathcal{G}, \prec_\mathcal{G} \rangle$ where its components are defined as follows:

1. $S_\mathcal{G} \subseteq \mathbb{R} \times \mathrm{Pr}_\mathcal{G}$, where $\mathbb{R}$ is the set of all region functions and $\mathrm{Pr}_\mathcal{G} = \texttt{rng}(\Omega)$ the set of all priorities in $\mathcal{G}$. A state $s = (r, p)$ is composed of a region function r and a priority p such that r is maximal below p and $p \in \texttt{rng}(r)$.

	1
6	$(a \mapsto \mathfrak{C}, i \mapsto \{c_1, c_2\}, b \mapsto \{c_1, c_2\}, d \mapsto \{c_1, c_2\}, g \mapsto \{c_1, c_2\})\downarrow$
5	$(b \mapsto \{c_3\}, h \mapsto \{c_3\}, f \mapsto \{c_3\})\downarrow$
4	$(c \mapsto \mathfrak{C}, e \mapsto \{c_1, c_2\})\downarrow$
3	$(d \mapsto \{c_3\})\downarrow$
2	$(f \mapsto \{c_1, c_2\}, h \mapsto \{c_1, c_2\}, e \mapsto \{c_3\})\uparrow_6$
	2
6	$(a \mapsto \mathfrak{C}, i \mapsto \{c_1, c_2\}, b \mapsto \{c_1, c_2\}, d \mapsto \{c_1, c_2\}, g \mapsto \{c_1, c_2\}, f \mapsto \{c_1, c_2\}, h \mapsto \{c_1, c_2\}, e \mapsto \{c_3\})\downarrow$
5	$(b \mapsto \{c_3\}, h \mapsto \{c_3\}, f \mapsto \{c_3\})\downarrow$
4	$(c \mapsto \mathfrak{C}, e \mapsto \{c_1, c_2\})\downarrow$
3	$(d \mapsto \{c_3\})\downarrow$
1	$(i \mapsto \{c_3\}, h \mapsto \emptyset, g \mapsto \{c_3\})\uparrow_3$
	3
⋮	
3	$(d \mapsto \{c_3\}, i \mapsto \{c_3\}, h \mapsto \{c_3\}, g \mapsto \{c_3\})\uparrow_5$
	4
⋮	
5	$(b \mapsto \{c_3\}, h \mapsto \{c_3\}, f \mapsto \{c_3\}, d \mapsto \{c_3\}, i \mapsto \{c_3\}, g \mapsto \{c_3\})_\star$

Table 6.1: Dominion search in VPG of Figure 6.2 on page 30.

2. For any two states $s_1 \triangleq (r_1, p_1), s_2 \triangleq (r_2, p_2) \in S_{\mathcal{G}}$, it holds that $s_1 \prec s_2$ iff either (a) there exists a priority $q \in \mathtt{rng}(r_1)$ with $q \leq p_1$ such that (a.i) $r_1^{<q} = r_2^{<q}$ and (a.ii) $r_2^{-1}(q) \subset r_1^{-1}(q)$, or both (b.i) $r_1 = r_2$ and (b.ii) $p_1 > p_2$ hold.

Note that without this state space we would not be able to formally reason about our search, as neither the region function r nor the priority p on their own are strictly increasing in the search. However, before we can continue, we will have to show that our new state space is a well-founded partial ordering w.r.t $\prec_{\mathcal{G}}$.

Lemma 5. The state space $\mathcal{S} = \langle S_{\mathcal{G}}, \prec_{\mathcal{G}} \rangle$ is a well-founded partial order w.r.t $\prec_{\mathcal{G}}$.

The proof of Lemma 5 is omitted here, as it is very similar to the original well-foundedness proof from the original paper on priority promotion [2]. For completeness the proof is included in Appendix A. With this ordering we can now prove the correctness of our `searchDominionVPG` function. Note that, although not explicitly mentioned in the proof, every restricted subgame $\mathcal{G}_{r \leq p}$ in which we are searching for a restriction is total. Since r is α-maximal below p, every restriction $r^{-1}(p')$, with $p' < p$, we exclude from the restricted game, is α-maximal hence $\mathcal{G}_{r \leq p}$ is total by Lemma 2.

Theorem 2. $\mathtt{searchDominionVPG}(\mathcal{G}, r, p)$ with $(r, p) \in S_{\mathcal{G}}$ returns an α-maximal dominion in game $\mathcal{G}$ and the player $\alpha \in \{0, 1\}$ that it belongs to.

Proof. Since $(r, p) \in S_{\mathcal{G}}$ we know that $\mathtt{searchDominionVPG}(\mathcal{G}, r, p)$ will terminate, since all subsequent calls to `searchDominionVPG` are with states (r', p') such that $(r', p') \prec_{\mathcal{G}} (r, p)$ and there are only a finite amount of states in $S_{\mathcal{G}}$. Hence it remains to prove that it returns an α-maximal dominion in $\mathcal{G}$. We will prove our claim by induction on the well-founded partial order $\mathcal{S}$.

Base case Let $s = (r, p)$ be a smallest element in $S_{\mathcal{G}}$, i.e. $\neg\exists s' \in S_{\mathcal{G}} : s' \prec_{\mathcal{G}} s$. For the sake of contradiction, assume that $R^\star = \alpha\text{-FAttr}(\varrho, r^{-1}(p))$ is an open α-region for $\alpha = p \mod 2$. We now have two cases:

1) $R^\star$ is open in the game $\mathcal{G}_{r \geq p}$. This means there exists a region of priority $q > p$ such that for some $v \in \mathtt{vert}(\mathtt{esc}^{\bar{\alpha}}_{\mathcal{G}_{r \geq p}}(R^\star))$ there exists $w \in vE$ such that $\mathtt{esc}^{\bar{\alpha}}_{\mathcal{G}_{r \geq p}}(R^\star)(v) \cap \theta(v, w) \cap r^{-1}(q)(w) \neq \emptyset$. However, this contradicts our assumption that s is a smallest element, as we now have a smaller state $s' \prec_{\mathcal{G}}$ such that $s' = (r', p')$ with $p' = \min\{n \in \mathtt{rng}(r) \mid r^{*^{-1}}(n) \neq \emptyset^\lambda \wedge n > p\}$ and $r' = r[R^\star \mapsto p]$. Observe that s' is a state in $S_{\mathcal{G}}$ as by our assumption r is a region function maximal below p, since $R^\star$ is α-maximal, r' is maximal below p'.

2) $R^\star$ is closed in the subgame $\mathcal{G}_{r \geq p}$. Since $R^\star$ is open in $\mathcal{G}$, there must be some region(s) of priority $p' < p$ such that we have a vertex $v \in$

$\texttt{vert}(\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star))$ such that there exists $w \in vE$ such that $\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star)(v) \cap \theta(v,w) \cap r^{-1}(p')(w) \neq \emptyset$. Furthermore, the priority p' of one such a region has parity α, as otherwise we have a region in r that is not $\bar{\alpha}$-maximal, as the vertex v and configuration(s) $\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star)(v) \cap \theta(v,w) \cap r^{-1}(p')(w)$ would belong to that region of player $\bar{\alpha}$. By *region merging* $R^{\star'} = R^\star \cup \lambda v \in V.\{c \in \mathfrak{C} \mid r(v)(c) = p'\}$ is also an α-maximal region. However, this contradicts our assumption that s is a smallest element, as we have $s' = (r' = r^{\underline{0}} \uplus r^{\leq p'}, p')$ and there exists $q \leq p'$ such that $r'^{<q} = r^{<q}$ and $r^{-1}(q) \subset r'^{-1}(q)$, and therefore $s' \prec_{\mathcal{G}} s$.

