Verifying Featured Transition Systems using Variability Parity Games

Sjef van Loo

October 1, 2019

Abstract

abstract

Contents

1 Introduction

Model verification techniques can be used to improve the quality of software. These techniques require the behaviour of the software to be modelled, after which the model can checked to verify that it behaves conforming to some requirement. Different languages are proposed and well studied to express these requirements, examples are LTL, CTL, CTL* and μ-calculus (TODO: cite). Once the behaviour is modelled and the requirement is expressed in some language we can use modal checking techniques to determine if the model satisfies the requirement.

These techniques are well suited to model and verify the behaviour of a single software product. However software systems can be designed to have certain parts enabled or disabled. This gives rise to many software products that all behave very similar but not identical, such a collection is often called a *product family*. The differences between the products in a product family is called the *variability* of the family. A family can be verified by using the above mentioned techniques to verify every single product independently. However this approach does not use the similarities in behaviour of these different products, an approach that would make use of the similarities could potentially be a lot more efficient.

Labelled transition systems (LTSs) are often used to model the behaviour of a system, while it can model behaviour well it can't model variability. Efforts to also model variability include I/O automata, modal transition systems and *featured transition systems* (FTSs) (TODO: cite). Specifically the latter is well suited to model all the different behaviours of the software products as well as the variability of the entire system in a single model.

Efforts have been made to verify requirements for entire FTSs, as well as to be able to reason about features. Notable contributions are fLTL, fCTL and fNuSMV (TODO: cite). However, as far as we know, there is no technique to verify an FTS against a μ-calculus formula. Since the modal μ-calculus is very expressive, it subsumes other temporal logics like LTL, CTL and CTL*, this is desired. In this thesis we will introduce a technique to do this. We first look at LTSs, the modal μ-calculus and FTSs. Next we will look at an existing technique to verify an LTS, namely solving *parity games*, as well as show how this technique can be used to verify an FTS by verifying every software product it describes independently. An extension to this technique is then proposed, namely solving *variability parity games*. We will formally define variability parity games and prove that solving them can be used to verify FTSs.

2 Preliminaries

2.1 Fixed-point theory

A fixed-point of a function is an element in the domain of that function such that the function maps to itself for that element. Fixed-points are used in model verification as well as in some parity game algorithms.

Fixed-point theory goes hand in hand with lattice theory which we introduce first.

2.1.1 Lattices

We introduce definitions for ordering and lattices taken from [1].

Definition 2.1 ([1])**.** *A partial order is a binary relation $x \leq y$ on set S where for all $x, y, z \in S$ we have:*

- *$x \leq x$. (Reflexive)*
- *If $x \leq y$ and $y \leq x$, then $x = y$. (Antisymmetric)*
- *If $x \leq y$ and $y \leq z$, then $x \leq z$. (Transitive)*

Definition 2.2 ([1])**.** *A partially ordered set is a set S and a partial order $\leq$ for that set, we denote a partially ordered set by $\langle S, \leq \rangle$.*

Definition 2.3 ([1])**.** *Given partially ordered set $\langle P, \leq \rangle$ and subset $X \subseteq P$. An upper bound to X is an element $a \in P$ such that $x \leq a$ for every $x \in X$. A least upper bound to X is an upper bound $a \in P$ such every other upper bound is larger or equal to a.*

The term least upper bound is synonymous with the term supremum.

Definition 2.4 ([1]). *Given partially ordered set $\langle P, \leq \rangle$ and subset $X \subseteq P$. A lower bound to X is an element $a \in P$ such that $a \leq x$ for every $x \in X$. A greatest lower bound to X is a lower bound $a \in P$ such that every other lower bound is smaller or equal to a.*

The term greatest lower bound is synonymous with the term infimum.

Definition 2.5 ([1]). *A lattice is a partially ordered set where any two of its elements have a supremum and an infimum.*

Definition 2.6 ([1]). *A complete lattice is a partially ordered set in which every subset has a supremum and an infimum.*

Definition 2.7 ([1]). *A function $f : D \to D'$ is monotonic, also called order preserving, if for all $x \in D$ and $y \in D$ it holds that if $x \leq y$ then $f(x) \leq f(y)$.*

2.1.2 Fixed-points

Fixed-points are formally defined as follows:

Definition 2.8. *Given function $f : D \to D$ the value $x \in D$ is a fixed point for f if and only if $f(x) = x$.*

Definition 2.9. *Given function $f : D \to D$ the value $x \in D$ is the least fixed point for f if and only if x is a fixed point for f and every other fixed point for f is greater or equal to x.*

Definition 2.10. *Given function $f : D \to D$ the value $x \in D$ is the greatest fixed point for f if and only if x is a fixed point for f and every other fixed point for f is less or equal to x.*

The Knaster-Tarski theorem states that least and greatest fixed points exist for some domain and function given that a few conditions hold. The theorem, as written down by Tarski in [17], states:

Theorem 2.1 (Knaster-Tarski[17]). *Let*

- *$\langle A, \leq \rangle$ be a complete lattice,*
- *f be an increasing function on A to A,*
- *P be the set of all fixpoints of f.*

Then the set P is not empty and the system $\langle P, \leq \rangle$ is a complete lattice; in particular we have

$$\sup P = \sup\{x \mid f(x) \geq x\} \in P$$

and

$$\inf P = \inf\{x \mid f(x) \leq x\} \in P$$

2.2 Model verification

It is difficult to develop correct software, one way to improve reliability of software is through model verification; the behaviour of software is specified in a model and formal verification techniques are used to show that the behaviour adheres to certain requirements. In this section we inspect how to model behaviour and how to specify requirements.

Behaviour can be modelled as a *labelled transition system* (LTS). An LTS consists of states in which the system can find itself and transitions between states. Transitions represent the possible state change of the system. Transitions are labelled with actions that indicate what kind of change is happening. Formally we define an LTS as follows.

Definition 2.11 ([11]). *A labelled transition system (LTS) is a tuple $M = (S, Act, trans, s_0)$, where:*

- *S is a finite set of states,*

- *Act a finite set of actions,*
- $trans \subseteq S \times Act \times S$ *is the transition relation with* $(s, a, s') \in trans$ *denoted by* $s \xrightarrow{a} s'$,
- $s_0 \in S$ *is the initial state.*

An LTS is usually depicted as a graph where the vertices represent the states, the edges represent the transitions, edges are labelled with actions and an edge with no origin vertex indicates the initial state. Such a representation is depicted in the example below.

Example 2.1 ([18])**.** Consider the behaviour of a coffee machine that accepts a coin after which it serves a standard coffee, this can be repeated infinitely often.

The behaviour can be modelled as an LTS that has two states: in the initial state it is ready to accept a coin and in the second state it is ready to serve a standard coffee. We introduce two actions: *ins*, which represents a coin being inserted, and *std*, which represents a standard coffee being served. We get the following LTS which is also depicted in Figure 1.

$$(\{s_1, s_2\}, \{std, ins\}, \{(s_1, ins, s_2), (s_2, std, s_1)\}, s_1)$$

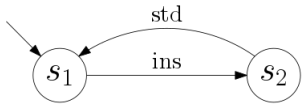


Figure 1: Coffee machine LTS C

LTSs might be non-deterministic, meaning that from a state there might be multiple transitions that can be taken, moreover multiple transitions with the same action can be taken. This is depicted in the example below.

Example 2.2. We extend the coffee machine example where at some point the coffee machine can be empty and needs a fill before the system is ready to receive a coin again. This LTS is depicted in Figure 2. When the *std* transition is taken from state s_2 it is non-determined in which states the system ends.

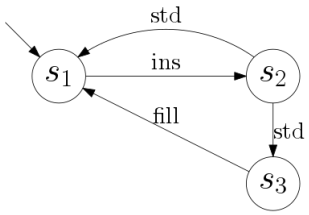


Figure 2: Coffee machine with non-deterministic behaviour

A system can be verified by checking if its behaviour adheres to certain requirements. The behaviour can be modelled in an LTS. Requirements can be expressed in a temporal logic; with a temporal logic we can express certain propositions with a time constraint such as *always*, *never* or *eventually*. For example (relating to the coffee machine example) we can express the following constraint: "After a coin is inserted the machine always serves a standard coffee immediately afterwards". The most expressive temporal logic is the modal μ-calculus. A modal μ-calculus formula is expressed over a set of actions and a set of variables.

We define the syntax of the modal μ-calculus below. Note that the syntax is in positive normal form, ie. no negations.

Definition 2.12 ([11])**.** *A modal μ-calculus formula over the set of actions Act and a set of variables $\mathcal{X}$ is defined by*

$$\varphi = \top \mid \bot \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X.\varphi \mid \nu X.\varphi$$

with $a \in Act$ and $X \in \mathcal{X}$.

The modal μ-calculus contains boolean constants $\top$ and $\bot$, propositional operators $\vee$ and $\wedge$, modal operators $\langle\,\rangle$ and $[\,]$ and fixpoint operators μ and ν.

A variable $X \in \mathcal{X}$ *occurs free* in formula ϕ if and only if X occurs in ϕ such that X is not a sub-formula of $\mu X.\phi'$ or $\nu X.\phi'$ in ϕ. A formula is *closed* if and only if there are no variables that occurs free.

A formula can be interpreted in the context of an LTS, such an interpretation results in a set of states in which the formula holds. Given formula φ we define the interpretation of φ as $[\![\varphi]\!]^\eta \subseteq S$ where $\eta : \mathcal{X} \to 2^S$ maps a variable to a set of states. We can assign $S' \subseteq S$ to variable X in η by writing $\eta[X := S']$, ie. $(\eta[X := S'])(X) = S'$.

Definition 2.13. *For LTS $(S, Act, trans, s_0)$ we inductively define the interpretation of a modal μ-calculus formula φ, notation $[\![\varphi]\!]^\eta$, where $\eta : \mathcal{X} \to \mathcal{P}(S)$ is a variable valuation, as a set of states where φ is valid, by:*

$$
\begin{aligned}
[\![\top]\!]^\eta &= S \\
[\![\bot]\!]^\eta &= \emptyset \\
[\![\varphi_1 \wedge \varphi_2]\!]^\eta &= [\![\varphi_1]\!]^\eta \cap [\![\varphi_2]\!]^\eta \\
[\![\varphi_1 \vee \varphi_2]\!]^\eta &= [\![\varphi_1]\!]^\eta \cup [\![\varphi_2]\!]^\eta \\
[\![\langle a \rangle \varphi]\!]^\eta &= \{s \in S | \exists_{s' \in S}\ s \xrightarrow{a} s' \wedge s' \in [\![\varphi]\!]^\eta\} \\
[\![[a] \varphi]\!]^\eta &= \{s \in S | \forall_{s' \in S}\ s \xrightarrow{a} s' \implies s' \in [\![\varphi]\!]^\eta\} \\
[\![\mu X.\varphi]\!]^\eta &= \bigcap \{f \subseteq S | f \supseteq [\![\varphi]\!]^{\eta[X:=f]}\} \\
[\![\nu X.\varphi]\!]^\eta &= \bigcup \{f \subseteq S | f \subseteq [\![\varphi]\!]^{\eta[X:=f]}\} \\
[\![X]\!]^\eta &= \eta(X)
\end{aligned}
$$

Since there are no negations in the syntax we find that every modal μ-calculus formula is monotone, ie. if we have for $U \subseteq S$ and $U' \subseteq S$ that $U \subseteq U'$ holds then $[\![\varphi]\!]^{\eta[X:=U]} \subseteq [\![\varphi]\!]^{\eta[X:=U']}$ holds for any variable $X \in \mathcal{X}$. Using the Knaster-Tarski theorem (Theorem 2.1) we find that the least and greatest fixed-points always exist.

Given closed formula φ, LTS $M = (S, Act, trans, s_0)$ and $s \in S$ we say that M satisfies formula φ in state s, and write $(M, s) \models \varphi$, if and only if $s \in [\![\varphi]\!]^\eta$. If and only if M satisfies φ in the initial state do we say that M satisfies formula φ and write $M \models \varphi$.

Example 2.3 ([18])**.** Consider the coffee machine example from Figure 1 and formula $\varphi = \nu X.\mu Y([inst]Y \wedge [std]X)$ which states that action *std* must occur infinitely often over all infinite runs. Obviously this holds for the coffee machine, therefore we have $C \models \varphi$.

2.3 Parity games

A *parity game* is a game played by two players: player 0 (also called player *even*) and player 1 (also called player *odd*). We write $\alpha \in \{0, 1\}$ to denote an arbitrary player and $\overline{\alpha}$ to denote α's opponent, ie. $\overline{0} = 1$ and $\overline{1} = 0$. A parity game is played on a playing field which is a directed graph where every vertex is owned by either player 0 or player 1. Furthermore every vertex has a natural number, called its *priority*, associated with it.

Definition 2.14 ([2])**.** *A parity game is a tuple (V, V_0, V_1, E, Ω), where:*

- *V is a finite set of vertices partitioned in sets V_0 and V_1, containing vertices owned by player 0 and player 1 respectively,*
- *$E \subseteq V \times V$ is the edge relation,*
- *$\Omega : V \to \mathbb{N}$ is the priority assignment function.*

Parity games are usually represented as a graph where vertices owned by player 0 are shown as diamonds and vertices owned by player 1 are shown as boxes, furthermore the priorities are depicted as numbers inside the vertices. Such a representation is shown in the example below.

Example 2.4. Figure 3 shows the parity game:

$$V_0 = \{v_1, v_4, v_5\}, V_1 = \{v_2, v_3\}, V = V_0 \cup V_1$$

$$E = \{(v_1, v_2), (v_2, v_1), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_4)\}$$

$$\Omega = \{v_1 \mapsto 2, v_2 \mapsto 3, v_3 \mapsto 0, v_4 \mapsto 0, v_5 \mapsto 1\}$$

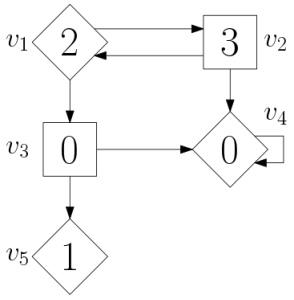


Figure 3: Parity game example

A parity game can be played for a vertex $v \in V$. We start by placing a token on vertex v, the player that owns vertex v can choose to move the token along an edge to a vertex $w \in V$ such that $(v, w) \in E$. Again the player that owns vertex w can choose where to move the token next. This is repeated either infinitely often or until a player cannot make a move, ie. the token is on a vertex with no outgoing edges. Playing in this manner gives a sequence of vertices, called a *path*, starting from vertex v. For path π we write π_i to denote i^{th} vertex in path π. Every path is associated with a winner (either player 0 or 1). If a player α cannot move at some point we get a finite path and player $\overline{\alpha}$ wins the path. If we get an infinite path π then the winner is determined by the parity of the highest priority that occurs infinitely often in the path. Formally we determine the highest priority occurring infinitely often by the following formula.

$$\max\{p \mid \forall_j \exists_i j < i \wedge p = \Omega(\pi_i)\}$$

If the highest priority is odd then player 1 wins the path, if it is even player 0 wins the path.

A path is *valid* if and only if for every $i > 0$ such that π_i exists we have $(\pi_{i-1}, \pi_i) \in E$.

Example 2.5. Again consider the example in Figure 3. If we play the game for vertex v_1 we start by placing a token on v_1. Consider the following exemplary paths where $(w_1 \ldots w_m)^\omega$ indicates an infinite repetition of vertices $w_1 \ldots w_m$.

- $\pi = v_1 v_3 v_5$ is won by player 1 since player 0 cannot move at v_5.
- $\pi = (v_1 v_2)^\omega$ is won by player 1 since the highest priority occurring infinitely often is 3.
- $\pi = v_1 v_3 (v_4)^\omega$ is won by player 0 since the highest priority occurring infinitely often is 0.

The moves that the players make are determined by their *strategies*. A strategy σ_α determines for a vertex in V_α where the token goes next. We can define a strategy for player α as a partial function $\sigma_\alpha : V^* V_\alpha \to V$ that maps a series of vertices ending with a vertex owned by player α to the next vertex such that for any $\sigma_\alpha(w_0 \ldots w_m) = w$ we have $(w_m, w) \in E$. A path π *conforms to* strategy σ_α if for every $i > 0$ such that π_i exists and $\pi_{i-1} \in V_\alpha$ we have $\pi_i = \sigma_\alpha(\pi_0 \pi_1 \ldots \pi_{i-1})$.

A strategy is *winning* for player α from vertex v if and only if α is the winner of every valid path starting in v that conforms to σ_α. If such a strategy exists for player α from vertex v we say that vertex v is winning for player α.

Example 2.6. In the parity game seen in Figure 3 vertex v_1 is winning for player 1. Player 1 has a strategy that plays every vertex sequence ending in v_2 to v_1 and plays every vertex sequence ending in v_3 to v_5. Regardless of the strategy for player 0 the path will either end up in v_5 or will pass v_2 infinitely often. In the former case player 1 wins the path because player 0 can not move at v_5. In the latter case the highest priority occurring infinitely often is 3.