Inductive step Let $s = (r,p)$ be an arbitrary state in $S_{\mathcal{G}}$ such that $\exists s' \in S_{\mathcal{G}} : s' \prec_{\mathcal{G}} s$. As our induction hypothesis, assume that $\texttt{searchDominionVPG}(\mathcal{G}, r', p')$ returns an α-maximal dominion for all $s' = (r', p') \prec s = (r,p)$. From lines 2 through 4 we have that $R^\star = \alpha\text{-}\texttt{FAttr}(\mathcal{G}^{\geq p}_r, r^{-1}(p))$. We now have the following three cases:

$\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}}(R^\star) = \emptyset$ As $R^\star$ is a closed α-maximal region, it is also an α-maximal dominion. Hence $\texttt{searchDominionVPG}(\mathcal{G}, r, p)$ returns an α-maximal dominion in $\mathcal{G}$.

$\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}^{\geq p}_r}(R^\star) \neq \emptyset$ $R^\star$ is open in the subgame $\mathcal{G}^{\geq p}_r$. On line 3 we assign $R = r^{-1}(p)$, hence we have that $r^{-1}(p) \subseteq R^\star$ since $R \subseteq R^\star$. We can distinguish two possible cases:

$r^{-1}(p) \subset R^\star$ Let $q = p$, we have $r^* = r[R^\star \mapsto p]$ and $r^{*^{<q}} = r^{<q}$ and $r^{*^{-1}}(p) = r^{-1}(p) \cup R^\star$ and therefore $r^{-1}(q) \subset r^{*^{-1}}(q)$ satisfying Definition 6.4.1.3(a). As r is maximal below p, and $r^{*^{-1}}(p) = R^\star$ which is α-maximal, r^* is maximal below p^*, from Algorithm 7 line 11, with $(r^*, p^*) \prec (r,p)$.

$r^{-1}(p) = R^\star$ We have $r^* = r[R^\star \mapsto p] = r$. Furthermore, we have that $p^* > p$, thus satisfying Definition 6.4.1.4(b) . Again, as r is maximal below p, and $r^{*^{-1}}(p) = r^{-1}(p) = R^\star$, which is α-maximal, r^* is maximal below p^*, with $p^\star$ from Algorithm 7, line 11 and we have $(r^*, p^*) \prec (r,p)$.

In both cases above, by our induction hypothesis $\texttt{searchDominionVPG}(\mathcal{G}, r^*, p^*)$ returns an α-maximal dominion in game $\mathcal{G}$, therefore $\texttt{searchDominionVPG}(\mathcal{G}, r, p)$ also returns an α-maximal dominion in $\mathcal{G}$ by Proposition 4.

$\texttt{esc}^{\bar{\alpha}}_{\mathcal{G}^{\geq p}_r}(R^\star) = \emptyset$ $R^\star$ is closed in the subgame $\mathcal{G}^{\geq p}_r$. From the definition of the $\texttt{bep}$ and the fact that $R^\star$ is closed in the subgame, it follows that $p^* < p$, with $p^\star$ from Algorithm 7 line 13. Furthermore we have that $r^{*^{-1}}(p^*) = r^{-1}(p^*) \cup R^\star$ according to line 14 in Algorithm 7. Clearly we have $r^{-1}(p^*) \subset r^{*^{-1}}(p^*)$ (as $R^\star \neq \emptyset$) and $p^* \in \texttt{rng}(r^\star)$.

As r is α-maximal below p, clearly it is also α-maximal below p^*. Furthermore, note that $R^\star$ is also closed in $\mathcal{G}^{\geq p^*}_r \setminus r^{-1}(p^*)$ hence by Proposition 4 (Region Merging) $r^{-1}(p^*) \cup R^\star$ is also an α-maximal region in $\mathcal{G}^{\geq p^*}_r$, r^* is α-maximal below p^* and by validity of the original region function r^* is also a valid region function.

We have $(r^*, p^*) \prec (r,p)$ and therefore by our induction hypothesis we have that $\texttt{searchDominionVPG}(\mathcal{G}, r^*, p^*)$ returns an α-maximal dominion, and

therefore `searchDominionVPG`$(\mathcal{G}, r, p)$ returns an α-maximal dominion in $\mathcal{G}$.

As we have shown our hypothesis holds for both the base case and inductive step, we can conclude by induction on $\mathcal{S}$ that `searchDominionVPG`$(\mathcal{G}, r, p)$ returns an α-maximal dominion in $\mathcal{G}$. □

6.5 Conclusions

In this section we have introduced the original priority promotion algorithm and have shown how to adapt it to solve variability parity games and proved its correctness. We described how to optimize determining whether the escape set of an α-maximal region is open or closed and a different approach of searching for closed dominions in a variability parity game using an early termination approach. In the next section we will introduce self-loop elimination - a well-known pre-processing steps for parity games - and adapt it to the VPG setting.

Chapter 7

Self-loop elimination for Variability Parity Games

Pre-processing is a step which is often performed before solving parity games [14, 21, 8]. When pre-processing a game, we transform the problem without affecting the solution, often with the aim of decreasing the complexity of the problem. One common pre-processing step is self-loop elimination. By removing parts of the parity game before solving it, we can potentially speed up the computation time. Depending on the parity and owner of a vertex with a self-loop, we can in some cases disregard the loop entirely, or use the loop to solve part of the parity game.

First, let v be an arbitrary vertex with a self-edge $(v, v) \in E$ with even priority and owner. Therefore, we have a trivial strategy for player even to win this vertex: always stay in this vertex by taking the self-loop. Because the lowest priority of the resulting path occuring infinitely often will be even, the player has a winning strategy for this vertex. Furthermore, note that the set of vertices with such a self-loop is also a dominion for player even, since any play that enters one of the vertices will stay in the dominion indefinitely, and is won by player even. All vertices in the even attractor set of the dominion are won by player even, since player even can force the play to enter the dominion.

Alternatively, the owner of the vertex v is player odd. If the vertex has any outgoing edges we can remove the self-edge, as taking the self-edge would be a losing strategy for player odd. In both cases we can narrow down the edges which will be in the strategy for player even or odd, allowing us to remove edges which will not be taken by either player.

In case of VPGs, we will also need to take into account the edge guards while eliminating self-loops. In Figure 7.1 we again have two vertices with self-loops. We will now describe how to perform self-loop elimination on VPGs.

(a) Vertex with a self-edge and even priority and owner.

(b) Vertex with a self-edge and even priority owned by player odd.

Figure 7.1: Self-loop elimination for a VPG, with $\mathfrak{C} = \{c_1\}$.

Let $\mathcal{G}$ be a parity game that is total w.r.t restriction ϱ. Again, in case the owner and priority of the vertex are the same, as in Figure 7.1a, the owner of the vertex $v \in V$ with a self-edge $(v, v) \in E$ and $\theta((v, v)) = \{c_2, c_3\}$ will have a strategy for winning the vertex for configurations $\{c_2, c_3\}$ (taking the self-edge). We can compute the featured attractor for player even to the set of all vertices with such a self-edge, which are also won by player even. Let A^λ be this featured attractor, we can remove it from the game as it is won by player even. We can then continue solving the subgame $\mathcal{G}$ restricted to $\varrho \setminus A^\lambda$.

When the owner and the priority are not the same, we can still solve parts of the game. Since taking the self-edge will never result in a winning strategy for player odd, player even can only win from the vertex if the odd player is forced to take the self-edge. Therefore, we can compute the configurations that are winning for player even for the vertex as follows:

$$\mathfrak{w}(v) = \theta(v, v) \setminus \sum_{w \in vE \wedge w \neq v} \theta(v, w)$$

If we take all vertices $v \in V$ with such a self-edge and their winning configurations $\mathfrak{w}(v)$, we can compute their featured attractor, A^λ, which is won by player even. We again solve the subgame $\mathcal{G}$ restricted to $\varrho \setminus A^\lambda$.

The same set of elimination rules can be applied to vertices with odd priority.

In the next section, we will perform experiments on all the algorithms proposed up until now, comparing their computation time when solving parity games.

Chapter 8

Experiments

We will perform experiments by comparing the solving times of the implementations of the different algorithms as well as compare other metrics to gain more insight into the performance of our implemented algorithms. In this chapter we will describe in more detail our implementation and experimental setup in Section 8.1, our test cases we used to run our different algorithms on and describe the methods we used to generate our own random VPGs in Section 8.2, we present our results in Section 8.4 and lastly the discussion in Section 8.5.