Parity games are known to be positionally determined [2], meaning that every vertex in a parity game is winning for one of the two players. Also every player has a *positional strategy* that is winning starting from each of his/her winning vertices. A positional strategy is a strategy that only takes the current vertex into account to determine the next vertex, it does not look at already visited vertices. Therefore we can consider a strategy for player α as a complete functions $\sigma_\alpha : V_\alpha \to V$. Finally it is decidable for each of the vertices in a parity who the winner is [2].

A parity game is *solved* if the vertices are partitioned in two sets, namely W_0 and W_1, such that every vertex in W_0 is winning for player 0 and every vertex in W_1 is winning for player 1. We call these sets the *winning sets* of a parity game.

Finally parity games are considered *total* if and only if every vertex has at least one outgoing edge. Playing a total parity game always results in an infinite path. We can make a non-total parity game total by adding two sink vertices: l_0 and l_1. Each sink vertex has only one outgoing edge, namely to itself. Vertex l_0 has priority 1 and vertex l_1 has priority 0. Clearly if the token ends up in l_α then player α looses the game because with only one outgoing edge we only get a single priority that occurs infinitely often, namely priority $\overline{\alpha}$. For every vertex $v \in V_\alpha$ that does not have an outgoing edge we create an edge from v to l_α. In the original game player α lost when the token was in vertex v because he/she could not move any more. In the total game player α can only play to l_α from v where he/she still looses. So using this method vertices in the total game have the same winner as they had in the original game (except for l_0 and l_1 which did not exist in the original game). In general we try to only work with total games because no distinction is required between finite paths and infinite paths when reasoning about them, however we will encounter some scenario's where non-total games are still considered.

2.3.1 Relation between parity games and model checking

Verifying LTSs against a modal μ-calculus formula can be done by solving a parity game. This is done by translating an LTS in combination with a formula to a parity game, the solution of the parity game provides the information needed to conclude if the model satisfies the formula. This relation is depicted in figure 4.

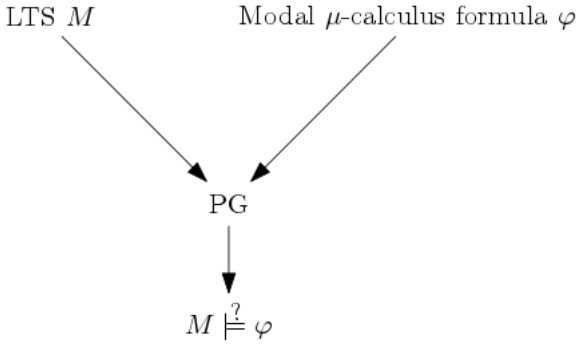


Figure 4: LTS verification using PG

We consider a method of creating parity games from an LTS and a modal μ-calculus formula such that there is a special vertex w in the parity game that indicates if the LTS satisfies the formula; if and only if w is won by player 0 is the formula satisfied.

First we introduce the notion of unfolding. A fixpoint formula $\mu X.\varphi$ can be unfolded, resulting in formula φ where every occurrence of X is replaced by $\mu X.\varphi$, denoted by $\varphi[X := \mu X.\varphi]$. Interpreting a fixpoint formula results in the same set as interpreting its unfolding as shown in [2]; i.e. $[\![\mu X.\varphi]\!]^\eta = [\![\varphi[X := \mu X.\varphi]]\!]^\eta$. The same holds for the fixpoint operator ν.

Next we define the Fischer-Ladner closure for a closed μ-calculus formula [16, 7]. The Fischer-Ladner closure of φ is the set $FL(\varphi)$ of closed formulas containing at least φ. Furthermore for every formula ψ in $FL(\varphi)$ it holds that for every direct subformula ψ' of ψ there is a formula in $FL(\varphi)$ that is equivalent to ψ'.

Definition 2.15. *The Fischer-Ladner closure of closed μ-calculus formula φ is the smallest set $FL(\varphi)$ satisfying the following constraints:*

- $\varphi \in FL(\varphi)$,
- *if* $\varphi_1 \vee \varphi_2 \in FL(\varphi)$ *then* $\varphi_1, \varphi_2 \in FL(\varphi)$,
- *if* $\varphi_1 \wedge \varphi_2 \in FL(\varphi)$ *then* $\varphi_1, \varphi_2 \in FL(\varphi)$,
- *if* $\langle a \rangle \varphi' \in FL(\varphi)$ *then* $\varphi' \in FL(\varphi)$,
- *if* $[a]\varphi' \in FL(\varphi)$ *then* $\varphi' \in FL(\varphi)$,
- *if* $\mu X.\varphi' \in FL(\varphi)$ *then* $\varphi'[X := \mu X.\varphi'] \in FL(\varphi)$ *and*
- *if* $\nu X.\varphi' \in FL(\varphi)$ *then* $\varphi'[X := \nu X.\varphi'] \in FL(\varphi)$.

We also define the alternation depth of a formula.

Definition 2.16 ([2])**.** *The dependency order on bound variables of φ is the smallest partial order such that $X \leq_\varphi Y$ if X occurs free in $\sigma Y.\psi$. The alternation depth of a μ-variable X in formula φ is the maximal length of a chain $X_1 \leq_\varphi \cdots \leq_\varphi X_n$ where $X = X_1$, variables $X_1, X_3, \ldots$ are μ-variables and variables $X_2, X_4, \ldots$ are ν-variables. The alternation depth of a ν-variable is defined similarly. The alternation depth of formula φ, denoted $adepth(\varphi)$, is the maximum of the alternation depths of the variables bound in φ, or zero if there are no fixpoints.*

Example 2.7. Consider the formula $\varphi = \nu X.\mu Y.([ins]Y \wedge [std]X)$ which states that for an LTS with $Act = \{ins, std\}$ the action std must occur infinitely often over all infinite runs. Since X occurs free in $\mu Y.([ins]Y \wedge [std]X)$ we have $adepth(Y) = 1$ and $adepth(X) = 2$.

As shown in [2] it holds that formula $\mu X.\psi$ has the same alternation depth as its unfolding $\psi[X := \mu X.\psi]$. Similarly for the greatest fixpoint.

Next we define the transformation from an LTS and a formula to a parity game.

Definition 2.17 ([2])**.** *LTS2PG(M, φ) converts LTS $M = (S, Act, trans, s_0)$ and closed formula φ to a parity game (V, V_0, V_1, E, Ω).*

Vertices in the parity game are presented as pairs of states and sub-formulas. A vertex is created for every state with every formula in the Fischer-Ladner closure of φ. We define the set of vertices:

$$V = S \times FL(\varphi)$$

Vertices have the following owner and successors:

Vertex	*Owner*	*Successor(s)*
$(s, \bot)$	*0*	
$(s, \top)$	*1*	
$(s, \psi_1 \vee \psi_2)$	*0*	(s, ψ_1) *and* (s, ψ_2)
$(s, \psi_1 \wedge \psi_2)$	*1*	(s, ψ_1) *and* (s, ψ_2)
$(s, \langle a \rangle \psi)$	*0*	(s', ψ) *for every* $s \xrightarrow{a} s'$
$(s, [a]\psi)$	*1*	(s', ψ) *for every* $s \xrightarrow{a} s'$
$(s, \mu X.\psi)$	*1*	$(s, \psi[X := \mu X.\psi])$
$(s, \nu X.\psi)$	*1*	$(s, \psi[X := \nu X.\psi])$

Since the Fischer-Ladner formula's are closed we never get a vertex (s, X).

Finally we have $\Omega(v) = \begin{cases} 2\lfloor adepth(X)/2 \rfloor & \text{if } v = (s, \nu X.\psi) \\ 2\lfloor adepth(X)/2 \rfloor + 1 & \text{if } v = (s, \mu X.\psi) \\ 0 & \text{otherwise} \end{cases}$

Example 2.8. Consider LTS M in figure 5 and formula $\varphi = \mu X.([a]X \vee \langle b \rangle \top)$ expressing that on any path reached by a's we can eventually do a b action.

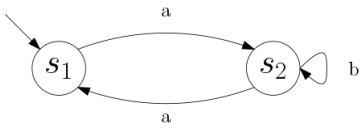


Figure 5: LTS M

The resulting parity game is depicted in figure 6. Let V denote the set of vertices of this parity game. There are two vertices with more than one outgoing edge. From vertex $(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top)$ player 0 does not want to play to $(s_1, \langle b \rangle \top)$ because he/she will not be able to make another move and looses the path. From vertex $(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top)$ player 0 can play to $(s_2, \langle b \rangle \top)$ to bring the play in $(s_2, \top)$ to win the path. We get the following winning sets:

$$W_1 = \{(s_1, \langle b \rangle \top)\}$$
$$W_0 = V \backslash W_1$$

With the strategies σ_0 for player 0 and σ_1 for player 1 being (vertices with one outgoing edge are omitted):

$$\sigma_0 = \{(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top) \mapsto (s_1, [a](\mu X.\phi)),$$
$$(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top) \mapsto (s_2, \langle b \rangle \top)\}$$
$$\sigma_1 = \{\}$$

Note that the choice where to go from $(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top)$ does not matter for the winning sets.

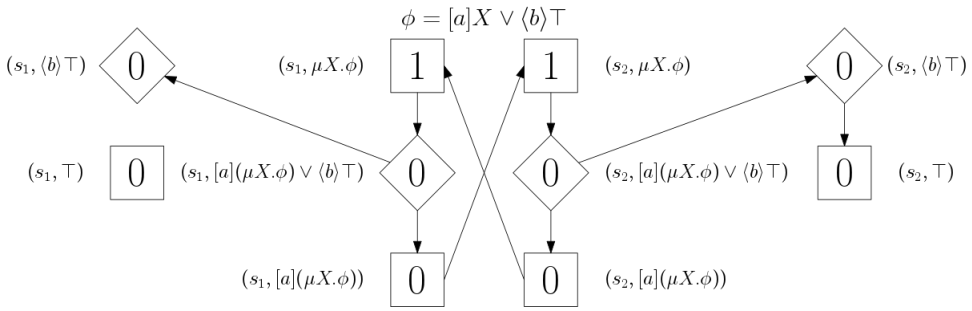


Figure 6: Parity game $LTS2PG(M, \varphi)$

Parity games created in this manner relate back to the model verification question; state s in LTS M satisfies φ if and only if player 0 wins vertex (s, φ). This is formally stated in the following theorem which is proven in [2].

Theorem 2.2 ([2])**.** *Given LTS* $M = (S, Act, trans, s_0)$*, modal* μ*-calculus formula* φ *and state* $s \in S$ *it holds that* $(M, s) \models \varphi$ *if and only if* $(s, \varphi) \in W_0$ *for the game* $LTS2PG(M, \varphi)$.

2.3.2 Globally and locally solving parity games

Parity games can be solved *globally* or *locally*; globally solving a parity game means that for every vertex in the game it is determined who the winner is. Locally solving a parity game means that for a specific vertex

in the game it is determined who the winner is. For some applications of parity games, including model checking, there is a specific vertex that needs to be solved to solve the original problem. Locally solving the parity game is sufficient in such cases to solve the original problem.

Most parity game algorithms (including the two considered next) are concerned with globally solving, when talking about solving a parity game we talk about globally solving it unless stated otherwise.

2.3.3 Parity game algorithms

Various algorithms for solving parity games are known, we introduce two of them. First Zielonka's recursive algorithm which is well studied and generally considered to be one of the best performing parity game algorithms in practice [19, 8]. We also inspect the fixed-point iteration algorithm which tends to perform well for model-checking problems with a low number of distinct priorities [15].

Zielonka's recursive algorithm First we consider Zielonka's recursive algorithm [23, 13], which solves total parity games. Pseudo code is presented in algorithm 1. Zielonka's recursive algorithm has a worst-case time complexity of $O(e * n^d)$ where e is the number of edges, n the number of vertices and d the number of distinct priorities.

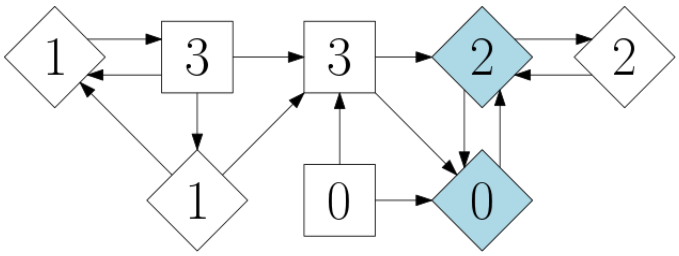
Algorithm 1 RecursivePG($PG\ G = (V, V_0, V_1, E, \Omega)$)

```
1: m ← min{Ω(v) | v ∈ V}
2: h ← max{Ω(v) | v ∈ V}
3: if h = m or V = ∅ then
4:     if h is even or V = ∅ then
5:         return (V, ∅)
6:     else
7:         return (∅, V)
8:     end if
9: end if
10: α ← 0 if h is even and 1 otherwise
11: U ← {v ∈ V | Ω(v) = h}
12: A ← α-Attr(G, U)
13: (W'_0, W'_1) ← RecursivePG(G\A)
14: if W'_ᾱ = ∅ then
15:     W_α ← A ∪ W'_α
16:     W_ᾱ ← ∅
17: else
18:     B ← ᾱ-Attr(G, W'_ᾱ)
19:     (W''_0, W''_1) ← RecursivePG(G\B)
20:     W_α ← W''_α
21:     W_ᾱ ← W''_ᾱ ∪ B
22: end if
23: return (W_0, W_1)
```

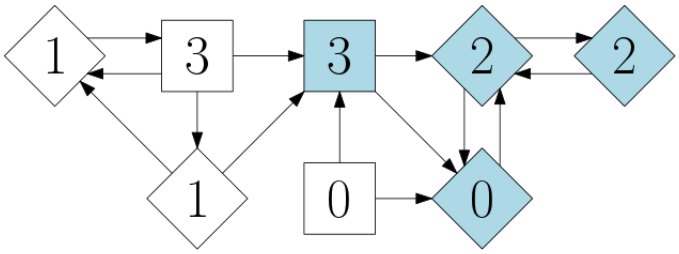
The algorithm solves G by taking the set of vertices with the highest priority and choosing player α such that α has the same parity as the highest priority. Next the algorithm finds set A such that player α can force the play to one of these high priority vertices. Next this set of vertices is removed from G and the resulting subgame G' is solved recursively.

If G' is entirely won by player α then we distinguish three cases for any path played in G. Either the path eventually stays in G', A is infinitely often visited or the path eventually stays in A. In the first case player α wins because game G' was entirely won by player α. In the second and third case player α can play to the highest priority from A. The highest priority, which has parity α, is visited infinitely often and player α wins.

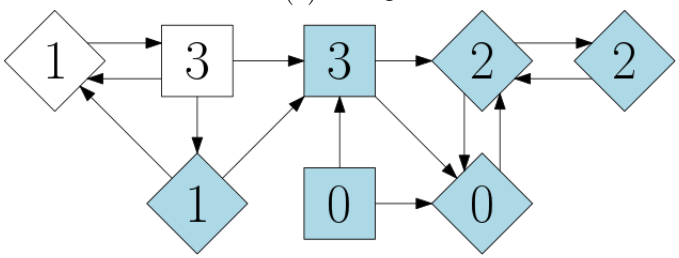
If G' is not entirely won by player α we consider winning sets W'_0 and W'_1 of subgame G'. Vertices in set $W'_{\overline{\alpha}}$ are won by player $\overline{\alpha}$ in G' but are also won by player $\overline{\alpha}$ in G. The algorithm tries to find all the vertices



(a) Set $U = U_0$



(b) Set U_1



(c) Set $U_2 = 0\text{-}Attr(G, U)$

Figure 7: Game G showing the attractor calculation for $0\text{-}Attr(G, U)$

in G such that player $\overline{\alpha}$ can force the play to a vertex in $W'_{\overline{\alpha}}$ and therefore winning the game. We now have a set of vertices that are definitely won by player $\overline{\alpha}$ in game G. In the rest of the game player α can keep the play from $W'_{\overline{\alpha}}$ so the algorithm solves the rest of the game recursively to find the complete winning sets for game G.

A complete explanation of the algorithm can be found in [23], we do introduce definitions for the attractor set and for subgames.

An attractor set is a set of vertices $A \subseteq V$ calculated for player α given set $U \subseteq V$ where player α has a strategy to force the play starting in any vertex in $A \backslash U$ to a vertex in U. Such a set is calculated by adding vertices owned by player α that have an edge to the attractor set and adding vertices owned by player $\overline{\alpha}$ that only have edges to the attractor set.