8.1 Implementation

The algorithms and pre-processing method from Chapters 4,5,6 and 7 have been implemented in C++14. The edge guards of VPGs are represented using the BDD library BuDDy[1]. In the SPL setting the different products are the collection of features that are enabled for that configuration or product. Since we are interested in solving SPL model checking problems, we will also represent the configurations in our VPG as a collection of features, which can usually be represented efficiently with *Binary Decision Diagrams* (BDDs). The same code for parsing and printing solutions of games is used as in [20], as well as the underlying representation of variability parity games, to ensure that the different algorithms can be compared fairly against the existing algorithms.

All of the implemented algorithms and scripts to perform the experiments are available on GitHub[2]).

The algorithms implemented are:

- Priority Promotion (with and without early termination),
- Small Progress Measures,
- Zielonka's algorithm with tight SCC decomposition

[1] `https://sourceforge.net/projects/buddy`
[2] `https://github.com/Dodecahedra/VPGSolvers`

Recall that the SCC Decomposition on VPGs only computes the SCCs without taking into account the configurations, as described earlier in Chapter 4.

To ensure our implementations are correct, we compared the output of our algorithms against the output of Zielonka's algorithm for VPGs, as well as verified our solution by comparing each projection against the output of the corresponding projected game using Oink[3] [10].

In the remainder of this chapter we will discuss the test cases we used when performing our experiments, describe the results from our experiments and lastly discuss the results. In Section 8.4.1 we compare the performance of the different algorithms on the games generated in Section 8.2.1.

8.2 Test cases

We compare the aforementioned algorithms on the same set of VPGs as used in [20]. Sadly there are only a limited amount of VPGs available from an SPL setting. To be able to compare our algorithms on a rigorous set of VPGs, we also generated our own VPGs by adapting parity games generated using `PGSolver`[4] [14].

8.2.1 Random Variability Parity Games

In order to create VPGs, we first generate parity games using the `PGSolver` tool, which can generate three different classes:

Randomgames Randomgames are generated satisfying the given number of nodes, maximum priority and a lower and upper bound on the number of outgoing edges for each vertex. These games are usually the simplest.

Steadygames Similar to randomgames but tries to circumvent often used optimisations, such as priority compression, and self-loop elimination and is better suited to test the performance of the algorithm solving the game instead of any pre-processing steps.

Clusteredgames Usually the randomgames and steadygames that are generated will consist of one big SCC, therefore not being a good game to be solved by the SCC decomposition algorithm. *Clusteredgames* are generated to contain more SCCs and will be a better benchmark specifically for algorithms using SCC decomposition.

[3] `https://github.com/trolando/oink`
[4] `https://github.com/tcsprojects/pgsolver`

When verifying products in an SPL, a *product* in an SPL consists of a set of *features* which are enabled for that specific product. In VPGs such a product can be seen as one *configuration* for which we play the VPG. Therefore to model a product from an SPL a *configuration* consists of a set of *features*, where $\mathcal{F}$ is the set of all our features in the SPL. We will now describe how we can generate VPGs consisting of such sets.

Given a parity game, a number m describing the amount of features in $\mathcal{F}$ and a *threshold* $\psi \in (0, 1]$, we can iteratively generate a VPG. Let $l = \frac{k}{|E| \cdot m}$ where k the sum of the number of enabled - or disabled - features for all configurations that we have added to our game. Hence before we added any configurations it holds that $k = 0$.

The next step is to generate configurations to add to our parity game. First, let p be a random integer in $[0, ..., m]$. Next, we randomly pick p features from the set $\mathcal{F}$, where with probability $\frac{1}{2}$ we take the negation of a feature $f \in \mathcal{F}$. This set of features becomes our configuration we can add to our game.

Our last step is to pick an edge to add a configuration to. We will use two approaches: (uniform) pick an edge using a uniform distribution, such that each edge is equally likely to be picked, or (grouped) we increase the chance of picking an edge after its source vertex has been picked before. This second approach will generate VPGs where the edge guards are more "clustered" together, which might mean the games will be able to exploit the greater commonality between different products. Note that when we pick an edge that already has a configuration, we take the disjunction of both configurations.

We repeat this process of generating configurations until $\frac{k}{|E| \cdot m} \geq \psi$.

In the next section we will describe our experimental setup and how we gathered the results.

8.3 Benchmark

For each algorithm we keep a set of metrics to evaluate their performance, the metrics we keep for each algorithm can be seen in Table 8.1. Performing all the experiments for the Small Progress Measures algorithm was not possible due to the long solving times of the SPM algorithm, which was longer than our 10 minute timeout. We use a script to run each algorithm on the set of games in our dataset and collect the metrics in a csv file. In the next section we will describe our results.

In Section 8.4.2 we discuss the lower performance of the SPM algorithms compared to the other algorithms. In Section 8.4.3 we compare the performance of the different algorithms (excluding SPM) on the original dataset from [20]. Lastly, we compare

Priority Promotion	Progress Measures	SCC Decomposition	Zielonka
# Attractions	# Lifts	# Attractions	# Attractions
Attractor time (ns)	# Upgrades	# SCC Decompositions	# Attractors
Escape-set time (ns)	-	Decomposition time (ns)	Attractor time (ns)
# Promotions	-	Attractor time (ns)	# Recursions

Table 8.1: Collected metrics for each algorithm.

the algorithms in some special cases in Sections 8.4.4 and 8.4.5, where consider clusteredgames and games increasing in size respectively.

8.4 Results

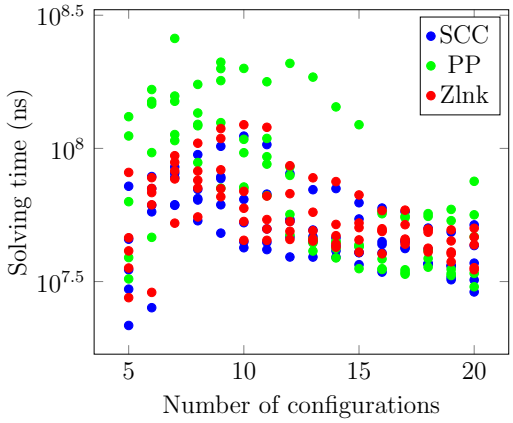
We will now describe the results of our experiments. For each dataset that we ran our algorithms on we will describe the results we gathered. The raw data from the results is available through Zenodo[5][9]. We use the following abbreviations for the different type of algorithms: (Zlnk) Zielonka's recursive algorithm, (PP) Priority promotion (without early termination), (ET) Priority promotion with early termination, (SPM) Small Progress Measures and (SCC) Zielonka's algorithm with tight SCC integration.

8.4.1 Game generation

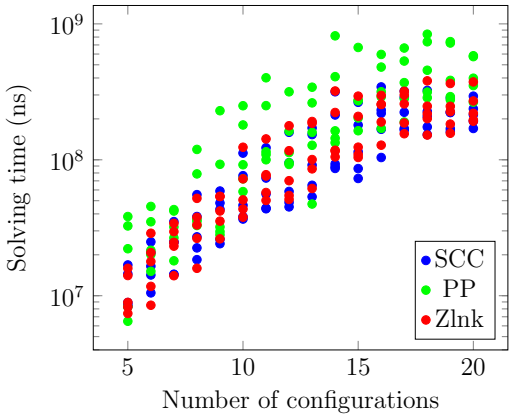
In Section 8.2.1 we described two different approaches for generating random VPGs from parity games. In Figure 8.1 we can see the solving times for uniform and grouped random VPGs, where each data point is the solving time of one individual game. In Figure 8.1a we can see that the solving time does not increase as the number of configurations gets larger, whereas we can see an exponential increase in Figure 8.1b. In Figure 8.1c we can see the solving times for all random games and in Figure 8.1d only games with 50 vertices. As we can see, for smaller games the solving times do not really increase depending on the configurations, this is likely because the games are too small for more than 10 configurations. For this reason we will only consider games with more than 50 vertices.

Also note that in Figure 8.1a games with 5 configurations already take one magnitude longer to solve than the grouped random games. We think this is because the uniform games will have a random distribution of edges with guards, while the grouped random games will have the guards more localised in parts of the graph, which the algorithms can exploit using the family-based approach.

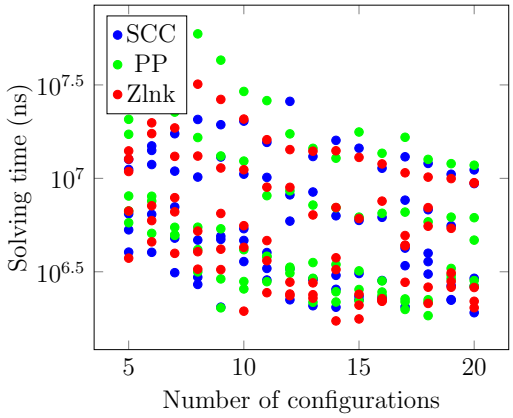
[5] `https://doi.org/10.5281/zenodo.5637419`



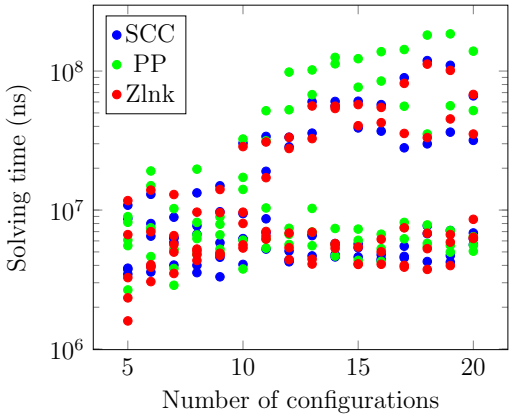
(a) Increasing configurations for games with 100 vertices of unified random games.



(b) Increasing configurations for games with 100 vertices of grouped random games.



(c) Increasing configurations for games with 50 vertices of unified random games.



(d) Increasing configurations for games with 50 vertices of grouped random games.

Figure 8.1: Solving times, in logarithmic scale, of unified and clusered random games against the number of configurations. Each point in the figures is the solving time of one game.

This might explain why the solving times of the uniform games do not increase exponentially, as is the case with grouped random games. Since the uniform games will have little "commonality" early on, adding more configurations will not cause a big increase of complexity.

As the grouped random games are more in-line with VPGs that we expect from problems from an SPL setting: high commonality and exponential increase with more configurations, we will use these games in the rest of our experiments.

8.4.2 Small Progress Measures

In the previous section we did not include the small progress measures algorithm when comparing the different type of games. In the next sections we will also mostly be excluding the small progress measures algorithm, as it is vastly outperformed by the other algorithms and will often take longer than 10 minutes when solving games with more than 50 vertices. In Figure 8.2 we can see the SPM algorithm for 50 vertices compared to the other algorithms.

Game	1	2	3	4	5
# Upgrades	31003	156239	6052	39905	824687
# Lifts	203167	769293	19821	140753	3659826
% Upgrades	15.3	20.4	31.5	27.8	20.7

Table 8.2: Metrics of the SPM algorithm on games with 50 vertices, where the number of lifts is the number of "Lift" operations that we performed, the number of upgrades is the amount of successful lifts and the % Upgrades is the percentage of succesful Lift operations out of the total Lift operations.

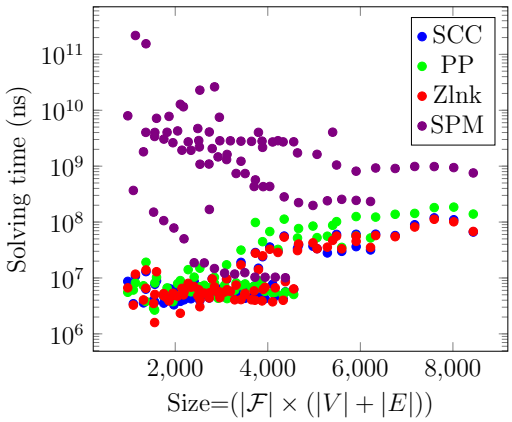


Figure 8.2: Solving times for grouped random games with 50 vertices, including the SPM algorithm. With $|\mathcal{F}|$,$|V|$ and $|E|$ we denote the sizes of the set of features, vertices and edges respectively.

As we can see from Figure 8.2 the solving times for random games with 50 vertices is in all cases slower than the SCC, priority promotion and recursive algorithm, in most cases by multiple orders of magnitude. For games with 100 vertices none of the games were solved within 10 minutes.

In Table 8.2 we can see some metrics of the SPM algorithm for five different smaller games.

8.4.3 Original dataset

In this section we will give an overview of the performance of the new algorithms on the dataset from [20]. In Table 8.3 we give the average solving time, attractor time and number of attractors.

An interesting thing we observe is that, compared to the randomly generated games, on average all the algorithms spend a lot less time on the attractor computation for the minepump games (roughly 35% instead of 80%). We also note that although the SCC decomposition algorithm performs the most attractor computations, the total attractor time is in almost all cases (except for the minepump games) the lowest for the SCC algorithm. This makes sense, as we first decompose the VPG into smaller connected components before calculating the attractor set in each. Lastly we observe that Zielonka's recursive algorithm has the best average performance over the whole dataset.

	Average solving time (ns)	Attractor time (ns)	% Attractor	# Attractors
PP	$3.09 \cdot 10^8$	$3.06 \cdot 10^8$	98.82	13.96
SCC	$2.85 \cdot 10^8$	$2.65 \cdot 10^8$	92.71	34.76
Zlnk	$2.71 \cdot 10^8$	$2.71 \cdot 10^8$	99.77	20.88

(a) Average solving time for random games of original dataset.

	Average solving time (ns)	Attractor time (ns)	% Attractor	# Attractors
PP	$5.68 \cdot 10^6$	$5.12 \cdot 10^6$	90.10	8.5
SCC	$5.77 \cdot 10^6$	$4.61 \cdot 10^6$	79.87	21.36
Zlnk	$4.99 \cdot 10^6$	$4.74 \cdot 10^6$	94.83	7.54

(b) Average solving time for random scaled games of original dataset.

	Average solving time (ns)	Attractor time (ns)	% Attractor	# Attractors
PP	$1.73 \cdot 10^6$	$1.48 \cdot 10^6$	85.73	11.70
SCC	$1.55 \cdot 10^6$	$1.11 \cdot 10^6$	71.96	26.92
Zlnk	$1.31 \cdot 10^6$	$1.16 \cdot 10^6$	88.38	17.98

(c) Average solving time for random verification games of original dataset.

	Average solving time (ns)	Attractor time (ns)	% Attractor	# Attractors
PP	$9.00 \cdot 10^6$	$2.21 \cdot 10^6$	30.60	4.33
SCC	$1.53 \cdot 10^7$	$5.46 \cdot 10^6$	35.76	20.33
Zlnk	$3.36 \cdot 10^6$	$1.52 \cdot 10^6$	47.05	2.00

(d) Average solving time for minepump games of original dataset.

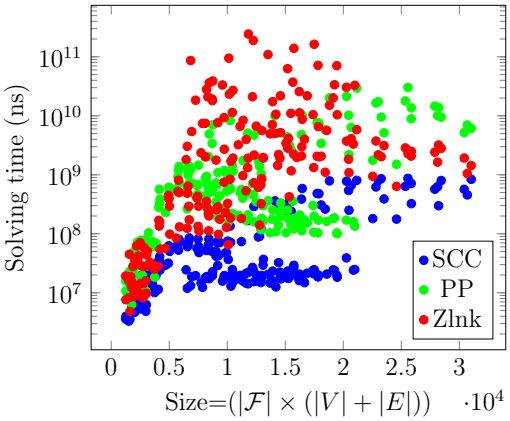
	Average solving time (ns)	Attractor time (ns)	% Attractor	# Attractors
PP	$3.95 \cdot 10^9$	$2.69 \cdot 10^9$	68.10	5.00
SCC	$1.53 \cdot 10^{11}$	$1.32 \cdot 10^{10}$	8.63	$2.69 \cdot 10^3$
Zlnk	$2.13 \cdot 10^9$	$1.80 \cdot 10^9$	84.51	2.00

(e) Average solving time for elevator games of original dataset.