Definition 2.18 ([23]). *Given parity game $G = (V, V_0, V_1, E, \Omega)$ and a non-empty set $U \subseteq V$ we define $\alpha\text{-}Attr(G, U)$ such that*

$$U_0 = U$$

For $i \geq 0$:

$$\begin{aligned} U_{i+1} = U_i \cup \{v \in V_\alpha \mid \exists v' \in V : v' \in U_i \wedge (v, v') \in E\} \\ \cup \{v \in V_{\overline{\alpha}} \mid \forall v' \in V : (v, v') \in E \implies v' \in U_i\} \end{aligned}$$

Finally:

$$\alpha\text{-}Attr(G, U) = \bigcup_{i \geq 0} U_i$$

Example 2.9. Figure 7 shows an example parity game in which an attractor set is calculated for player 0. For set U_2 no more vertices can be attracted so we found the complete attractor set.

The algorithm also creates subgames, where a set of vertices is removed from a parity game to create a new parity game.

Definition 2.19 ([23])**.** *Given a parity game $G = (V, V_0, V_1, E, \Omega)$ and $U \subseteq V$ we define the subgame $G \backslash U$ to be the game $(V', V_0', V_1', E', \Omega)$ with:*

- $V' = V \backslash U$,
- $V_0' = V_0 \cap V'$,
- $V_1' = V_1 \cap V'$ *and*
- $E' = E \cap (V' \times V')$.

Note that a subgame is not necessarily total, however the recursive algorithm always creates subgames that are total (shown in [23]).

Fixed-point iteration algorithm Parity games can be solved by solving an alternating fixed-point formula [20]. Consider PG $G = (V, V_0, V_1, E, \Omega)$ with d distinct priorities. We can apply *priority compression* to make sure every priority in G maps to a value in $\{0, \ldots, d-1\}$ or $\{1, \ldots, d\}$ [9, 3]. We assume without loss of generality that the priorities map to $\{0, \ldots, d-1\}$ and that $d-1$ is even.

Consider the following formula

$$S(G = (V, V_0, V_1, E, \Omega)) = \nu Z_{d-1}.\mu Z_{d-2}.\ldots.\nu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$$

with

$$F_0(Z_{d-1}, \ldots, Z_0) = \{v \in V_0 \mid \exists_{w \in V}(v, w) \in E \wedge Z_{\Omega(w)}\} \cup \{v \in V_1 \mid \forall_{w \in V}(v, w) \in E \implies Z_{\Omega(w)}\}$$

where $Z_i \subseteq V$. The formula $\nu X.f(X)$ solves the greatest fixed-point of X in f, similarly $\mu X.f(X)$ solves the least fixed-point of X in f. As shown in [20] formula $S(G)$ is calculates the set of vertices winning for player 0 in parity game G.

To understand the formula we consider sub-formula $\nu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$. This formula holds for vertices from which player 0 can either force the play into a node with priority $i > 0$ for which Z_i holds or the player can stay in vertices with priority 0 indefinitely. The formula $\mu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$ holds for vertices from which player 0 can force the play into a node with priority $i > 0$, for which Z_i holds in finitely many steps. By alternating fixed-points the formula allows infinitely many consecutive stays in even vertices and finitely many consecutive stays in odd vertices. For an extensive treatment we refer to [20].

We further inspect formula S. Given game G, consider the following sub-formulas:

$$S^{d-1}(Z_{d-1}) = \mu Z_{d-2}.S^{d-2}(Z_{d-2})$$

$$S^{d-2}(Z_{d-2}) = \nu Z_{d-3}.S^{d-3}(Z_{d-3})$$

$$\ldots$$

$$S^0(Z_0) = F_0(Z_{d-1}, \ldots, Z_0)$$

The fixed-point variables are all elements of 2^V, therefore we have for every sub-formula the following type:

$$S^i(Z_i) : 2^V \to 2^V$$

Furthermore, since V is finite, the partially ordered set $\langle 2^V, \subseteq \rangle$ is a complete lattice; for every subset $X \subseteq 2^V$ we have infimum $\bigcap_{x \in X} x$ and supremum $\bigcup_{x \in X} x$. Finally every sub-formula $S^i(Z_i)$ is monotonic, ie. if $S^i(Z_i) \geq S^i(Z_i')$ then $Z_i \geq Z_i'$.

Fixed-point formula's can be solved by *fixed-point iteration*. As shown in [6] we can calculate $\mu X.f(X)$, where f is monotonic in X and $X \in 2^V$, by iterating X:

$$\mu X.f(X) = \bigcup_{i \geq 0} X^i$$

where $X^i = f(X^{i-1})$ for $i > 0$ and $X^0 \subseteq \mu X.f(X)$. So picking the smallest value possible for X_0 will always correctly calculate $\mu X.f(X)$.

Similarly we can calculate fixed-point $\nu X.f(X)$ when f is monotonic in X by iterating X:

$$\nu X.f(X) = \bigcap_{i \geq 0} X^i$$

where $X^i = f(X^{i-1})$ for $i > 0$ and $X^0 \supseteq \nu X.f(X)$. So picking the largest value possible for X_0 will always correctly calculate $\nu X.f(X)$.

Since every subformula is monotonic and maps from a value in 2^V to another value in 2^V we can apply fixed-point iteration to solve the subformula's, we choose initial values $\emptyset$ for least fixed-point variables and V for greatest fixed-point variables.

An algorithm to perform the iteration is presented in [3] and shown in algorithm 2. This algorithm has a worst-case time complexity of $O(e * n^d)$ where e is the number of edges, n the number of vertices and d the number of distinct priorities.

Algorithm 2 Fixed-point iteration

```
1:  function FPIter(G = (V, V_0, V_1, E, Ω))
2:      for i ← d − 1, ..., 0 do
3:          Init(i)
4:      end for
5:      repeat
6:          Z'_0 ← Z_0
7:          Z_0 ← Diamond() ∪ Box()
8:          i ← 0
9:          while Z_i = Z'_i ∧ i < d − 1 do
10:             i ← i + 1
11:             Z'_i ← Z_i
12:             Z_i ← Z_{i−1}
13:             Init(i − 1)
14:         end while
15:     until i = d − 1 ∧ Z_{d−1} = Z'_{d−1}
16:     return (Z_{d−1}, V \ Z_{d−1})
17: end function
```

```
1: function Init(i)
2:     Z_i ← ∅ if i is odd, V otherwise
3: end function
```

```
1: function Diamond
2:     return {v ∈ V_0 | ∃_{w∈V}(v, w) ∈ E ∧ w ∈ Z_{Ω(w)}}
3: end function
```

```
1: function Box
2:     return {v ∈ V_1 | ∀_{w∈V}(v, w) ∈ E ⟹ w ∈ Z_{Ω(w)}}
3: end function
```

2.4 Symbolically representing sets

A set can straightforwardly be represented by a collection containing all the elements that are in the set. We call this an *explicit* representation of a set. We can also represent sets *symbolically* in which case the set of elements is represented by some sort of formula. A typical way to represent a set symbolically is through a boolean formula encoded in a *binary decision diagram* [21, 4].

Example 2.10. The set $S = \{2, 4, 6, 7\}$ can be expressed by boolean formula:

$$F(x_2, x_1, x_0) = (\neg x_2 \wedge x_1 \wedge \neg x_0) \vee (x_2 \wedge (x_1 \vee \neg x_0))$$

where x_0, x_1 and x_2 are boolean variables. The formula gives the following truth table:

$\mathbf{x_2x_1x_0}$	$\mathbf{F(x_2, x_1, x_0)}$
000	0
001	0
010	1
011	0
100	1
101	0
110	1
111	1

The function F defines set S' in the following way: $S' = \{x_2x_1x_0 \mid F(x_2, x_1, x_0) = 1\}$. As we can see set S' contains the same numbers as S but represented binary.

We can perform set operations on sets represented as boolean functions by performing logical operations on the functions. For example, given boolean formula's f and g representing sets V and W the formula $f \wedge g$ represents set $V \cap W$.

Given a set S with arbitrary elements we can represent subsets $S' \subseteq S$ as boolean formula's by assigning a number to every element in S and creating a boolean formula that maps boolean variables to true if and only if they represent a number such that the element associated with this number in S is also in S'.

2.4.1 Binary decision diagrams

A boolean function can efficiently be represented as a binary decision diagram (BDD), for a comprehensive treatment of BDDs we refer to [21, 4].

BDDs represent boolean formula's as a directed graph where every vertex represents a boolean variable and has two outgoing edges labelled 0 and 1. Furthermore the graph contains special vertices 0 and 1 that have no outgoing edges. We decide if a boolean variable assignment satisfies the formula by starting in the initial vertex of the graph and following a path until we get to either vertex 0 or 1. Since every vertex represents a boolean formula we can create a path from the initial vertex by choosing edge 0 at a vertex if the boolean variable represented by that vertex is false in the variable assignment and choosing edge 1 if is true. Eventually we end up in either vertex 0 or 1. In the former case the boolean variable assignment does not satisfy the formula, in the latter it does.

Example 2.11. Consider the boolean formula in example 2.10. This formula can be represented as the BDD shown in Figure 8. The vertices representing boolean variables are shown as circles and the boolean variables they represent are indicated inside them. The special vertices are represented as squares and the initial vertex is represented by an edge with that has no origin vertex.

The path created from variable assignment $x_2x_1x_0 = 011$ is highlighted in blue in the diagram and shows that this assignment is indeed not satisfied by the boolean formula. The red path shows the variable assignments 110 and 111. Determining the path and the outcome for every variable assignment results in the same truth table as seen in example 2.10.

Given n boolean variables and two boolean functions encoded as BDDs we can perform binary operations $\vee, \wedge$ on the BDDs in $O(2^{2n}) = O(m^2)$ where $m = 2^n$ is the maximum set size that can be represented by n variables [22, 4]. The running time specifically depends on the size of the decision diagrams, in general if the boolean functions are simple then the size of the decision diagram is also small and operations can be performed quickly.

3 Problem statement

If we have an SPL with certain requirements that must hold for every product then we want to apply verification techniques to formally verify that indeed every product satisfies the requirements. We could verify every product individually, however verification is expensive in terms of computing time and the number of different products can grow large. Differences in behaviour between products might be very small; large parts of the different products might behave similar. In this thesis we aim to exploit commonalities

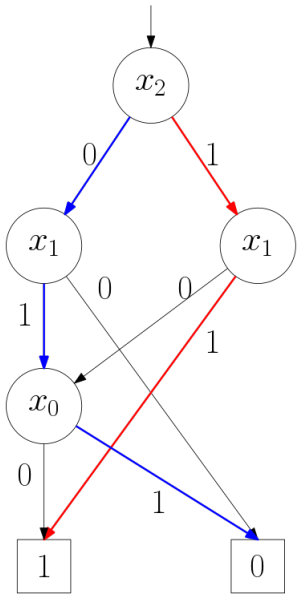


Figure 8: BDD highlighting boolean variable assignment $x_2x_1x_0 = 011$ in blue and $x_2x_1 = 11$ in red

between products to find a method that verifies an SPL in a more efficient way than verifying every product independently.

First we take a look at a method of modelling the behaviour of the different products in an SPL, namely *featured transition systems* (FTSs). An FTS extends the definition an LTS to express variability, it does so by introducing *features* and *products*. Features are options that can be enabled or disabled for the system. A product is a feature assignments, i.e. a set of features that is enabled for that product. Not all products are valid; some features might be mutually exclusive while others might always be required. To express the relation between features one can use feature diagrams as explained in [5]. Feature diagrams offer a nice way of expressing which feature assignments are valid, however for simplicity we represent the collection of valid products simply as a set of feature assignments.

An FTS models the behaviour of multiple products by guarding transitions with boolean expressions over the features such that the transition is only enabled for products that satisfy the guard.

Let $\mathbb{B}(A)$ denote the set of all boolean expressions over the set of boolean variables A, a boolean expression is a function that maps a boolean assignment to either true of false. A boolean expression over a set of features is called a feature expression, it maps a feature assignment, i.e. a product, to either true or false. Given boolean expression f and boolean variable assignment p we write $p \models f$ if and only if f is true for p and write $p \not\models f$ otherwise. Boolean expression $\top$ denotes the boolean expression that is satisfied by all boolean assignments.

Definition 3.1 ([5])**.** *A featured transition system (FTS) is a tuple $M = (S, Act, trans, s_0, N, P, \gamma)$, where:*

- *$S, Act, trans, s_0$ are defined as in an LTS,*
- *N is a non-empty set of features,*
- *$P \subseteq 2^N$ is a non-empty set of products, i.e. feature assignments, that are valid,*
- *$\gamma : trans \to \mathbb{B}(N)$ is a total function, labelling each transition with a features expression.*

A transition $s \xrightarrow{a} s'$ and $\gamma(s, a, s') = f$ is denoted by $s \xrightarrow{a \mid f} s'$. FTSs are presented similarly as LTSs, the labels of the transition are expanded to represent both the action and the feature expression associated with it.

Example 3.1 ([18])**.** Consider a coffee machine that has two variants: in the first variant it takes a single coin and serves a standard coffee, in the second variant the machine either serves a standard coffee after a

coin is inserted or it takes another coin after which it serves an xxl coffee. Note that there is no variant that only serves xxl coffees. We introduce two features: \$ which, if enabled, allows the coffee machine to serve xxl coffees and € which, if enabled, allows the coffee machine to serve standard coffees. The valid products are: $\{\{€\}, \{€, \$\}\}$. This FTS is depicted in Figure 9.

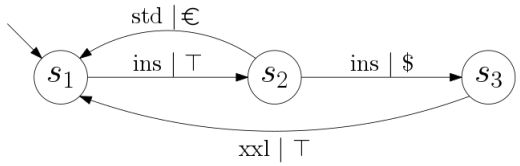
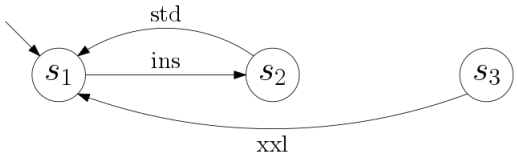


Figure 9: Coffee machine FTS C

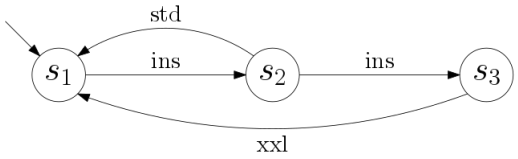
An FTS expresses the behaviour of multiple products. The behaviour of a single product can be derived by simply removing all the transitions from the FTS for which the product does not satisfy the feature expression guarding the transition. We call this a *projection*.

Definition 3.2 ([5])**.** *The projection of FTS* $M = (S, Act, trans, s_0, N, P, \gamma)$ *onto product* $p \in P$*, denoted by* $M_{|p}$*, is the LTS* $(S, Act, trans', s_0)$*, where* $trans' = \{t \in trans \mid p \models \gamma(t)\}$.

Example 3.2 ([18])**.** The coffee machine example can be projected to its two products, which results in the LTSs in figure 10.



(a) C projected to the *euro* product: $C_{|\{€\}}$



(b) C projected to the *euro and dollar* product: $C_{|\{€,\$\}}$

Figure 10: Projections of the coffee machine FTS

Given an FTS M, that models the behaviour of an SPL, with products P and modal μ-calculus formula φ we want to verify that φ holds for every product in P. Formally we want to find set P_s such that:

- for every $p \in P_s$ we have $M_{|p} \models \varphi$ and
- for every $p \in P \backslash P_s$ we have $M_{|p} \not\models \varphi$.

This could be done by projecting the FTS to an LTS for every product and subsequently verifying every such LTS. However, as stated before, we aim to improve upon this method by finding P_s in a way that utilizes the commonalities in behaviour between the different products.

4 Related work

5 Variability parity games

In the preliminaries we have seen how parity games can be used to verify if a modal μ-calculus formula is satisfied by an LTS. Such a parity game contains the information needed to answer the question if an LTS satisfies a modal μ-calculus formula. We have also seen how an LTS can be extended with transition guards to model the behaviour of multiple LTSs. In this section we introduce *variability parity games* (VPGs); a VPG extends the definition of a parity game much like an FTS extends the definition of an LTS. Similar as to how an FTS expresses multiple LTSs does a VPG express multiple parity games, moreover we introduce a way of creating VPGs such that every parity game it expresses contains the information needed to answer the question if a product in an FTS satisfies a modal μ-calculus formula.

We extend parity games such that edges in the game are guarded. Instead of using features, feature expressions and products we choose a syntactically simpler representation and introduce *configurations*. A VPG has a set of configurations and is played for a single configuration. Edges are guarded by sets of configurations; if the VPG is played for a configuration that is in the guard set then the edge is enabled, otherwise it is disabled.

Definition 5.1. *A variability parity game (VPG) is a tuple* $(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$*, where:*

- V,V_0,V_1, E *and* Ω *are defined as in parity games,*
- $\mathfrak{C}$ *is a non-empty finite set of configurations,*
- $\theta : E \to 2^{\mathfrak{C}} \setminus \emptyset$ *is a total function mapping every edge to a set of configurations guarding that edge.*

VPGs are considered total when every vertex has at least one outgoing edge. Since edges are guarded with sets of configurations we also require that for every configuration $c \in \mathfrak{C}$ every vertex has at least one outgoing edge that admits configuration c, formally a VPG is total if and only if for all $v \in V$:

$$\bigcup \{\theta(v, w) \mid (v, w) \in E\} = \mathfrak{C}$$

VPGs are depicted as parity games with labelled edges that represent the sets of configurations guarding them.

Example 5.1. Figure 11 shows an example of a VPG with configuration $\mathfrak{C} = \{c_0, c_1, c_2\}$.

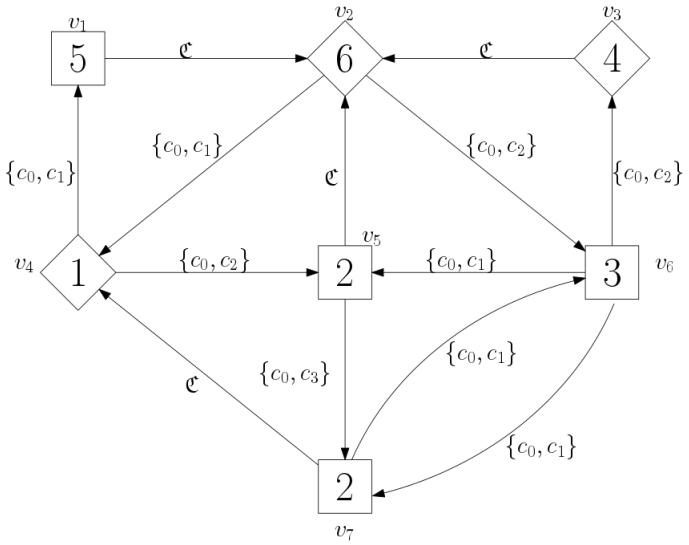


Figure 11: VPG with configurations $\mathfrak{C} = \{c_0, c_1, c_2\}$

A VPG can be played for a vertex-configuration pair. When playing a VPG for $v \in V$ and $c \in \mathfrak{C}$ we start by placing a token on vertex v. We proceed with the game similar as with a parity game however player α can only move the token from $v \in V_\alpha$ to $w \in V$ if $(v, w) \in E$ and $c \in \theta(v, w)$. Similar as in a parity game this results in a path. Again the winner is determined by the highest priority occurring infinitely often in the path or, in case of a finite path, the winner is the opponent of the player that can't make a move any more. Paths might be valid for some configurations but not valid for others, we call a path valid for configuration c if and only if for every $i > 0$ we get $(\pi_i, \pi_{i+1}) \in E$ and $c \in \theta(\pi_i, \pi_{i+1})$.

Moves made by the players can again be determined by strategies, however for different configurations for which the game is played different strategies might be needed. So we define a strategy not only for a player but also for a configuration. We define a strategy for player α and configuration $c \in \mathfrak{C}$ as a partial function $\sigma^c_\alpha : V^* V_\alpha \to V$ that maps a series of vertices ending with a vertex owned by player α to the next vertex such that for any $\sigma^c_\alpha(w_0 \dots w_m) = w$ we have $(w_m, w) \in E$ and $c \in \theta(w_m, w)$. A path π conforms to strategy σ^c_α if for every $i > 0$ with $\pi_{i-1} \in V_\alpha$ we have $\pi_i = \sigma_\alpha(\pi_0 \pi_1 \dots \pi_{i-1})$.

A strategy is winning in configuration c for player α from vertex v if and only if α is the winner of every valid path for c starting in v that conforms to σ_α.

Example 5.2. Consider the VPG in Figure 11. When playing the game for vertex v_5 and configuration c_0 we can define strategy

$$\sigma_1^{c_0} = \{v_5 \mapsto v_7, v_7 \mapsto v_6, v_6 \mapsto v_7, \dots\}$$

This always results in the path $v_5(v_7 v_6)\omega$ where the highest priority occurring infinitely often is 3 so player 1 wins. Since this is the only valid path vertex v_5 is won by player 1 in configuration c_0.

If the game is played for vertex v_5 and configuration c_1 the strategy $\sigma_1^{c_0}$ is not valid because the edge (v_5, v_7) is not enabled. For player 0 we can define strategy

$$\sigma_0^{c_1} = \{v_2 \mapsto v_4, v_4 \mapsto v_1, \dots\}$$

Player 1 can only play from v_5 to v_2 so the only path that conforms to $\sigma_0^{c_1}$ is $v_5(v_2 v_4 v_1)^\omega$ which is winning for player 0. So vertex v_5 is won by player 0 in configuration c_1.

When playing the game for vertex v_5 and configuration c_2 the we use the same strategy as used in configuration c_1, we get $\sigma_1^{c_2} = \sigma_1^{c_0}$ and vertex v_5 is winning for player 1 in configuration c_2.

A VPG is solved if for every configuration c the vertices are partitioned in two sets, namely W_0^c and W_1^c, such that every vertex in W_α^c is winning for player α in configuration c. We call these sets the winning sets of a VPG.

We can create a parity game from a VPG by simply choosing configuration c and removing all the edges that do not have c in their guard set. We call this a projection.

Definition 5.2. *The projection of VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ onto configuration $c \in \mathfrak{C}$, denoted by $G_{|c}$, is the parity game $(V, V_0, V_1, E', \Omega)$ where $E' = \{e \in E \mid c \in \theta(e)\}$.*

If a VPG is total then there is at least one outging edge for every vertex that admits configuration $c \in \mathfrak{C}$. This edge will be in the projection $G_{|c}$ so clearly when the VPG is total then its projections are also total.

A VPG contains multiple parity games, in fact playing a VPG G for configuration c is the same as playing the parity game $G_{|c}$ which we show in the following lemma's and theorem.

Lemma 5.1. *Path π is valid in $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ for configuration c if and only if path π is valid in $G_{|c} = (V, V_0, V_1, E', \Omega)$.*

Proof. Consider path π that is valid in G for configuration c. For every $i > 0$ we have $(\pi_{i-1}, \pi_i) \in E$ and $c \in \theta(\pi_{i-1}, \pi_i)$. Using the projection definition (Definition 5.2) we can conclude that $(\pi_{i-1}, \pi_1) \in E'$ making the path valid in $G_{|c}$.

Consider path π that is valid in $G_{|c}$. For every $i > 0$ we have $(\pi_{i-1}, \pi_i) \in E'$. Given the projection definition we find that because $(\pi_{i-1}, \pi_i) \in E'$ we must have $(\pi_{i-1}, \pi_i) \in E$ and $c \in \theta(\pi_{i-1}, \pi_i)$. This makes the path valid in G for configuration c. □

Lemma 5.2. *Any strategy σ^c_α for player α and configuration $c \in \mathfrak{C}$ in VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ is also a strategy in $G_{|c}$ for player α and any strategy σ_α for player α in $G_{|c}$ is also a strategy in G for player α and configuration c.*

Proof. The exact same reasoning as in lemma 5.1 can be applied to prove this lemma. □

Theorem 5.3. *Winning sets (W_0^c, W_1^c) of VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ played for configuration $c \in \mathfrak{C}$ are equal to winning sets (Q_0, Q_1) of parity game $G_{|c}$.*

Proof. Let $v \in W_\alpha^c$ for some $\alpha \in \{0, 1\}$. There exists a strategy σ_α^c in VPG G for player α and configuration c such that any valid path starting in v is winning for player α. As shown in lemma 5.2, σ_α^c is also a strategy for player α in $G_{|c}$. Any valid path starting in v in game G played for configuration c is also valid in game G as shown in lemma 5.1, additionally any path valid in $G_{|c}$ is also valid in G played for c. Assume there is a valid path in $G_{|c}$ that conforms to σ_α^c starting from v that is not won by player α. This path is also valid in G played for configuration c which contradicts $v \in W_\alpha^c$, therefore no such path exists and strategy σ_α^c is winning for player α from v in parity game $G_{|c}$, hence $v \in Q_\alpha$.

Let $v \in Q_\alpha$ for some $\alpha \in \{0, 1\}$. There exists a strategy σ_α in parity game $G_{|c}$ for player α such that any valid path starting in v is winning for player α. Using lemma 5.2 we find that σ_α is a strategy in game G for player α and configuration c. Assume there is a valid path in G for c that conforms to σ_α starting from v that is not won by player α. This path is also valid in $G_{|c}$ which contradicts $v \in Q_\alpha$, therefore no such path exists and strategy σ_α is winning for player α and configuration c from v in VPG G, hence $v \in W_\alpha^c$. □

Parity games have a unique winner for every vertex, from theorem 5.3 we can conclude that a VPG player for a configuration also has a unique winner for every vertex. Moreover since it is decidable who wins a vertex in a parity game it is also decidable who wins a vertex in a VPG for configuration c. Finally, in a parity game there exists a positional strategy for player α that is winning for all the vertices won by player α in the game. In theorem 5.3 we argued that a strategy that is winning for player α starting in vertex v in a projection of G onto c is also winning in G for player α and configuration c starting in vertex v. So we can conclude that VPGs are also positionally determined and we can consider a strategy for player α and configuration c as a total function $\sigma_\alpha^c : V_\alpha \to V$.

5.1 Verifying featured transition systems

Given an LTS and a modal μ-calculus formula we can construct a parity game such that solving this parity game tells us if the LTS satisfies the formula. Similarly we can construct a VPG from an FTS and a modal μ-calculus in such a way that solving the VPG tells us what products satisfy the formula.

We create a VPG from an FTS by choosing the set of configurations to be equal to the set of products in the FTS. The game graph is created similar as to how a parity game is created from an LTS. Finally we guard edges going from vertices representing a modal formula.

Definition 5.3. *FTS2VPG(M, φ) converts FTS $M = (S, Act, trans, s_0, N, P, \gamma)$ and closed formula φ to VPG $(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$.*

The set of configurations is equal to the set of products, ie. $\mathfrak{C} = P$.

Vertices are created for every state with every formula in the Fischer-Ladner closure of φ. We define the set of vertices:

$$V = (S \times FL(\varphi))$$

The following table shows the owner, successors and edge guards of vertices where we write $w \mid C$ as a successor of v to denote that there is an edge $(v, w) \in E$ such that the edge is guarded by set $C \subseteq \mathfrak{C}$, i.e. $\theta(v, w) = C$.

Vertex	*Owner*	*Successor* | *guard set*
$(s, \bot)$	*0*	
$(s, \top)$	*1*	
$(s, \psi_1 \vee \psi_2)$	*0*	$(s, \psi_1) \mid \mathfrak{C}$ *and* $(s, \psi_2) \mid \mathfrak{C}$
$(s, \psi_1 \wedge \psi_2)$	*1*	$(s, \psi_1) \mid \mathfrak{C}$ *and* $(s, \psi_2) \mid \mathfrak{C}$
$(s, \langle a \rangle \psi)$	*0*	$(s', \psi) \mid \{c \in \mathfrak{C} \mid c \models g\}$ *for every* $s \xrightarrow{a \mid g} s'$
$(s, [a] \psi)$	*1*	$(s', \psi) \mid \{c \in \mathfrak{C} \mid c \models g\}$ *for every* $s \xrightarrow{a \mid g} s'$
$(s, \mu X.\psi)$	*1*	$(s, \psi[X := \mu X.\psi]) \mid \mathfrak{C}$
$(s, \nu X.\psi)$	*1*	$(s, \psi[X := \nu X.\psi]) \mid \mathfrak{C}$

Since the Fischer-Ladner formula's are closed we never get a vertex (s, X).

Finally we have $\Omega(v) = \begin{cases} 2\lfloor adepth(X)/2 \rfloor & \text{if } v = (s, \nu X.\psi) \\ 2\lfloor adepth(X)/2 \rfloor + 1 & \text{if } v = (s, \mu X.\psi) \\ 0 & \text{otherwise} \end{cases}$

Similar to parity games a VPG can be made total by creating sink vertices l_0 and l_1 with priority 1 and 0 respectively and each having an edge to itself with guard set $\mathfrak{C}$. When the VPG is played for configuration c and the token ends up in l_α then clearly player α looses. We make a VPG total by adding vertices l_0 and l_1 and adding an edge from every vertex $v \in V_\alpha$ that has $\bigcup\{\theta(v,w) \mid (v,w) \in E\} \neq \mathfrak{C}$ to l_α with guard set $\mathfrak{C} \setminus \bigcup\{\theta(v,w) \mid (v,w) \in E\}$. Any vertex v_α where player α could not have made a move in the original game played for configuration c now has an edge admitting c to l_α where player α still looses. An edge admitting c is only added if there was no outgoing edge admitting c so the winner of vertex v for configuration c in the original game is the same as in the total game.

Example 5.3. Consider FTS M as shown in Figure 12 that has features f and g and products $\{\emptyset, \{f\}, \{f,g\}\}$. Modal μ-calculus formula $\varphi = \mu X.([a]X \vee \langle b \rangle \top)$ expresses that on any path reached by a's we can eventually do a b action. This holds true for product $\{\emptyset\}$ because s_1 can only go to s_2 where b can always be done. For product $\{f\}$ this doesn't hold because ones in s_1 it is possible to stay in s_1 indefinitely through an a transition. For product $\{f,g\}$ however the formula does hold because we can indeed stay in s_1 indefinitely however from s_1 we can always do a b step.

Figure 13 shows the VPG resulting from $FTS2VPG(M, \varphi)$ made total using sink vertices l_0 and l_1. Products $\{\emptyset\}, \{f\}, \{f,g\}$ are depicted as configurations c_0, c_1, c_2 respectively.

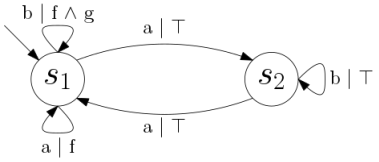


Figure 12: FTS M

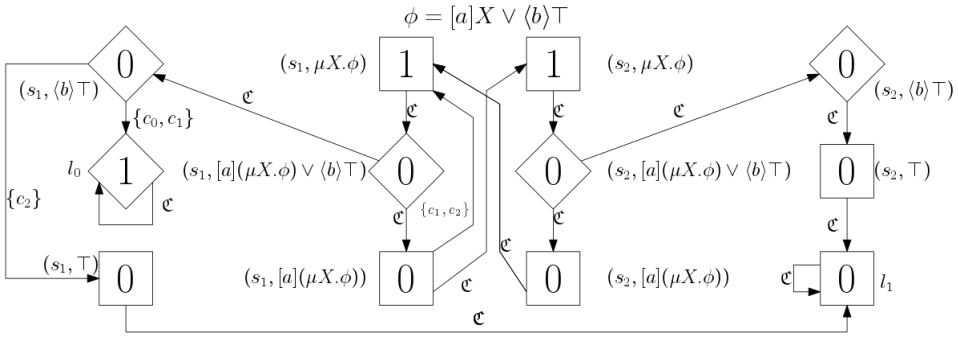
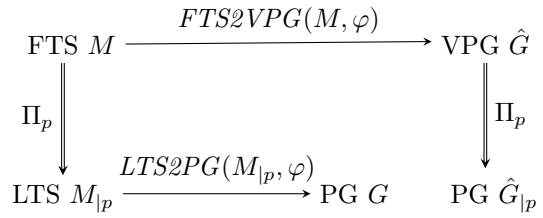


Figure 13: Total VPG created by *FTS2VPG*

In order to prove that solving a VPG created by *FTS2VPG* can be used to model check an FTS we inspect the relations we have seen between FTSs, LTSs, parity games and VPGs. Given FTS M and formula φ we can project M onto a product to create an LTS and create a parity game from the resulting LTS and φ using *LTS2PG*. Alternatively we can create a VPG from M and φ using *FTS2VPG* which can be projected onto a configuration to get a parity game. These different transformations are shown in the following diagram, where Π_p depicts a projection onto product p or configuration p:



In the following lemma we prove that in fact parity game G and $\hat{G}_{|p}$ are identical.

Lemma 5.4. *Given FTS $M = (S, Act, trans, s_0, N, P, \gamma)$, closed modal μ-calculus formula φ and product $p \in P$ it holds that parity games $LTS2PG(M_{|p}, \varphi)$ and $FTS2VPG(M, \varphi)_{|p}$ are identical.*

Proof. Let $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$ be the VPG created from $FTS2VPG(M, \varphi)$ and let $G = (V, V_0, V_1, E, \Omega)$ be the parity game created from $LTS2PG(M_{|p}, \varphi)$. Let the projection of $\hat{G}$ onto p (using Definition 5.2) be the parity game $G_{|p} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}', \hat{\Omega})$. We will prove that $\hat{G}_{|p} = G$.

First observe that when an FTS is projected onto a product (using Definition 3.2) the FTS has the same states as the projection, we find that FTS M has the same states as $M_{|p}$. The vertices created by *LTS2PG* and *FTS2VPG* rely only on the formula and the states in the LTS and FTS respectively. Similarly the owner and priority of these vertices is only determined by the states and the formula. Given that M and $M_{|p}$ have the same states we find that $\hat{V} = V$, $\hat{V}_0 = V_0$, $\hat{V}_1 = V_1$ and $\hat{\Omega} = \Omega$.