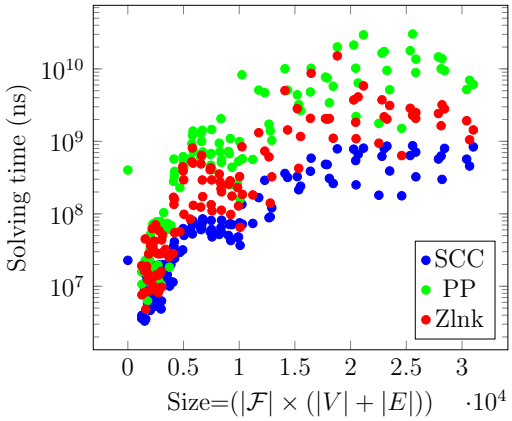
Table 8.3: Performance and metrics of the priority promotion, SCC and Zielonka recursive algorithm on the dataset from [20]. The average solving times are the average of the random (75), scaled random (55), random verification (203), minepump (9) and elevator (7) games respectively.

8.4.4 Clusteredgames

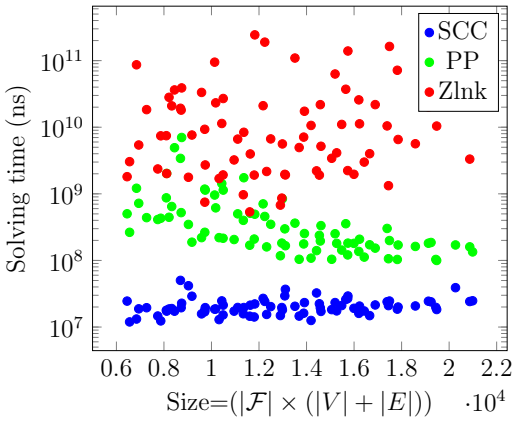
One interesting class of games to consider is the "clustered games", which are generated to contain a large amount of strongly connected components. Although these games are not likely to be encountered when solving verification problems, they do provide for an interesting comparison between the SCC decomposition algorithm and the others. We generated clusteredgames using `PGSolver` and generated VPGs of using the grouped approach as described in Section 8.2.1. In Figure 8.3 we can see the solving times for the different algorithms.



(a) Solving time for all clustered games depending the size of the game.



(b) Solving time for games with less than 400 vertices, depending on the size of the game.



(c) Solving times for games greater than 400 vertices, depending on the size of the game.

Figure 8.3: Solving times for the grouped clusteredgames. With $|\mathcal{F}|$,$|V|$ and $|E|$ we denote the sizes of the set of features, vertices and edges respectively.

In Figure 8.3a we can see the solving times depending on the size of the game for the entire set of clustered games. Upon inspection of the solving times of the games, we notice that for games with more than 400 vertices the SCC decomposition algorithm is multiple magnitudes faster than the other algorithms. In Figures 8.3b and 8.3c we can see the solving times for the games with less than 400 and more than 400 vertices. Interesting to note is that for the smaller games in Figure 8.3b the solving times are similar to those of the other class of games, where Zielonka's and the SCC algorithm perform similarly and the priority promotion algorithm performs slightly worse. However, for the larger games the SCC decomposition algorithm performs several order of magnitude better than both priority promotion and Zielonka's algorithm. This might be because the games are large enough to contain multiple smaller SCCs, giving the SCC decomposition algorithm an advantage, as it only has to solve the smaller SCCs.

Vertices	Attractor calls	Tarjan calls	% Attractor	% Tarjan
≥ 400	113.1	1538.46	55	11
< 400	265.41	1027.99	92	6

Table 8.4: Average number of attractor set calculations, SCC decomposition using Tarjan's algorithm and average percentage of time spent on each for different clustered games.

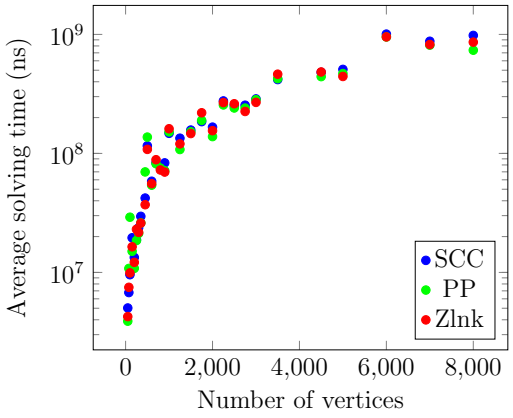
In Table 8.4 we can see metrics of the SCC decomposition algorithm for games with more than 400 vertices and with less than 400 vertices. As we can see from Table 8.4, for the larger games we on average have fewer attractor set calculations (113) and spend less time on the attractor set calculation (55%) compared to the smaller games (265 attractor sets and 92%). However, for the larger games we spend more time on computing the SCC decompositions (11% against 6% for smaller games), which might explain the huge decrease in solving times compared to Zielonka's recursive algorithm and the priority promotion algorithm.

8.4.5 VPGs increasing in size

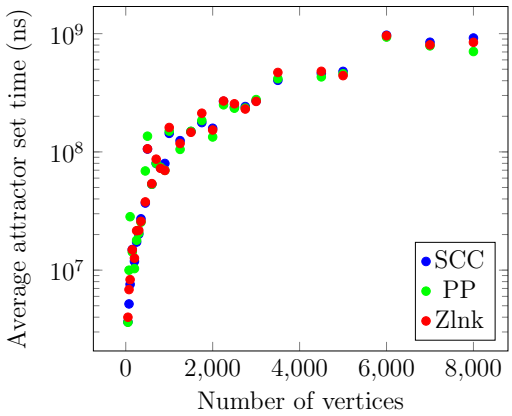
In previous sections we looked into different classes of games and compared the performance of the different algorithms depending on their size or number of configurations. Another metric that is interesting to look at is the effect of the number of vertices on the solving times. In Figure 8.4 we can see the solving times, and attractor set metrics, for VPGs increasing in size.

	Average solving time (ns)	% Attracting	% Escape-set	% Tarjan
PP	$2.15 \cdot 10^8$	97.50	0.37	-
SCC	$2.31 \cdot 10^8$	95.11	-	4.21
Zlnk	$2.21 \cdot 10^8$	99.24	-	-

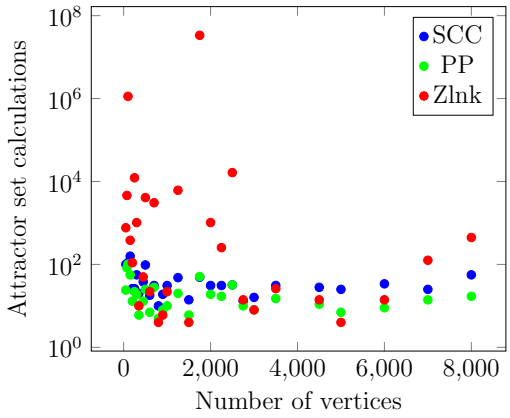
Table 8.5: Average solving times, attractor set calculaction, escape-set calculation and SCC decomposition times of games increasing in vertices.



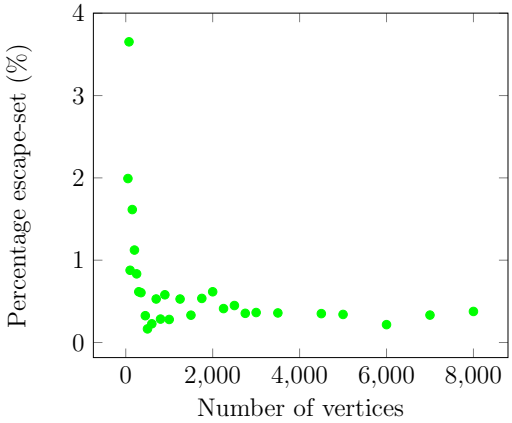
(a) Average solving time in nanoseconds, for VPGs with 5 features and an increasing number of vertices.