We are left with showing $\hat{E}' = E$ in order to conclude $\hat{G}_{|p} = G$. Consider vertex v, we distinguish two cases.

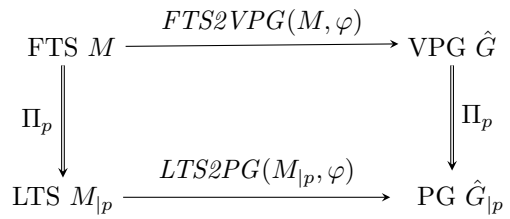
Let $v = (s, \langle a \rangle \psi)$ or $v = (s, [a]\psi)$. If v has a successor to (s', ψ) in G then we have $s \xrightarrow{a} s'$ in $M_{|p}$ and therefore $s \xrightarrow{a \mid f} s'$ with $p \models f$ in M. Using the *FTS2VPG* definition we find that vertex v in $\hat{G}$ has successor (s', ψ) with a guard containing p. Since p is in the guard set we also find this successor in the projection $\hat{G}_{|p}$.

If v has a successor to (s', ψ) in $\hat{G}_{|p}$ then in $\hat{G}$ the edge from v to (s', ψ) also exists and the set guarding it contains p. In M we find $s \xrightarrow{a \mid g} s'$ with $p \models g$, therefore we find $s \xrightarrow{a} s'$ in $M_{|p}$. Using the *LTS2PG* definition we find that vertex v in G has successor (s', ψ).

Let $v \neq (s, \langle a \rangle \psi)$ and $v \neq (s, [a]\psi)$. Any successor of v created by *LTS2PG* does not depend on the LTS but only on the formula. Similarly any successor of v created by *FTS2VPG* does not depend on the FTS and has guard set $\mathfrak{C}$. The two definitions create the same successors for v so the successors in games G and $\hat{G}$ are the same. Since the guard sets of these successors are always $\mathfrak{C}$ the successors are also the same in $\hat{G}_{|p}$.

We have proven $\hat{E}' = E$ and therefore $\hat{G}_{|p} = G$. □

Using this Lemma we get the following diagram showing the relation between FTSs, LTSs, parity games and VPGs.



We know from existing theory that solving a parity game constructed using *LTS2PG* can be used to model check an LTS, furthermore we have seen that the winning sets of an VPG for configuration c are equal to the winning sets of that VPG projected onto c. Given these facts and the lemma above we can prove that VPGs can be used to model check FTSs.

Theorem 5.5. *Given:*

- *FTS $M = (S, Act, trans, s_0, N, P, \gamma)$,*

- *closed modal μ-calculus formula φ,*
- *product $p \in P$ and*
- *state $s \in S$*

it holds that $(M_{|p}, s) \models \varphi$ if and only if $(s, \varphi) \in W_0^p$ in FTS2VPG(M, φ).

Proof. Assume $(M_{|p}, s) \models \varphi$, using the relation between LTSs and parity games (Theorem 2.2) we find that vertex (s_0, φ) in parity game LTS2PG$(M_{|p}, \varphi)$ is won by player 0. Using Lemma 5.4 we find that vertex (s_0, φ) is also in game FTS2VPG$(M, \varphi)_{|p}$ and is also won by player 0. Using Theorem 5.3 we find that the winning sets of a VPG for configuration c are the same as the winning sets of the projection of the VPG onto c. We find that vertex (s_0, φ) is winning in game FTS2VPG(M, φ) for configuration p, hence $(s, \varphi) \in W_0^p$.

Similarly if $(M_{|p}, s) \not\models \varphi$ vertex (s_0, φ) is won by player 1 in parity game LTS2PG$(M_{|p}, \varphi)$ and we get $(s, \varphi) \notin W_0^p$. □

Example 5.4. Again consider Example 5.3. We argued that M satisfies φ for products $\{\emptyset\}$ and $\{f, g\}$. We see that in VPG in Figure 13 that $(s_1, \mu X.([a]X \vee \langle b\rangle\top))$ is indeed winning for player 0 when played for $\{\emptyset\} = c_0$ using the strategy

$$\sigma_0^{c_0} = \{(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top) \mapsto (s_1, [a](\mu X.\phi)), (s_2, [a](\mu X.\phi) \vee \langle b\rangle\top) \mapsto (s_2, \langle b\rangle\top), \dots\}$$

Using this strategy play always ends up in l_1 which is winning for player 0.

For product $\{f\} = c_1$ player 1 wins using strategy

$$\sigma_1^{c_1} = \{(s_1, [a](\mu X.\phi)) \mapsto (s_1, \mu X.\psi), \dots\}$$

Using this strategy we either infinitely often visit $(s_1, \mu X.\psi)$ in which case player 1 wins or player 0 can decide to play to $(s_1, \langle b\rangle\top)$ in which case play ends in l_0 and player 1 wins.

For product $\{f, g\} = c_2$ player 0 wins using strategy

$$\sigma_0^{c_2} = \{(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top) \mapsto (s_1, \langle b\rangle\top), (s_1, \langle b\rangle\top) \mapsto (s_1, \top), \dots\}$$

Using this strategy player 0 can prevent the path infinitely often visiting $(s_1, \mu X.\psi)$ by playing to $(s_1, \langle b\rangle\top)$ and to $(s_1, \top)$ next which brings the play in l_1 winning it for player 0.

We conclude by showing visualizing the verification of an FTS in figure 14.

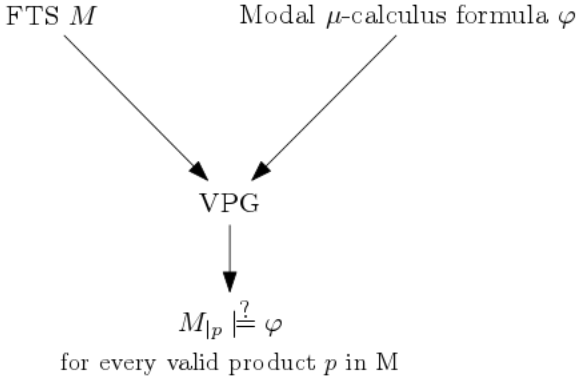


Figure 14: FTS verification using VPG

6 Solving variability parity games

In this section we inspect methods so solve VPGs, for convenience we assume that VPGs are total. We distinguish two general approaches for solving VPGs, the first approach is to simply project the VPG to the different configurations and solve all the resulting parity games independently. We call this *independently* solving a VPG, existing parity game algorithms can be used in this approach. Alternatively we solve the VPG *collectively*, where a VPG is solved in its entirety and similarities between the configurations are used to improve performance.

In the next sections we explore collective algorithms and analyse their time complexity. We aim to solve VPGs originating from model verification problems, such VPGs generally have certain properties that a completely random VPG might not have. In general parity games originating from model verification problems have a relatively low number of distinct priorities compared to the number of vertices because new priorities are only introduced when fixed points are nested in the μ-calculus formula. Furthermore the transition guards of featured transition systems are expressed over features. In general these transition guards will be quite simple, specifically excluding or including a small number of features.

In this section we analyse time complexities of PGs are VPGs, we use n to denote the number of vertices, e the number of edges, and d the number of distinct priorities. When analysing a VPG then we also use c to indicate the number of configurations.

6.1 Recursive algorithm for variability parity games

We can use the original Zielonka's recursive algorithm to solve VPGs, to do so we will first consider a method of creating a parity game from a VPG, called *unification*.

6.1.1 Unified parity games

We can create a PG from a VPG by taking all the projections of the VPG, which are PGs, and combining them into one PG by taking the union of them. We call the resulting PG the *unification* of the VPG. A parity game that is the result of a unification is called a *unified PG*, also any total subgame of it will be called a unified PG. A unified PG always has a VPG from which it originated.

Definition 6.1. *Given VPG $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$ we define the unification of $\hat{G}$, denoted as $\hat{G}_\downarrow$, as*

$$\hat{G}_\downarrow = \biguplus_{c \in \mathfrak{C}} \hat{G}_{|c}$$

where the union of two PGs is defined as

$$(V, V_0, V_1, E, \Omega) \uplus (V', V_0', V_1', E', \Omega') = (V \uplus V', V_0 \uplus V_0', V_1 \uplus V_1', E \uplus E', \Omega \uplus \Omega')$$

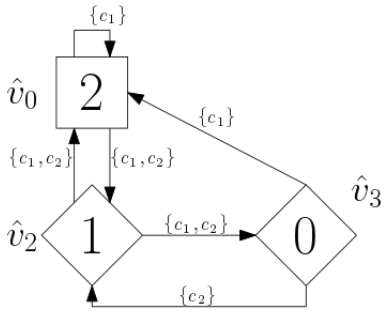
In this section we use the hat decoration $(\hat{G}, \hat{V}, \hat{E}, \hat{\Omega}, \hat{W})$ when referring to a VPG and use no hat decoration when referring to a (unified) PG.

Every vertex in game $\hat{G}_\downarrow$ originates from a configuration and an original vertex. Therefore we can consider every vertex in a unification as a pair consisting of a vertex and a configuration, ie. $V = \mathfrak{C} \times \hat{V}$. We can consider edges in a unification similarly, so $E \subseteq (\mathfrak{C} \times \hat{V}) \times (\mathfrak{C} \times \hat{V})$. Note that edges don't cross configurations so for every $((c, \hat{v}), (c', \hat{v}')) \in E$ we have $c = c'$. Figure 15 shows an example of a unification.

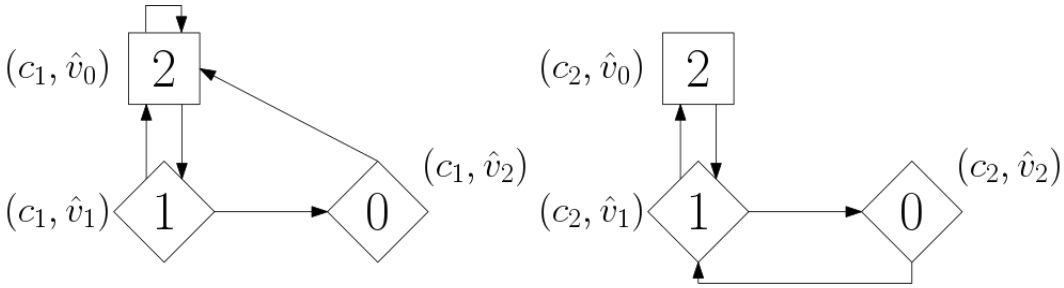
If we solve the PG that is the unification of a VPG we have solved the VPG, as shown in the following theorem.

Theorem 6.1. *Given*

- *VPG $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$,*
- *some configuration $c \in \mathfrak{C}$,*
- *winning sets $\hat{W}_0^c$ and $\hat{W}_1^c$ for game $\hat{G}$ and*
- *winning sets W_0 and W_1 for game $\hat{G}_\downarrow$*



(a) VPG consisting of 2 configurations



(b) Unified PG, created from unifying the two projections

Figure 15: A VPG with its corresponding unified PG

it holds that

$$(c, \hat{v}) \in W_\alpha \iff \hat{v} \in \hat{W}^c_\alpha, \textit{ for } \alpha \in \{0, 1\}$$

Proof. The bi-implication is equal to the following to implications.

$$(c, \hat{v}) \in W_\alpha \implies \hat{v} \in \hat{W}^c_\alpha, \text{ for } \alpha \in \{0, 1\}$$

and

$$(c, \hat{v}) \notin W_\alpha \implies \hat{v} \notin \hat{W}^c_\alpha, \text{ for } \alpha \in \{0, 1\}$$

Since the winning sets partition the game we have $\hat{v} \notin \hat{W}^c_\alpha \implies \hat{v} \in \hat{W}^c_{\overline{\alpha}}$ (similar for set W). Therefore it is sufficient to prove only the first implication.

Let $(c, \hat{v}) \in W_\alpha$, player α has a strategy to win game $\hat{G}_\downarrow$ from vertex $(c, \hat{v})$. Since $\hat{G}_\downarrow$ is the union of all the projections of $\hat{G}$ we can apply the same strategy to game $\hat{G}_{|c}$ to win vertex $\hat{v}$ as player α. Because we can win $\hat{v}$ in the projection of $\hat{G}$ to c we have $\hat{v} \in \hat{W}^c_\alpha$. □

6.1.2 Solving unified parity games

A unified PG can be solved using Zielonka's recursive algorithm, since it is a total parity game. The algorithm revolves around the attractor operation. Consider the example presented in figure 15. Vertices with the highest priority are

$$\{(c_1, \hat{v}_0), (c_2, \hat{v}_0)\}$$

attracting these for player 0 gives the set

$$\begin{aligned}\{&(c_1, \hat{v}_0),(c_2, \hat{v}_0),\\ &(c_1, \hat{v}_1),(c_2, \hat{v}_1),\\ &(c_2, \hat{v}_2)\}\end{aligned}$$

The algorithm tries to attract vertices $(c_1, \hat{v}_1)$ and $(c_2, \hat{v}_1)$ because they have edges to $\{(c_1, \hat{v}_0), (c_2, \hat{v}_0)\}$. So the algorithm in this case asks the questions: "Can vertices $(c_1, \hat{v}_1)$ and $(c_2, \hat{v}_1)$ be attracted?" We could also

ask the question: "For which configurations can we attract origin vertex $\hat{v}_1$?" Since the vertices in unified parity games are pairs of configurations and origin vertices we can instead of considering vertices individually consider origin vertices and try to attract as much configurations as possible for each origin vertex. This is the idea for the collective recursive VPG we present next, we modify the recursive algorithm to solve unified PGs such that it tries to attract multiple configurations per origin vertex simultaneously.

6.1.3 Representing unified parity games

In order to create an algorithm based around attracting multiple configurations we need to take a look at how unified parity games can be represented.

Unified PGs have a specific structure because they are the union of PGs that have the same vertices with the same owner and priority. Because they have the same priority we don't actually need to create a new function that is the unification of all the projections, we can simply use the original priority assignment function because the following relation holds:

$$\Omega(c, \hat{v}) = \hat{\Omega}(\hat{v})$$

Similarly we can use the original partition sets $\hat{V}_0$ and $\hat{V}_1$ instead of having the new partition V_0 and V_1 because the following relations holds:

$$(c, \hat{v}) \in V_0 \iff \hat{v} \in \hat{V}_0$$
$$(c, \hat{v}) \in V_1 \iff \hat{v} \in \hat{V}_1$$

So instead of considering unified PG (V, V_0, V_1, E, Ω) we will consider $(V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$.

Next we consider how we represent vertices and edges in a unified PG. A set $X \subseteq (\mathfrak{C} \times \hat{V})$ can be represented as a complete function $f : \hat{V} \to 2^{\mathfrak{C}}$. The set X and function f are equivalent, denoted by the operator $=_\lambda$, iff the following relation holds:

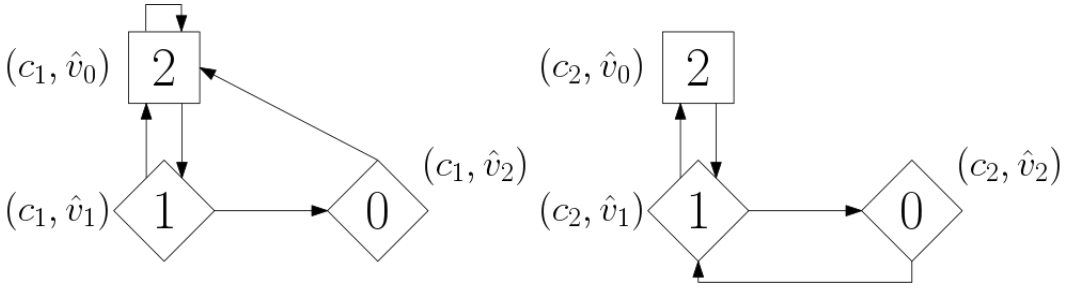
$$(c, \hat{v}) \in X \iff c \in f(\hat{v})$$

We can also represent edges as a complete function $f : \hat{E} \to 2^{\mathfrak{C}}$. The set E and function f are equivalent, denoted by the operator $=_\lambda$, iff the following relation holds:

$$((c, \hat{v}), (c, \hat{v}')) \in E \iff c \in f(\hat{v}, \hat{v}')$$

We define $\lambda^{\emptyset}$ to be the function that maps every element to $\emptyset$, clearly $\lambda^{\emptyset} =_\lambda \emptyset$. We call using a set of pairs to represent vertices and edges a *set-wise* representation and using functions a *function-wise* representation.

Next we consider a few examples of (sub)games and show their set-wise and function-wise representation. First reconsider the following unified parity game.



This game can be represented set-wise:

$$V = \{(c_1, \hat{v}_0), (c_2, \hat{v}_0), (c_1, \hat{v}_1), (c_2, \hat{v}_1), (c_1, \hat{v}_2), (c_2, \hat{v}_2)\}$$
$$E = \{((c_1, \hat{v}_0), (c_1, \hat{v}_0)), ((c_1, \hat{v}_0), (c_1, \hat{v}_1)), ((c_1, \hat{v}_1), (c_1, \hat{v}_0)), ((c_1, \hat{v}_1), (c_1, \hat{v}_2)), ((c_1, \hat{v}_2), (c_1, \hat{v}_0)),$$
$$((c_2, \hat{v}_0), (c_2, \hat{v}_1)), ((c_2, \hat{v}_1), (c_2, \hat{v}_0)), ((c_2, \hat{v}_1), (c_2, \hat{v}_2)), ((c_2, \hat{v}_2), (c_2, \hat{v}_1))\}$$

and function-wise:

$$
\begin{aligned}
V &= \{\hat{v}_0 \mapsto \{c_1, c_2\}, \hat{v}_1 \mapsto \{c_1, c_2\}, \hat{v}_2 \mapsto \{c_1, c_2\}\} \\
E &= \{(\hat{v}_0, \hat{v}_1) \mapsto \{c_1, c_2\}, (\hat{v}_1, \hat{v}_0) \mapsto \{c_1, c_2\}, (\hat{v}_1, \hat{v}_2) \mapsto \{c_1, c_2\}, \\
&\quad (\hat{v}_0, \hat{v}_0) \mapsto \{c_1\}, \\
&\quad (\hat{v}_2, \hat{v}_0) \mapsto \{c_1\}, \\
&\quad (\hat{v}_2, \hat{v}_1) \mapsto \{c_2\}\}
\end{aligned}
$$

Consider the following subgame:

This subgame can be represented set-wise:

$$
\begin{aligned}
V &= \{(c_1, \hat{v}_0), (c_2, \hat{v}_0), (c_2, \hat{v}_1)\} \\
E &= \{((c_1, \hat{v}_0), (c_1, \hat{v}_0)), \\
&\quad ((c_2, \hat{v}_0), (c_2, \hat{v}_1)), ((c_2, \hat{v}_1), (c_2, \hat{v}_0))\}
\end{aligned}
$$

and function-wise:

$$
\begin{aligned}
V &= \{\hat{v}_0 \mapsto \{c_1, c_2\}, \hat{v}_1 \mapsto \{c_2\}, \hat{v}_2 \mapsto \emptyset\} \\
E &= \{(\hat{v}_0, \hat{v}_1) \mapsto \{c_2\}, (\hat{v}_1, \hat{v}_0) \mapsto \{c_2\}, (\hat{v}_1, \hat{v}_2) \mapsto \emptyset, \\
&\quad (\hat{v}_0, \hat{v}_0) \mapsto \{c_1\}, \\
&\quad (\hat{v}_2, \hat{v}_0) \mapsto \emptyset, \\
&\quad (\hat{v}_2, \hat{v}_1) \mapsto \emptyset\}
\end{aligned}
$$

Finally consider an empty subgame which we can represent set-wise:

$$V = \emptyset, E = \emptyset$$

and function-wise:

$$V = \lambda^{\emptyset}, E = \lambda^{\emptyset}$$

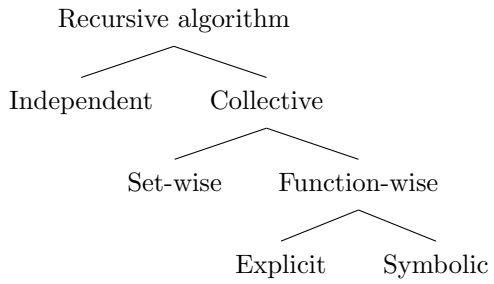
Projections and totality A unified PG can be projected back to one of the games from which it is the union.

Definition 6.2. *The projection of unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ to configuration c, denoted as $G_{|c}$, is the parity game $(V', \hat{V}_0, \hat{V}_1, E', \hat{\Omega})$ such that $V' = \{\hat{v} \mid (c, \hat{v}) \in V\}$ and $E' = \{(\hat{v}, \hat{w}) \mid ((c, \hat{v}), (c, \hat{w})) \in E\}$.*

One of the properties of a PG is its totality; a game is total if every vertex has at least one outgoing vertex. VPGs are also total, meaning that every vertex has, for every configuration $c \in \mathfrak{C}$, at least one outgoing vertex admitting c. Because VPGs are total their unifications are also total. Since edges in a unified PG don't cross configurations we can conclude that a unified PG is total if and only if every projection is total.

6.1.4 Algorithms

Using the recursive algorithm as a basis we can solve a VPG in numerous ways. First of wall we can solve the projections, ie. solve the VPG independently. Alternatively we can solve it collectively using a set-wise presentation or a function-wise presentation. For the function-wise presentation we are working with functions mapping vertices and edges to sets of configurations. These sets of configurations can either be represented explicitly or symbolically. The following diagram shows the different algorithms:



The independent approach uses the original algorithm, the collective set-wise approach also uses the original algorithm applied to a unified PG. The function-wise representation requires modifications to the algorithm, as we try to attract multiple configurations at the same time. As we will discuss later, this modified algorithm relies heavily on set operations over sets of configurations.

Symbolically representing sets of configurations For VPGs originating from an FTS the configuration sets are boolean functions over the features. The formula's guarding the edges in the VPG will generally have relatively simple boolean functions, therefore they are specifically appropriate to represent as BDDs.

A set operation over two explicit sets can be performed in $O(n)$ where n is the maximum size of the sets, this is better than the time complexity of a set operation using BDDs ($O(n^2)$). However if the BDDs are small then the set size can still be large but the set operations are performed very quickly. This is a trade-off between worst-case time complexity and actual running time; using a symbolic representation might yield better results if the sets are structured in such a way that the BDDs are small, however its worst-case time complexity will be worse.

We hypothesize that since the collective function-wise symbolic recursive algorithm relies heavily on set operations over sets of configurations this algorithm will perform well when solving VPGs originating from FTSs.

A note on symbolically solving games The function-wise algorithm has two variants: an explicit and a symbolic variant. In the explicit variant everything is represented explicitly. In the symbolic variant the sets of configurations are represented symbolically, however the graph is still represented explicitly so the algorithm is partially symbolic and partially explicit. Alternatively an algorithm could completely work symbolically by representing both the graph and the sets of configurations symbolically.

Solving parity games symbolically has been studied in [15]. The obstacle is that representing graphs with a large number of nodes makes the corresponding BDDs very complex and performance decreases rapidly. As to not repeat work done in [15] we only consider algorithms where we represent the graph explicitly.

6.1.5 Recursive algorithm using a function-wise representation

We can modify the recursive algorithm to work with the function-wise representation of vertices and edges. The algorithm behaves the same, only operations are modified to work with the different representation. Pseudo code for the modified algorithm is presented in algorithm 3. Note that for this pseudo code no distinction is needed between explicit and symbolic representations of sets of configurations.

We introduce a modified attractor definition to work with the function-wise representation.

Algorithm 3 RecursiveUPG($PG\ G = (V : \hat{V} \to 2^{\mathfrak{C}}, \hat{V}_0 \subseteq \hat{V}, \hat{V}_1 \subseteq \hat{V}, E : \hat{E} \to 2^{\mathfrak{C}}, \hat{\Omega} : \hat{V} \to \mathbb{N})$)

$m \leftarrow \min\{\hat{\Omega}(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
$h \leftarrow \max\{\hat{\Omega}(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
if $h = m$ or $V = \lambda^{\emptyset}$ **then**
    if h is even or $V = \lambda^{\emptyset}$ **then**
        return $(V, \lambda^{\emptyset})$
    else
        return $(\lambda^{\emptyset}, V)$
    end if
end if
$\alpha \leftarrow 0$ if h is even and 1 otherwise
$U \leftarrow \lambda^{\emptyset}$, $U(\hat{v}) \leftarrow V(\hat{v})$ for all $\hat{v}$ with $\hat{\Omega}(\hat{v}) = h$
$A \leftarrow \alpha$-*FAttr*(G, U)
$(W_0', W_1') \leftarrow$ RecursiveUPG$(G \backslash A)$
if $W'_{\overline{\alpha}} = \lambda^{\emptyset}$ **then**
    $W_{\alpha} \leftarrow A \cup W'_{\alpha}$
    $W_{\overline{\alpha}} \leftarrow \lambda^{\emptyset}$
else
    $B \leftarrow \overline{\alpha}$-*FAttr*$(G, W'_{\overline{\alpha}})$
    $(W_0'', W_1'') \leftarrow$ RecursiveUPG$(G \backslash B)$
    $W_{\alpha} \leftarrow W''_{\alpha}$
    $W_{\overline{\alpha}} \leftarrow W''_{\overline{\alpha}} \cup B$
end if
return (W_0, W_1)

Definition 6.3. *Given unified PG* $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ *and a non-empty set* $U \subseteq V$*, both represented function-wise, we define* α*-FAttr*(G, U) *such that*

$$U_0 = U$$

For $i \geq 0$*:*

$$U_{i+1}(\hat{v}) = U_i(\hat{v}) \cup \begin{cases} V(\hat{v}) \cap \bigcup_{\hat{v}'}(E(\hat{v}, \hat{v}') \cap U_i(\hat{v}')) & \text{if } \hat{v} \in \hat{V}_\alpha \\ V(\hat{v}) \cap \bigcap_{\hat{v}'}((\mathfrak{C} \backslash E(\hat{v}, \hat{v}')) \cup U_i(\hat{v}') & \text{if } \hat{v} \in \hat{V}_{\overline{\alpha}} \end{cases}$$

Finally:

$$\alpha\text{-}FAttr(G, U) = \bigcup_{i \geq 0} U_i$$

The function-wise attractor definition gives a result equal to the result of the original attractor definition as we show in the following lemma.

Lemma 6.2. *Given unified PG* $G = (\mathcal{V}, \hat{V}_0, \hat{V}_1, \mathcal{E}, \hat{\Omega})$ *and set* $\mathcal{U} \subseteq \mathcal{V}$ *the function-wise attractor* α*-FAttr*$(G, \mathcal{U})$ *is equivalent to the set-wise attractor* α*-Attr*$(G, \mathcal{U})$ *for any* $\alpha \in \{0, 1\}$*.*

Proof. Let V, E, U be the set-wise representation and $V^\lambda, E^\lambda, U^\lambda$ be the function-wise representation of $\mathcal{V}, \mathcal{E}, \mathcal{U}$ respectively.

The following properties hold by definition:

$$(c, \hat{v}) \in V \iff c \in V^\lambda(\hat{v})$$

$$(c, \hat{v}) \in U \iff c \in U^\lambda(\hat{v})$$

$$((c, \hat{v}), (c, \hat{v}')) \in E \iff c \in E^\lambda(\hat{v}, \hat{v}')$$

Since the attractors are inductively defined and $U_0 =_\lambda U_0^\lambda$ (because $U =_\lambda U^\lambda$) we have to prove that for some $i \geq 0$, with $U_i =_\lambda U_i^\lambda$, we have $U_{i+1} =_\lambda U_{i+1}^\lambda$, which holds iff:

$$(c, \hat{v}) \in U_{i+1} \iff c \in U_{i+1}^\lambda(\hat{v})$$

Let $(c, \hat{v}) \in V$ (and therefore $c \in V^\lambda(\hat{v})$), we consider 4 cases.

- Case: $\hat{v} \in \hat{V}_\alpha$ and $(c, \hat{v}) \in U_{i+1}$:
  To prove: $c \in U_{i+1}^\lambda(\hat{v})$.

  If $(c, \hat{v}) \in U_i$ then $c \in U_i^\lambda(\hat{v})$ and therefore $c \in U_{i+1}^\lambda(\hat{v})$. If $(c, \hat{v}) \notin U_i$ then we have $c \notin U_i^\lambda(\hat{v})$.

  Because $\hat{v} \in \hat{V}_\alpha$ and $c \in V^\lambda(\hat{v})$ we get

  $$U_{i+1}^\lambda = \bigcup_{\hat{v}'}(E^\lambda(\hat{v}, \hat{v}') \cap U_i^\lambda(\hat{v}'))$$

  There exists an $(c', \hat{v}') \in V$ such that $(c', \hat{v}') \in U_i$ and $((c, \hat{v}), (c', \hat{v}')) \in E$. Because edges don't cross configurations we can conclude that $c' = c$. Due to equivalence we have $c \in U_i^\lambda(\hat{v}')$ and $c \in E^\lambda(\hat{v}, \hat{v}')$. If we fill this in in the above formula we can conclude that $c \in U_{i+1}^\lambda(\hat{v})$.

- Case: $\hat{v} \in \hat{V}_\alpha$ and $(c, \hat{v}) \notin U_{i+1}$:
  To prove: $c \notin U_{i+1}^\lambda(\hat{v})$.

  First we observe that since $(c, \hat{v}) \notin U_{i+1}$ we get $(c, \hat{v}) \notin U_i$ and therefore $c \notin U_i^\lambda(\hat{v})$.

  Because $\hat{v} \in \hat{V}_\alpha$ and $c \in V^\lambda(\hat{v})$ we get

  $$U_{i+1}^\lambda = \bigcup_{\hat{v}'}(E^\lambda(\hat{v}, \hat{v}') \cap U_i^\lambda(\hat{v}'))$$

  Assume $c \in U_{i+1}^\lambda(\hat{v})$. There must exist a $\hat{v}'$ such that $c \in E^\lambda(\hat{v}, \hat{v}')$ and $c \in U_i^\lambda(\hat{v}')$. Due to equivalence we have a vertex $((c, \hat{v}), (c, \hat{v}')) \in E$ and $(c, \hat{v}') \in U_i$. In which case $(c, \hat{v})$ would be attracted and would be in U_{i+1} which is a contradiction.

- Case: $\hat{v} \in \hat{V}_{\overline{\alpha}}$ and $(c, \hat{v}) \in U_{i+1}$:
  To prove: $c \in U^{\lambda}_{i+1}(\hat{v})$.

  If $(c, \hat{v}) \in U_i$ then $c \in U^{\lambda}_i(\hat{v})$ and therefore $c \in U^{\lambda}_{i+1}(\hat{v})$. If $(c, \hat{v}) \notin U_i$ then we have $c \notin U^{\lambda}_i(\hat{v})$.

  Because $\hat{v} \in \hat{V}_{\overline{\alpha}}$ we get
  $$U^{\lambda}_{i+1} = V^{\lambda}(\hat{v}) \cap \bigcap_{\hat{v'}} ((\mathfrak{C} \backslash E^{\lambda}(\hat{v}, \hat{v'})) \cup U^{\lambda}_i(\hat{v'})$$

  Assume $c \notin U^{\lambda}_{i+1}(\hat{v})$. Because $c \in V^{\lambda}(\hat{v})$ there must exist an $\hat{v}$ such that
  $$c \notin ((\mathfrak{C} \backslash E^{\lambda}(\hat{v}, \hat{v'})) \text{ and } c \notin U^{\lambda}_i(\hat{v'})$$
  which is equal to
  $$c \in E^{\lambda}(\hat{v}, \hat{v'}) \text{ and } c \notin U^{\lambda}_i(\hat{v'})$$
  By equivalence we have $((c, \hat{v}), (c, \hat{v'})) \in E$ and $(c, \hat{v'}) \notin U_i$. Which means that $(c, \hat{v})$ will not be attracted and $(c, \hat{v}) \notin U_{i+1}$ which is a contradiction.

- Case: $\hat{v} \in \hat{V}_{\overline{\alpha}}$ and $(c, \hat{v}) \notin U_{i+1}$:
  To prove: $c \notin U^{\lambda}_{i+1}(\hat{v})$.

  First we observe that since $(c, \hat{v}) \notin U_{i+1}$ we get $(c, \hat{v}) \notin U_i$ and therefore $c \notin U^{\lambda}_i(\hat{v})$.

  Because $\hat{v} \in \hat{V}_{\overline{\alpha}}$ we get
  $$U^{\lambda}_{i+1} = V^{\lambda}(\hat{v}) \cap \bigcap_{\hat{v'}} ((\mathfrak{C} \backslash E^{\lambda}(\hat{v}, \hat{v'})) \cup U^{\lambda}_i(\hat{v'})$$

  Since $(c, \hat{v})$ is not attracted there must exist a $(c, \hat{v'}) \in V$ such that
  $$((c, \hat{v}), (c, \hat{v'})) \in E \text{ and } (c, \hat{v'}) \notin U_i$$
  By equivalence we have
  $$c \in E^{\lambda}(\hat{v}, \hat{v'}) \text{ and } c \notin U^{\lambda}_i(\hat{v'})$$
  Which is equal to
  $$c \notin (\mathfrak{C} \backslash E^{\lambda}(\hat{v}, \hat{v'})) \text{ and } c \notin U^{\lambda}_i(\hat{v'})$$
  From which we conclude
  $$c \notin ((\mathfrak{C} \backslash E^{\lambda}(\hat{v}, \hat{v'})) \cup U^{\lambda}_i(\hat{v'})))$$
  Therefore we have $c \notin U^{\lambda}_{i+1}(\hat{v})$.