(b) Average time spent on attractor set calculation for VPGs with 5 features and an increasing number of vertices.



(c) Number of attractor set calculations for VPGs with 5 features and an increasing number of vertices.



(d) Percentage escape-set calculation of the total solving time for the priority promotion algorithm and games increasing in size.

Figure 8.4: Solving times, attracting time and number of attractor set calculations under VPGs with increasing number of vertices.

In Figure 8.4a and 8.4b we can see the average solving and attracting times. Based on the theoretical time complexity, we would expect the solving time to grow polynomially as the number of vertices of the games increases. We can confirm that the solving time and attractor set time increase sub-exponentially as the games increase in the

number of vertices. We also note that both figures are almost identical, suggesting that the solving time is dominated by the attractor set calculation, this can also be seen in Table 8.5.

Table 8.5 shows the average solving time and the percentage of time spent on the attractor-set, escape-set or Tarjan's calculation for the appropiate algorithm. We note that the average solving times are very close together and that there is no clear "winner".

In Figure 8.4d we can see the percentage of the escape-set calculation as the number of vertices of the game increases. As the game increases in size the percentage of the escape-set calculation seems to converge to around 0.3%, together with Table 8.5 this seems to indicate that most of the solving time is spent on the attractor set (for all the algorithms).

In Figure 8.4c we can see the number of attractor set calculations. Unlike the average attractor and solving time, the number of attractor calculations does not seem to increase linearly as the number of vertices of the game increases.

8.4.6 Self-loop elimination and early termination

In Tables 8.6 and 8.7 we can see the solving times of the algorithms with self-loop elimination, compared to the solving times without first eliminating the self-loops on the VPGs from the original dataset and randomly generated grouped games.

Solving time	After elimination	Including elimination
PP	-5.53%	28.13%
SCC	-4.70%	40.34%
Zlnk	-5.23%	39.38%

Table 8.6: Average solving time increase on random games using self-loop elimination compared to the average solving times without performing self-loop elimination.

Solving time	After elimination	Including elimination
PP	-6.13%	44.04%
SCC	-8.81%	42.24%
Zlnk	-9.81%	54.67%

Table 8.7: Average solving time increase on minepump and random verification games from the original dataset using self-loop elimination compared to the average solving times without performing self-loop elimination.

As we can see from Table 8.6 and 8.7 the solving time after eliminating the self-loops

of a VPG decreases by roughly 5% to 10%. However, if we include the time spent on solving these self-loops we see that the overall solving times actually increase on both random and verification games. We propose this is because removing the self-loops is a computationally expensive operation, since we have to update the neighboring edges after removing (part of) the self-loop.

In Section 6.3.2 we discussed another method for determining whether a region is open or closed by removing any configurations from which the opponent can leave the region. We compare the priority promotion algorithm against priority promotion with early termination on the random games and the dataset from [20].

	Solving time (ns)	Attractor %	Escape-set %	Attractions
Priority Promotion	$6.47 \cdot 10^7$	98.5	0.24	11.7
Early termination	$2.15 \cdot 10^8$	99.5	0.16	55.7

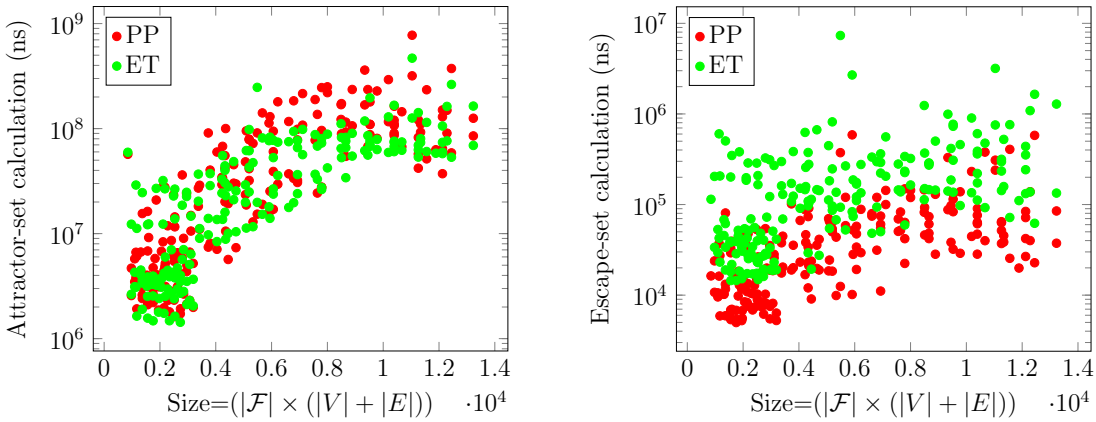
Table 8.8: Average solving time increase and attractor and escape-set calculation on random verification games of the original dataset.

	Solving time (ns)	Attractor %	Escape-set %	Attractions
Priority Promotion	$9.00 \cdot 10^6$	30.60	13.74	4.33
Early termination	$1.86 \cdot 10^9$	99.55	0.11	$5.21 \cdot 10^3$

Table 8.9: Average solving time increase and attractor and escape-set calculation on minepump games of the original dataset.

In Table 8.8 we list the average solving times, attractor and escape-set times for random verification games from the original dataset. Quite clearly priority promotion without early termination performs better than priority promotion with early termination. Even though we spend relatively less time on computing the escape set, the overall solving time increases. This is probably due to the amount of attractor sets that we have to compute in total is more for the early termination variant, causing the solving time to increase. Looking at the average amount of attractor set calculations for all the games, we can verify that this is indeed the case, with the average for early termination being roughly 5 times higher than the original algorithm. Looking at the number of attractions for the minepump games in Table 8.9 we can see a vast difference between priority promotion and early termination, where early termination uses more than 5000 attractor set calculations, compared to the 4 by priority promotion.

When looking at the random games however, we find opposite results, which we can see in Table 8.10. In Figure 8.5 we can see the attractor and escape-set calculation as the games increase in size. As we would expect, we spend more time on the escape-set calculation. In terms of attractor set calculation, there is no significant difference between both algorithms. Lastly, note that early termination is significantly slower for the verification games, but decreases the solving times for random games.



(a) Attractor set calculation times of the priority promotion and priority promotion with early termination algorithms.

(b) Escape-set calculation times of the priority promotion and early termination algorithm.

Figure 8.5: Attractor-set times and escape-set times for the priority promotion and priority promotion with early termination.

	Solving time (ns)	Attractor %	Escape-set %
Priority Promotion	$6.43 \cdot 10^7$	97.37	0.086
Early termination	$4.36 \cdot 10^7$	96.62	0.62

Table 8.10: Average solving time and attractor and escape-set calculation times for random games.

8.5 Discussion

We will now discuss the performance of the algorithms on the different datasets. We will discuss the results we have found in Section 8.4 in the same order as introduced there. Afterwards we will discuss the threats to validity to our experiments.

In Section 8.4.1 we described the solving times of the different algorithms as the number of configurations increases for uniform and grouped random games. For the remainder of the experiments we only used grouped random games, as the games generated using this method are different from a lot of the random games already present in the existing datasets.

We notice that the small progress measures algorithm has the worst performance of all the algorithms. This is not surprising as the original algorithm for parity games is also outperformed by the recursive algorithm and priority promotion for parity games. In fact, for most of the games which are easily solved by the recursive algorithm and priority promotion, the SPM algorithm will take longer than 10 minutes, and due to

this we ran the progress measures algorithm only on smaller games of each dataset. We think this poor performance is probably due to two factors: (1) the algorithm often has to compute the new progress measure of a vertex, which uses a lot of BDD operations, which can be very expensive, (2) a lot of the progress measures that are computed are the same as the old value, hence adding to the running time.