□

We also introduce a modified subgame definition to work with the function-wise representation.

Definition 6.4. *For unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$, represented function-wise, and set $X \subseteq V$ we define the subgame $G \backslash X = (V', \hat{V}_0, \hat{V}_1, E', \hat{\Omega})$ such that:*

- $V'(\hat{v}) = V(\hat{v}) \backslash X(\hat{v})$
- $E'(\hat{v}, \hat{v'}) = E(\hat{v}, \hat{v'}) \cap V'(\hat{v}) \cap V'(\hat{v'})$

The new subgame definition gives a result equal to the original subgame definition as we show next. Note that when using the original subgame definition for unified PGs we can omit the modification to the partition because, as we have seen, we can use the partitioning from the VPG in the representation of unified PGs.

Lemma 6.3. *Given unified PG $G = (\mathcal{V}, \hat{V}_0, \hat{V}_1, \mathcal{E}, \hat{\Omega})$ and set $\mathcal{U} \subseteq \mathcal{V}$ the subgame $G \backslash \mathcal{U} = (\mathcal{V}', \hat{V}_0, \hat{V}_1, \mathcal{E}', \hat{\Omega})$ represented set-wise is equal to the subgame represented function-wise.*

Proof. Let V, V', E, E', U be the set-wise and $V^\lambda, V^{\lambda'}, E^\lambda, E^{\lambda'}, U^\lambda$ the function-wise representations of $\mathcal{V}, \mathcal{V}', \mathcal{E}, \mathcal{E}', \mathcal{U}$ respectively. We know $V =_\lambda V^\lambda$, $E =_\lambda E^\lambda$ and $U =_\lambda U^\lambda$. To prove: $V' =_\lambda V^{\lambda'}$ and $E' =_\lambda E^{\lambda'}$.

Let $(c, \hat{v}) \in V$.

If $(c, \hat{v}) \in U$ then $c \in U^\lambda(\hat{v})$, also $(c, \hat{v}) \notin V'$ (by definition 2.19) and $c \notin V^{\lambda'}(\hat{v})$ (by definition 6.4).

If $(c, \hat{v}) \notin U$ then $c \notin U^\lambda(\hat{v})$, also $(c, \hat{v}) \in V'$ (by definition 2.19) and $c \in V^{\lambda'}(\hat{v})$ (by definition 6.4).

Let $((c, \hat{v}), (c, \hat{w})) \in E$.

If $(c, \hat{v}) \in U$ then $(c, \hat{v}) \notin V'$ and $c \notin V^{\lambda'}(\hat{v})$ (as shown above). We get $((c, \hat{v}), (c, \hat{w})) \notin V' \times V'$ so $((c, \hat{v}), (c, \hat{w})) \notin E'$ (by definition 2.19). Also $c \notin E^{\lambda'}(\hat{v}, \hat{w})$ (by definition 6.4).

If $(c, \hat{w}) \in U$ then we apply the same logic.

If neither is in U then both are in V' and in $V' \times V'$ and therefore the $((c, \hat{v}), (c, \hat{w})) \in E'$. Also we get $c \in V^{\lambda'}(\hat{v})$ and $c \in V^{\lambda'}(\hat{w})$ so we get $c \in E^{\lambda'}(\hat{v}, \hat{w})$ (by definition 6.4). □

Next we prove the correctness of the algorithm by showing that the winning sets of the function-wise algorithm are equal to the winning sets of the set-wise algorithm.

Theorem 6.4. *Given unified PG $\mathcal{G} = (\mathcal{V}, \hat{V}_0, \hat{V}_1, \mathcal{E}, \hat{\Omega})$ the winning sets resulting from* RecursiveUPG($\mathcal{G}$) *ran over the function-wise representation of $\mathcal{G}$ is equal to the winning sets resulting from* RecursivePG($\mathcal{G}$) *ran over the set-wise representation of $\mathcal{G}$.*

Proof. Let $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ be the set-wise representation of $\mathcal{G}$ and $G^\lambda = (V^\lambda, \hat{V}_0, \hat{V}_1, E^\lambda, \hat{\Omega})$ be the function-wise representation of $\mathcal{G}$.

Proof by induction on $\mathcal{G}$.

Base When there are no vertices then RecursiveUPG(G^λ) returns $(\lambda^\emptyset, \lambda^\emptyset)$ and RecursivePG(G) returns $(\emptyset, \emptyset)$, these two results are equal therefore the theorem holds in this case.

Base When there is only one priority, which is even, then RecursiveUPG(G^λ) returns $(V^\lambda, \lambda^\emptyset)$ and RecursivePG(G) return $(V, \emptyset)$, these two results are equal therefore the theorem holds in this case. Similarly when the one priority is odd.

Step Player α gets the same value in both algorithms since the highest priority is equal for both algorithms.

Let $U = \{(c, \hat{v}) \in V \mid \hat{\Omega}(\hat{v}) = h\}$ (as calculated by RecursivePG) and $U^\lambda(\hat{v}) = V^\lambda(\hat{v})$ for all $\hat{v}$ with $\hat{\Omega}(\hat{v}) = h$ (as calculated by RecursiveUPG). We will show that $U =_\lambda U^\lambda$.

Let $(c, \hat{v}) \in U$ then $\hat{\Omega}(\hat{v}) = h$, therefore $U^\lambda(\hat{v}) = V^\lambda(\hat{v})$. Since $U \subseteq V$ we have $(c, \hat{v}) \in V$ and because the equality between V and V^λ we get $c \in V^\lambda(\hat{v})$ and $c \in U^\lambda(\hat{V})$.

Let $c \in U^\lambda(\hat{v})$, since $U^\lambda(\hat{v})$ is not empty we have $\hat{\Omega}(\hat{v}) = h$, furthermore $c \in V^\lambda(\hat{v})$ and therefore $(c, \hat{v}) \in V$. We can conclude that $(c, \hat{v}) \in U$ and $U =_\lambda U^\lambda$.

For the rest of the algorithm it is sufficient to see that attractor sets are equal if the game and input set are equal (as shown in lemma 6.2) and that the created subgames are equal (as shown in lemma 6.3). Since the subgames are equal we can apply the theorem on it by induction and conclude that the winning sets are also equal. □

Theorem 6.1 shows that solving a unified PG solves the VPG, furthermore the algorithm RecursiveUPG correctly solves a unified PG therefore we can conclude that for VPG $\hat{G}$ vertex $\hat{v}$ is won by player α for configuration c iff $c \in W_\alpha(\hat{v})$ with $(W_0, W_1) =$ RecursiveUPG($G_\downarrow$).

Function-wise attractor set Next we present an algorithm to calculate the function-wise attractor, the pseudo code is presented in algorithm 4. The algorithm considers vertices that are in the attracted set for some configuration, for every such vertex the algorithm tries to attract vertices that are connected by an incoming edge. If a vertex is attracted for some configuration then the incoming edges of that vertex will also be considered.

Consider unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ and vertex set $U \subseteq V$ both represented function-wise. We prove in the following lemma and theorem that the results calculated by α-Fattractor is equal to the definition of *α-FAttr* (definition 6.3).

Algorithm 4 α-FAttractor$(G, U : \hat{V} \to 2^{\mathfrak{C}})$

```
1: Queue Q ← {v̂ ∈ V̂ | U(v̂) ≠ ∅}
2: A ← U
3: while Q is not empty do
4:     v̂′ ← Q.pop()
5:     for every v̂ such that E(v̂, v̂′) ≠ ∅ do
6:         if v̂ ∈ V̂_α then
7:             a ← V(v̂) ∩ E(v̂, v̂′) ∩ A(v̂′)
8:         else
9:             a ← V(v̂)
10:            for every v̂″ such that E(v̂, v̂″) ≠ ∅ do
11:                a ← a ∩ (𝔠\E(v̂, v̂″) ∪ A(v̂″))
12:            end for
13:        end if
14:        if a\A(v̂) ≠ ∅ then
15:            A(v̂) ← A(v̂) ∪ a
16:            Q.push(v̂)
17:        end if
18:    end for
19: end while
20: return A
```

Lemma 6.5. *Vertex $\hat{v}$ and configuration c, with $c \in V(\hat{v})$, can only be attracted to U_i if there is a vertex $\hat{v}'$ such that $c \in E(\hat{v}, \hat{v}')$ and $c \in U_i(\hat{v}')$.*

Proof. We first observe that if $\hat{v} \in \hat{V}_\alpha$ then this property follows immediately from definition the function-wise attractor definition (6.3). If $\hat{v} \in \hat{V}_{\overline{\alpha}}$ we note that unified PGs are total and therefore all of their projections are also total. So vertex $\hat{v}$ has at least one outgoing edge for c, we have $\hat{w}$ such that $c \in E(\hat{v}, \hat{w})$. For $\hat{v}$ with c to be attracted we must have $c \in U_i(\hat{w})$. □

Theorem 6.6. *Set $A = \alpha$-FAttractor(G, U) satisfies $A = \alpha$-FAttr(G, U).*

Proof. We will prove two loop invariants over the while loop of the algorithm.

IV1: For every $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$ with $c \in A(\hat{w})$ we have $c \in \alpha$-FAttr$(G, U)(\hat{w})$.

IV2: For every $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$ that can be attracted to A either $c \in A(\hat{w})$ or there exists a $\hat{w}' \in Q$ such that $c \in E(\hat{w}, \hat{w}')$.

Base: Before the loop starts we have $A = U$, therefore IV1 holds. Furthermore all the vertices that are in A for some c are also in Q so IV2 holds.

Step: Consider the beginning of an iteration and assume IV1 and IV2 hold. To prove: IV1 and IV2 hold at the end of the iteration.

Set A only contains vertices with configurations that are in α-FAttr(G, U). The set is only updated through lines 6-13 and 15 of the algorithm which reflects the exact definition of the attractor set therefore IV1 holds at the end of the iteration.

Consider $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$, we distinguish three cases to prove IV2:

- $\hat{w}$ with c can be attracted by the beginning of the iteration but not by the end.

  This case can't happen because $A(\hat{w})$ only increases during the algorithm and the values for E and V are not changed throughout the algorithm.

- $\hat{w}$ with c can't be attracted by the beginning of the iteration but can by the end.

  For $\hat{w}$ with c to be able to be attracted at the end of the iteration there must be some $\hat{w}'$ with c such that during the iteration c was added to $A(\hat{w}')$ (lemma 6.5). Every $\hat{w}'$ for which $A(\hat{w}')$ is updated is added to the queue (lines 14-17). Therefore we have $\hat{w}' \in Q$ with $c \in E(\hat{w}, \hat{w}')$ and IV1 holds.

- $\hat{w}$ with c can be attracted by the beginning of the iteration and also by the end.

  Since IV2 holds at the beginning of the iteration we have either $c \in A(\hat{w})$ or we have some $\hat{w}' \in Q$ such that $c \in E(\hat{w}, \hat{w}')$. In the former case IV2 holds trivially by the end of the iteration since $A(\hat{w})$ can only increase. For the latter case we distinguish two scenario's.

  First we consider the scenario where vertex $\hat{v}'$ that is considered during the iteration (line 4 of the algorithm) is $\hat{w}'$. There is a vertex $c \in E(\hat{w}, \hat{w}')$ by IV2. Therefore we can conclude that $\hat{w}$ is considered in the for loop starting at line 5 and will be attracted in lines 6-13 and added to $A(\hat{w})$ in line 15. Therefore IV2 holds by the end of the iteration.

  Next we consider the scenario where $\hat{v}' \neq \hat{w}'$. In this case by the end of the iteration $\hat{w}'$ will still be in Q and IV2 holds.

Vertices are only added to the queue when something is added to A (if statement on line 14). This can only finitely often happen because $A(\hat{v})$ can never be larger than $V(\hat{v})$ so we can conclude that the while loop terminates after a finite number of iterations.

When the while loop terminates IV1 and IV2 hold so for every $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$ that can be attracted to A we have $c \in A(\hat{w})$. Since we start with $A = U$ we can conclude the soundness of the algorithm. IV1 shows the completeness. □

6.1.6 Running time

We consider the running time for solving VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ independently and collectively using the different types of representations.

The original algorithm runs in $O(e * n^d)$, if we run c parity games independently we get $O(c * e * n^d)$. We can also apply the original algorithm to a unified PG (represented set-wise) for a collective approach, in this case we get a parity game with $c * n$ vertices and $c * e$ edges. which gives a time complexity of $O(c * e * (c * n)^d)$. However, as we show next, this running time can be improved by using the property that a unified PG consists of c disconnected graphs.

We have introduced three types of collective algorithms: set-wise, function-wise with explicit configuration sets and function-wise with symbolic configuration sets. In all three algorithms the running time of the attractor set is dominant compared to the other operations performed, so we need three things: analyse the running time of the base cases, analyse the running time of the attractor set and analyse the recursion.

Base cases In the base cases the algorithm needs to do two things: find the highest and lowest priority and check if there are no more vertices in the game. For the set-wise variant we find the highest and lowest priorities by iterating all vertices, which takes $O(c * n)$. Checking if there are no more vertices is done in $O(1)$. For the function wise algorithms we can find the highest and lowest priority in $O(n)$ and checking if there are no vertices is also done in $O(n)$ since we have to check $V(\hat{v}) = \emptyset$ for every $\hat{v}$. Note that in a symbolic representation using BDDs we can check if a set is empty in $O(1)$ because the decision diagram contains a single node.

Attractor sets For the set-wise collective approach we can use the attractor calculation from the original algorithm which has a time complexity of $O(e)$, so for a unified PG having $c * e$ edges we have $O(c * e)$.

The function-wise variants uses a different attractor algorithm. First we consider the variant where sets of configurations are represented explicitly.

Consider algorithm 4. A vertex will be added to the queue when this vertex is attracted for some configuration, this can only happen $c * n$ times, once for every vertex configuration combination.

The first for loop considers all the incoming edges of a vertex. When we consider all vertices the for loop will have considered all edges, since we consider every vertex at most c times the for loop will run at most $c * e$ times in total.

The second for loop considers all outgoing edges of a vertex. The vertices that are considered are the vertices that have an edge going to the vertex being considered by the while loop. Since the while loop considers $c * n$ vertices the second for loop runs in total at most $c * n * e$ times. The loop itself performs set operations on the set of configurations which can be done in $O(c)$. This gives a total time complexity for the attractor set of $O(n * c^2 * e)$.

For the symbolic representation set operations can be done in $O(c^2)$ so we get a time complexity of $O(n * c^3 * e)$.

This gives the following time complexities

	Base	Attractor set
Set-wise	$O(c * n)$	$O(c * e)$
Function-wise explicit	$O(n)$	$O(n * c^2 * e)$
Function-wise symbolic	$O(n)$	$O(n * c^3 * e)$

Recursion The three algorithms behave the same way with regards to their recursion, so we analyse the running time of the set-wise variant and can derive the time complexity of the others using the result.

The algorithm has two recursions, the first recursion lowers the number of distinct priorities by 1. The second recursion removes at least one vertex, however the game is comprised of disjoint projections. We can use this fact in the analyses. Consider unified PG G and set A as specified by the algorithm. Now consider the projection of G to an arbitrary configuration q, $G_{|q}$. If $(G \backslash A)_{|q}$ contains a vertex that is won by player $\overline{\alpha}$ then this vertex is removed in the second recursion step. If there is no vertex won by player $\overline{\alpha}$ then the game is won in its entirety and the only vertices won by player $\overline{\alpha}$ are in different projections. We can conclude that for every configuration q the second recursion either removes a vertex or $(G \backslash A)_{|q}$ is entirely won by player α. Let $\overline{n}$ denote be the maximum number of vertices that are won by player $\overline{\alpha}$ in game $(G \backslash A)_{|q}$. Since every projection has at most n vertices the value for $\overline{n}$ can be at most n. Furthermore since $\overline{n}$ depends on A, which depends on the maximum priority, the value $\overline{n}$ gets reset when the top priority is removed in the first recursion. We can now write down the recursion of the algorithm:

$$T(d, \overline{n}) \leq T(d-1, n) + T(d, \overline{n}-1) + O(c * e)$$

When $\overline{n} = 0$ we will get $W_{\overline{\alpha}} = \emptyset$ as a result of the first recursion. In such a case there will be only 1 recursion.

$$T(d, 0) \leq T(d-1, n) + O(c * e)$$

Finally we have the base case where there is 1 priority:

$$T(1, \overline{n}) \leq O(c * n)$$

Expanding the second recursion gives

$$\begin{aligned} T(d) &\leq (n+1)T(d-1) + (n+1)O(c * e) \\ T(1) &\leq O(c * n) \end{aligned}$$

We prove that $T(d) \leq (n+d)^d O(c * e)$ by induction on d.