Next we compared the different algorithms on the dataset from [20]. Like the random games, Zielonka's recursive algorithm performs the best on average for almost all the different games in the dataset. However, one interesting observation is that in case of the minepump games, which are derived from an SPL model checking problem, very little time of the solving time is spent on the attractor set computation compared to the randomly generated games. This is an indication that the randomly generated games might not represent VPGs that we would encounter in a model checking setting. Considering the different type of randomly generated games and random games from the original dataset, we think this is due to the structure and placement of the configurations rather than the distribution of the configurations.

Next, we looked at the performance of clustered games, which are randomly generated VPGs consisting of multiple clusters of strongly connected components. As we would expect, the recursive algorithm with SCC decomposition performs the best out of all the algorithms. We also found that for games larger than 400 vertices the SCC decomposition algorithm outperforms the other algorithms by multiple orders of magnitude. Looking at the internal metrics of the SCC algorithm, we found that significantly more time is spent on the SCC decomposition using Tarjan's algorithm than for the smaller games. This is most likely why we see this large increase in performance, as the "heavy lifting" is done by the SCC decomposition instead of the attractor set calculation.

Contrary to what we saw when increasing the number of configurations, we do not find an exponential increase in solving time as the number of vertices increase in Section 8.4.5. From Figure 8.4 and Table 8.5 we can see that the solving times for all the algorithms are dominated by the attractor set calculation. Like the SCC decomposition, calculating the escape-set does not increase linearly, such as the solving time and attractor time. We can conclude that all three algorithms are dominated by the attractor-set calculation.

Lastly, we looked at the performance of the different algorithms after self-loop elimination and with early termination, as described in Sections 8.4.6. We notice that the solving time after we eliminated the self-loops decreases, since we are able to remove parts of the VPG before running one of the algorithms. However, if we include the time it took to eliminate these self-loops, we actually see an increase in the solving time for all the algorithms. This means that it takes more time to compute the set of configurations for which we can eliminate the self-loop than solving the game without self-loop elimination.

When running the priority promotion algorithm with early termination, we see a significant increase in the solving times compared to original priority promotion when solving minepump and other SPL-based VPGs. However, when running on random games, early termination seems to perform marginally better.

8.5.1 Threaths to validity

We will now discuss some of the threats to the validity of our experiments. We will discuss the optimization and correctness of the algorithms, and the representativeness of our dataset.

When implementing the different algorithms, we made sure to validate the results with the output from Zielonka's recursive algorithm to ensure our output is correct. We also checked the results against an existing parity game checker, Oink [10]. This way we can be fairly certain that our implementation and algorithms are correct.

Another factor that may invalidate our findings is the optimization of the implemented algorithms. If certain operations are not optimized, it might negatively impact the running time of one of the algorithms. In case of the priority promotion and SCC decomposition algorithm we are fairly sure they are optimized, as we found that the attractor-set calculation is almost always dominating the solving time. In case of the small progress measures algorithm we can not make this observation, as it does not depend on the attractor calculation. However, our finding that it performs the worst among all implemented algorithms in practice is also found for the parity game variant of the small progress measures algorithm, indicating that it might be due to the algorithm itself rather than being unoptimized.

Lastly, we generated new games to expand our dataset we can compare our algorithms on. Though our dataset might give a good insight on how the algorithms perform in general, we also noted that a lot of the generated VPGs don't share the same characteristics compared to the minepump dataset. This means that our dataset might not be very representative of the type of games we might encounter in a model checking setting.

Chapter 9

Conclusions

Although variability parity games are a novel concept, a few algorithms to solve VPGs exist. Being a generalization of parity games, a lot of the definitions and concepts used to solve parity games can be adapted to apply to VPGs as well.

We adapted the concepts of regions, dominions and escape-sets used in priority promotion for parity games to be applicable to VPGs and introduced an algorithm to solve VPGs using priority promotion. We solve VPGs by iteratively constructing the maximal region, consisting of a set of vertices and configurations, until we find a dominion, which is a subgame won by one player. We have shown that this algorithm is correct in solving VPGs as well as provided an implementation.

Next we briefly introduced the original small progress measures algorithm for parity games and provide implementation details on an adaptation of the small progress measures algorithm for VPGs.

We introduced the notion of SCC-families, which lifts the concept of SCCs to VPGs, defining for each SCC the largest set of vertices and configurations such that it is still a valid SCC in the projections of all the configurations. Next we introduced an adaptation of Zielonka's recursive algorithm that solves VPGs by decomposing the VPG into its strongly connected components.

Next we gave an adaptation of self-loop elimination, a pre-processing step often used when solving parity games, applicable to VPGs. By taking into account the set of configurations for which the player is forced to, or able to, take the self-loop, we can solve part of the VPG before running one of the algorithms on the remaining game.

We provided methods to generate VPGs from parity games by iteratively adding edge guards to the game, with either a random or "clustered" approach. We looked into some of the properties of the resulting games and found that the clustered approach generates games that increase exponentially as we add more configurations. We used these games when comparing the different algorithms. Using this approach we generated multiple classes of VPGs, such as: clustered VPGs and random VPGs.

Next we used our newly generated VPGs and existing datasets to compare the performance of Zielonka's recursive algorithm, the priority promotion algorithm, small progress measures and the recursive algorithm with SCC decomposition. We found that Zielonka's recursive algorithm performs best out of the set algorithms in general,

with the priority promotion and SCC decomposition algorithms performing slightly worse. The small progress measures algorithm performed worst out of all the algorithms, in most cases taking longer than the timeout we set for the computation time. This poor performance is also seen in the original small progress measures algorithm.

When comparing the algoritms on the clustered games, which contain multiple strongly connected components, the SCC decomposition algorithm performs better than the recursive algorithm and priority promotion and works very well on larger games. Looking into the metrics of the SCC algorithm, we found that for smaller games most of the computation time is still spent on computing the attractor set, however for larger games the decomposition takes more time, indicating more of the work is done by the decomposition algorithm.

We looked into the performance of the different algorithms as the number of vertices of the VPGs increased. We found that the solving time and attractor set time increased linearly with the number of vertices. We found that most of the solving time of the algorithm is spent on the attractor set calculation. Optimizing this computation will probably see the most returns for algorithms using the attractor set.

Lastly we looked into the effect of self-loop elimination and early termination on the solving time. We found that removing self-loops from VPGs does decrease solving times, however if we include the time to remove these self-loops we found an increase in the solving times. This is most likely due to computing the set of configurations being quite expensive, causing it to be too expensive most of the time. We found similar results when comparing the original priority promotion algorithm against priority promotion with early termination. In most cases performing early termination does not decrease the solving times compared to the original algorithm.

Future work

In our work we looked into the effect of early termination in priority promotion, where we can stop our search for a dominion early by removing configurations from our region. More optimizations for the original priority promotion algorithm exist [2] and it would be interesting to look into the effect on performance these optimization might have. The original small progress measures algorithm paper also details some optimizations which could be made, such as SCC decomposition and an adapted progress measure [19] which might also be interesting to investigate further.

We introduced the concept of SCC-families, where SCCs are defined on VPGs, although we did not use this concept in the recursive decomposition algorithm. It would be interesting to compute the decomposition taking into account the edge guards of the graph.

When comparing the algorithms using the attractor set calculation, we noticed that most of the solving time is spent on computing the attractor sets. Therefore, it might be interesting to look into different approaches to compute the attractor set for VPGs, or optimize the existing attractor set calculation. One approach might be to first compute the (non-featured) attractor in the game. Similarly to the SCC-approximation, this attractor is an "over-approximation" of the featured attractor. Next we can compute the featured attractor on this smaller set of vertices, avoiding the more expensive featured attractor on the entire game.

We noted that the VPGs generated from the minepump and elevator application do not share the same internal metrics as the randomly generated VPGs. As there are not a lot of VPGs available from an SPL setting, we do not have a good insight into the performance of the algorithms on these types of games. It would be interesting to model more of these SPL problems in the future to be able to have a more reliable comparison.

Bibliography

[1] Maurice H. ter Beek et al. "Family-Based SPL Model Checking Using Parity Games with Variability". In: *Fundamental Approaches to Software Engineering*. Springer International Publishing, 2020, pp. 245–265.

[2] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "Solving parity games via priority promotion". In: *Formal Methods in System Design* 52.2 (2018), pp. 193–226.

[3] Cristian S. Calude et al. "Deciding Parity Games in Quasi-polynomial Time". In: *SIAM Journal on Computing* (Jan. 2020), STOC17–152–STOC17–188.

[4] Cristian S. Calude et al. "Deciding parity games in quasipolynomial time". In: ACM, June 2017.

[5] Andreas Classen et al. "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking". In: *IEEE Transactions on Software Engineering* 39.8 (Aug. 2013), pp. 1069–1089.

[6] Andreas Classen et al. "Model checking software product lines with SNIP". In: *International Journal on Software Tools for Technology Transfer* 14.5 (June 2012), pp. 589–612.

[7] Maxime Cordy et al. "ProVeLines". In: *Proceedings of the 17th International Software Product Line Conference co-located workshops on - SPLC '13 Workshops*. ACM Press, 2013.

[8] Sjoerd Cranen, Jeroen J. A. Keiren, and Tim A. C. Willemse. "Stuttering Mostly Speeds Up Solving Parity Games". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 207–221.

[9] Koen Degeling. *VPGSolver Experiments*. Nov. 2021. DOI: 10.5281/zenodo.5637419.

[10] Tom van Dijk. "Oink: An Implementation and Evaluation of Modern Parity Game Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 291–308. ISBN: 978-3-319-89960-2.

[11] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. "Variability Abstractions: Trading Precision for Speed in Family-Based Analyses". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). 2015, pp. 247–270.

[12] Uli Fahrenberg and Axel Legay. “Featured Games”. In: *International Symposium on Theoretical Aspects of Software Engineering, TASE 2021, Shanghai, China, August 25-27, 2021*. IEEE, Aug. 2021.

[13] John Fearnley et al. “An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space”. In: *International Journal on Software Tools for Technology Transfer* 21.3 (Feb. 2019), pp. 325–349.

[14] Oliver Friedmann and Martin Lange. “Solving Parity Games in Practice”. In: *Automated Technology for Verification and Analysis*. Springer Berlin Heidelberg, 2009, pp. 182–196.

[15] Maciej Gazda and Tim A.C. Willemse. “Zielonka’s Recursive Algorithm: dull, weak and solitaire games and tighter bounds”. In: *Electronic Proceedings in Theoretical Computer Science* 119 (July 2013), pp. 7–20.

[16] Gerard J. Holzmann. *The spin model checker: primer and reference manual*. Addison-Wesley, 2004.

[17] Marcin Jurdzinski and Ranko Lazic. “Succinct progress measures for solving parity games”. In: IEEE, June 2017.

[18] Marcin Jurdziński. “Deciding the winner in parity games is in UP ∩ co-UP”. In: *Information Processing Letters* 68.3 (Nov. 1998), pp. 119–124.

[19] Marcin Jurdziński. “Small Progress Measures for Solving Parity Games”. In: *STACS 2000*. Springer Berlin Heidelberg, 2000, pp. 290–301.

[20] Sjef van Loo. “Verifying SPLs using parity games expressing variability”. MA thesis. Technische Universiteit Eindhoven, 2019.

[21] Sven Schewe. “Solving Parity Games in Big Steps”. In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin Heidelberg, 2007, pp. 449–460.

[22] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2 (June 1972), pp. 146–160.

[23] Wieslaw Zielonka. “Infinite games on finitely coloured graphs with applications to automata on infinite trees”. In: 200.1-2 (June 1998), pp. 135–183.

Appendix A

Well-foundedness proof

Lemma 5. The state space $\mathcal{S} = (S_\mathcal{G}, \prec_\mathcal{G})$ as defined in Definition 6.4.1 is a well-founded partial order w.r.t $\prec$.

Proof. Since $S_\mathcal{G}$ is finite, to show that it is a well-founded partial order w.r.t. $\prec_\mathcal{G}$ it suffices to prove that it is a strict partial order on $S_\mathcal{G}$; to show it is irreflexive and transitive. Through property 2.(b.ii) of Definition 6.4.1, we can see that $s \not\prec_\mathcal{G} s$ for all states $s = (r, p) \in S_\mathcal{G}$ since neither $p \not< p$ nor does there exist a priority $q \in \texttt{rng}(r)$ such that $r^{-1}(q) \subset r^{-1}(q)$, since $r = r$.

For the transitive property, we consider three states $s_1 = (r_1, p_1), s_2 = (r_2, p_2)$ and $s_3 = (r_3, p_3)$ with $s_1, s_2, s_3 \in S_\mathcal{G}$, such that $s_1 \prec_\mathcal{G} s_2$ and $s_2 \prec_\mathcal{G} s_3$ holds. We can now make a case distinction on all four of the possible cases:

Case 1) Property 3(a) from Definition 6.4.1 holds for both $s_1 \prec_\mathcal{G} s_2$ and $s_2 \prec_\mathcal{G} s_3$: there exist priorities $q_1 \in \texttt{rng}(r_1)$ and $q_2 \in \texttt{rng}(r_2)$ with $q_1 \leq p_1$ and $q_2 \leq p_2$ such that $r_1^{(<q_1)} = r_2^{(<q_1)}$, $r_2^{(<q_2)} = r_3^{(<q_2)}$, $r_2^{-1}(q_1) \subset r_1^{-1}(q_1)$ and $r_3^{-1}(q_2) \subset r_2^{-1}(q_2)$. Let $q = \min\{q_1, q_2\} \leq p_1$. We get the following cases:

- If $q = q_1 = q_2$ then $r_1^{(<q)} = r_2^{(<q)} = r_3^{(<q)}$ and $r_3^{-1}(q) \subset r_2^{-1}(q) \subset r_1^{-1}(q)$.
- If $q = q_1 < q_2$ then we have that $r_1^{(<q)} = r_2^{(<q)} = (r_2^{(<q_2)})^{(<q)} = (r_3^{(<q_2)})^{(<q)} = r_3^{(<q)}$ and $r_3^{-1}(q) \subset r_2^{-1}(q) \subset r_1(q)$.
- Lastly, if $q = q_2 < q_1$ then $r_1^{(<q)} = (r_1^{(<q_1)})^{(<q)} = (r_2^{(<q_1)})^{(<q)} = r_2^{(<q)} = r_3^{(<q)}$ and $r_3^{-1}(q) \subset r_2^{-1}(q) = r_1^{-1}(q)$.

As q is either in the range of r_1 or r_2 by definition, it holds that $q \in \texttt{rng}(r_1)$ and $q \in \texttt{rng}(r_2)$ and therefore $s_1 \prec_\mathcal{G} s_3$.

Case 2) Property 3(a) for $s_1 \prec_\mathcal{G} s_2$ and 3(b) for $s_2 \prec_\mathcal{G} s_3$. Therefore, there exists priority $q \in \texttt{rng}(r_1)$ with $q \leq p_1$ such that $r_1^{(<q)} = r_2^{(<q)}$ and $r_2^{-1}(q) \subset r_1^{-1}(q)$ and by property 3.(b.i) $r_2 = r_3$. So $r_1^{(<q)} = r_2^{(<q)} = r_3^{(<q)}$ and $r_3^{-1}(q) = r_2^{-1}(q) \subset r_1^{-1}(q)$, and we can conclude that $s_1 \prec_\mathcal{G} s_3$. The symmetric case can be proven similarly.

Case 3) Property 3(b) for both $s_1 \prec_\mathcal{G} s_2$ and $s_2 \prec_\mathcal{G} s_3$. It follows that $r_1 = r_2$, $r_2 = r_3$,$p_1 > p_2$ and $p_2 > p_3$. Hence it follows that $r_1 = r_3$ and $p_1 > p_3$ therefore $s_1 \prec_\mathcal{G} s_3$.

□