Base $d = 1$: $T(1) \leq O(c * n) \leq O(c * e) \leq (n+1)^1 O(c * e)$

Step $d > 1$:

$$\begin{aligned} T(d) &\leq (n+1)T(d-1) + (n+1)O(c * e) \\ &\leq (n+1)(n+d-1)^{d-1} O(c * e) + (n+1)O(c * e) \end{aligned}$$

Since $n + 1 \leq n + d - 1$ we get:

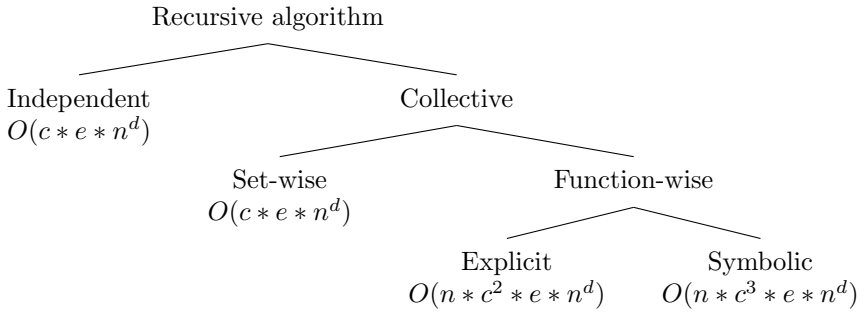
$$\begin{aligned} T(d) &\leq (n+d-1)(n+d-1)^{d-1} O(c * e) + (n+1)O(c * e) \\ &\leq ((n+d-1)(n+d-1)^{d-1} + n + 1)O(c * e) \\ &\leq (n(n+d-1)^{d-1} + d(n+d-1)^{d-1} - (n+d-1)^{d-1} + n + 1)O(c * e) \\ &\leq (n(n+d)^{d-1} + d(n+d)^{d-1} - (n+d-1)^{d-1} + n + 1)O(c * e) \end{aligned}$$

Because $(n+d-1)^{d-1} \geq n+1$ we have $-(n+d-1)^{d-1}+n+1 \leq 0$, therefore:

$$\begin{aligned} T(d) &\leq (n(n+d)^{d-1} + d(n+d)^{d-1})O(c*e) \\ &\leq (n+d)(n+d)^{d-1}O(c*e) \\ &\leq (n+d)^d O(c*e) \end{aligned}$$

This gives a time complexity of $O(c*e*(n+d)^d) = O(c*e*n^d)$. Note that the base time complexity is subsumed in the recursion by the time complexity of the attractor set. Since the time complexity of the attractor set is higher that the time complexity of the base cases for all three variants of algorithms we can simply fill in the attractor time complexity to get $O(n*c^2*e*n^d)$ for the function-wise explicit algorithm and $O(n*c^3*e*n^d)$ for the function-wise symbolic algorithm.

The different algorithms, including their time complexities, are repeated in the diagram below:



Running time in practice Earlier we hypothesized that the symbolic function-wise algorithm could have the best performance of the 4 algorithms, however it has the worst time complexity. Our hypothesis is based on the notion that VPGs most likely have a lot of commonalities and that sets of configurations in the VPG can be represented efficiently symbolically. Next we argue why the worst-case time complexity might not represent the running time in practice.

We dissect the running time of the function-wise algorithms. The running time complexities of the collective algorithms consist of two parts: the time complexity of the attractor set times n^d. The function-wise attractor set time complexity consists of three parts: the number of edges times the maximum number of vertices in the queue ($c*n$) times the time complexity for set operations ($O(c)$ for the explicit variant, $O(c^2)$ for the symbolic variant).

The number of vertices in the queue during attracting is at most $c*n$, however this number will only be large if we attract a very small number of configurations per time we evaluate an edge. As argued earlier we can most likely attract multiple configurations at the same time. This will decrease the number of vertices in the queue.

The time complexity of set operations is $O(c)$ when using an explicit representation and $O(c^2)$ when using a symbolic one. However, as shown in [22], a breadth-depth first implementation of BDDs keeps a table of already computed results. This allows us to get already calculated results in sublinear time. In total there are 2^c possible sets and therefore 2^{2c} possible set combinations and $O(2^c)$ possible set operations that can be computed. However when solving a VPG originating from an FTS there will most likely be a relatively small number of different edge guards, in which case the number of unique sets considered in the algorithm will be small and we can often retrieve a set calculation from the computed table.

We can see that even though the running time of the collective symbolic algorithm is the worse, its practical running time might be good when we are able to attract multiple configurations at the same time and have a small number of different edge guards.

6.2 Incremental pre-solve algorithm

Next we explore a collective algorithm that tries to solve the game for all configurations as much as possible, then split the configurations in two sets, create subgames using those two configuration sets and recursively repeat the process. Specifically we try to find vertices that are won by the same player for all configurations in $\mathfrak{C}$, if we have a vertex that is won by the same player for all configurations we call such a vertex *pre-solved*. The algorithm tries to recursively increase the set of pre-solved vertices until all vertices are either pre-solved or only 1 configuration remains. Pseudo code is presented in algorithm 5. The algorithm is based around finding sets P_0 and P_1, we want to find these sets in an efficient manner such that the algorithm doesn't spend time finding vertices that are already pre-solved. Finally when there is only 1 configuration left we want an algorithm that solves the parity game $G_{|c}$ in an efficient manner by using the vertices that are pre-solved.

Algorithm 5 IncPreSolve$(G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta), P_0, P_1)$

```
1: if |𝔠| = 1 then
2:     {c} ← 𝔠
3:     (W'_0, W'_1) ← solve G_|c using P_0 and P_1
4:     return (𝔠 × W'_0, 𝔠 × W'_1)
5: end if
6: P'_0 ← find vertices won by player 0 for all configuration in 𝔠
7: P'_1 ← find vertices won by player 1 for all configuration in 𝔠
8: if P'_0 ∪ P'_1 = V then
9:     return (𝔠 × P'_0, 𝔠 × P'_1)
10: end if
11: 𝔠^a, 𝔠^b ← partition 𝔠 in non-empty parts
12: (W^a_0, W^a_1) ← IncPreSolve(G ∩ 𝔠^a, P'_0, P'_1)
13: (W^b_0, W^b_1) ← IncPreSolve(G ∩ 𝔠^b, P'_0, P'_1)
14: W_0 ← W^a_0 ∪ W^b_0
15: W_1 ← W^a_1 ∪ W^b_1
16: return (W_0, W_1)
```

The subgames created are based on a set of configurations, we define the subgame operator as follows:

Definition 6.5. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ and non-empty set $\mathfrak{X} \subseteq \mathfrak{C}$ we define the subgame $G \cap \mathfrak{X} = (V, V_0, V_1, E', \Omega, \mathfrak{C}', \theta')$ such that*

- $\mathfrak{C}' = \mathfrak{C} \cap \mathfrak{X}$,
- $\theta'(e) = \theta(e) \cap \mathfrak{C}'$ *and*
- $E' = \{e \in E \mid \theta'(e) \neq \emptyset\}$.

VPGs are total, meaning that for every configuration and every vertex there is an outgoing edge from that vertex admitting that configuration. In subgames the set of configurations is restricted and only edge guards and edges are removed for configurations that fall outside the restricted set, therefore we still have totality. Furthermore it is trivial to see that every projection $G_{|c}$ is equal to $(G \cap \mathfrak{X})_{|c}$ for any $c \in \mathfrak{C} \cap \mathfrak{X}$.

Finally a subsubgame of two configuration sets is the same as the subgame of the intersection of these configuration sets, ie. $(G \cap \mathfrak{X}) \cap \mathfrak{X}' = G \cap (\mathfrak{X} \cap \mathfrak{X}') = G \cap \mathfrak{X} \cap \mathfrak{X}'$.

6.2.1 Finding P_0 and P_1

We can find P_0 and P_1 by using *pessimistic* parity games; a pessimistic PG is a parity game created from a VPG for a player $\alpha \in \{0, 1\}$ such that the PG allows all edges that player $\overline{\alpha}$ might take but only allows edges for α when that edge admits all the configurations in $\mathfrak{C}$.

Definition 6.6. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$, we define pessimistic PG $G_{\triangleright\alpha}$ for player $\alpha \in \{0, 1\}$, such that*

$$G_{\triangleright\alpha} = \{V, V_0, V_1, E', \Omega\}$$

with

$$E' = \{(v, w) \in E \mid v \in V_{\overline{\alpha}} \vee \theta(v, w) = \mathfrak{C}\}$$

Note that pessimistic parity games are not necessarily total. A play in a PG that is not total might result in a finite path, in such a case the player that can't make a move looses the play.

When solving a pessimistic PG $G_{\triangleright\alpha}$ we get winning sets W_0, W_1, every vertex in W_α is winning for player α in G played for any configuration, as shown in the following theorem.

Theorem 6.7. *Given:*

- *VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$,*
- *configuration $c \in \mathfrak{C}$,*
- *winning sets W_0^c, W_1^c for game G,*
- *player $\alpha \in \{0, 1\}$ and*
- *pessimistic PG $G_{\triangleright\alpha}$ with winning sets P_0 and P_1*

we have $P_\alpha \subseteq W_\alpha^c$.

Proof. Player α has a strategy in game $G_{\triangleright\alpha}$ such that vertices in P_α are won. We show that this strategy can also be applied to game $G_{|c}$ to win the same or more vertices.

First we observe that any edge that is taken by player α in game $G_{\triangleright\alpha}$ can also be taken in game $G_{|c}$ so player α can play the same strategy in game $G_{|c}$.

For player $\overline{\alpha}$ there are possibly edges that can be taken in $G_{\triangleright\alpha}$ but can't be taken in $G_{|c}$, in such a case player $\overline{\alpha}$'s choices are limited in game $G_{|c}$ compared to $G_{\triangleright\alpha}$ so if player $\overline{\alpha}$ can't win a vertex in $G_{\triangleright\alpha}$ then he/she can't win that vertex in $G_{|c}$.

We can conclude that applying the strategy from game $G_{\triangleright\alpha}$ in game $G_{|c}$ for player α wins the same or more vertices. □

Figure 16 shows an example VPG with corresponding pessimistic games, after solving the pessimistic games we find $P_0 = \{v_2\}$ and $P_1 = \{v_0\}$.

Pessimistic subgames Vertices in winning set P_α for $G_{\triangleright\alpha}$ are also winning for player α in pessimistic subgames of G, as shown in the following lemma.

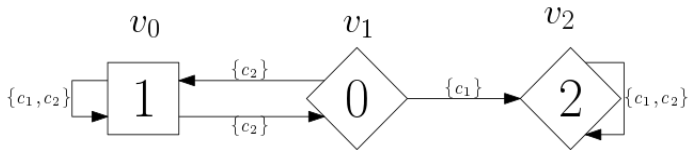
Lemma 6.8. *Given:*

- *VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$,*
- *P_0 being the winning set of game $G_{\triangleright 0}$ for player 0,*
- *P_1 being the winning set of game $G_{\triangleright 1}$ for player 1,*
- *non-empty set $\mathfrak{X} \subseteq \mathfrak{C}$,*
- *player $\alpha \in \{0, 1\}$ and*
- *winning sets Q_0, Q_1 for game $(G \cap \mathfrak{X})_{\triangleright\alpha}$*

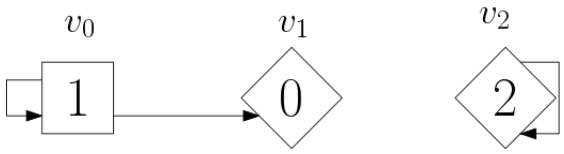
we have

$$P_0 \subseteq Q_0$$

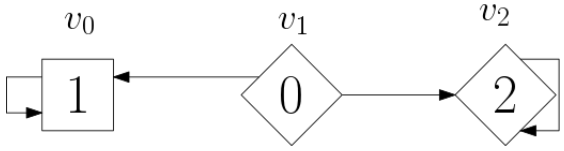
$$P_1 \subseteq Q_1$$



(a) VPG G consisting of 2 configurations



(b) Pessimistic game $G_{\triangleright 0}$



(c) Pessimistic game $G_{\triangleright 1}$

Figure 16: A VPG with its corresponding pessimistic games

Proof. Let edge (v, w) be an edge in game $G_{\triangleright\alpha}$ with $v \in V_\alpha$. Edge (v, w) admits all configuration in $\mathfrak{C}$ so it also admits all configuration in $\mathfrak{C} \cap \mathfrak{X}$, therefore we can conclude that edge (v, w) is also an edge of game $(G \cap \mathfrak{X})_{\triangleright\alpha}$.

Let edge (v, w) be an edge in game $(G \cap \mathfrak{X})_{\triangleright\alpha}$ with $v \in V_{\overline{\alpha}}$. The edge admits some configuration in $\mathfrak{C} \cap \mathfrak{X}$, this configuration is also in $\mathfrak{C}$ so we can conclude that edge (v, w) is also an edge of game $G_{\triangleright\alpha}$.

We have concluded that game $(G \cap \mathfrak{X})_{\triangleright\alpha}$ has the same or more edges for player α as game $G_{\triangleright\alpha}$ and the same or less edges for player $\overline{\alpha}$. Therefore we can conclude that any vertex won by player α in $G_{\triangleright\alpha}$ is also won by α in game $(G \cap \mathfrak{X})_{\triangleright\alpha}$, ie. $P_\alpha \subseteq Q_\alpha$.

Let $v \in P_{\overline{\alpha}}$, using theorem 6.7 we find that v is winning for player $\overline{\alpha}$ in $G_{|c}$ for any $c \in \mathfrak{C}$. Because projections of subgames are the same as projections of the original game we can conclude that v is winning for player $\overline{\alpha}$ in $(G \cap \mathfrak{X})_{|c}$ for any $c \in \mathfrak{C} \cap \mathfrak{X}$. Assume $v \notin Q_{\overline{\alpha}}$ then $v \in Q_\alpha$ and using theorem 6.7 we find that v is winning for player α in $(G \cap \mathfrak{X})_{|c}$ for any $c \in \mathfrak{C} \cap \mathfrak{X}$. This is a contradiction so we can conclude $v \in Q_{\overline{\alpha}}$ and therefore $P_{\overline{\alpha}} \subseteq Q_{\overline{\alpha}}$. □

6.2.2 Algorithm

In order to find P_0 and P_1 we need to solve a pessimistic parity game, specifically we want to solve a parity game that uses the vertices that are already pre-solved to efficiently solve the parity game. Note that when there is 1 configuration left we need a similar algorithm. In algorithm 6 we present the IncPreSolve algorithm using pessimistic parity games. The algorithm uses a Solve algorithm for solving parity games using the pre-solved vertices. First we show the correctness of the IncPreSolve algorithm, later we explore an appropriate Solve algorithm.

A Solve algorithm must correctly solve a game as long as the sets P_0 and P_1 are in fact vertices that are won by player 0 and 1 respectively. We prove that this is the case in the IncPreSolve algorithm.

Theorem 6.9. *Given VPG $\hat{G}$. For every* Solve(G, P_0, P_1) *that is invoked during* IncPreSolve$(\hat{G}, \emptyset, \emptyset)$ *we have winning sets W_0, W_1 for game G for which the following holds:*

$$P_0 \subseteq W_0$$

$$P_1 \subseteq W_1$$

Proof. When $P_0 = \emptyset$ and $P_1 = \emptyset$ the theorem holds trivially. So we will start the analyses after the first recursion.

Algorithm 6 IncPreSolve($G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta), P_0, P_1$)

1: **if** $|\mathfrak{C}| = 1$ **then**
2: $\quad \{c\} \leftarrow \mathfrak{C}$
3: $\quad (W_0', W_1') \leftarrow \text{Solve}(G_{|c}, P_0, P_1)$
4: $\quad$ **return** $(\mathfrak{C} \times W_0', \mathfrak{C} \times W_1')$
5: **end if**
6: $(P_0', -) \leftarrow \text{Solve}(G_{\triangleright 0}, P_0, P_1)$
7: $(-, P_1') \leftarrow \text{Solve}(G_{\triangleright 1}, P_0, P_1)$
8: **if** $P_0' \cup P_1' = V$ **then**
9: $\quad$ **return** $(\mathfrak{C} \times P_0', \mathfrak{C} \times P_1')$
10: **end if**
11: $\mathfrak{C}^a, \mathfrak{C}^b \leftarrow$ partition $\mathfrak{C}$ in non-empty parts
12: $(W_0^a, W_1^a) \leftarrow \text{IncPreSolve}(G \cap \mathfrak{C}^a, P_0', P_1')$
13: $(W_0^b, W_1^b) \leftarrow \text{IncPreSolve}(G \cap \mathfrak{C}^b, P_0', P_1')$
14: $W_0 \leftarrow W_0^a \cup W_0^b$
15: $W_1 \leftarrow W_1^a \cup W_1^b$
16: **return** (W_0, W_1)

After the first recursion the game is $\hat{G} \cap \mathfrak{X}$ with $\mathfrak{X}$ being either $\mathfrak{C}^a$ or $\mathfrak{C}^b$. The set P_0 is the winning set for player 0 for game $\hat{G}_{\triangleright 0}$ and the set P_1 is the winning set for player 1 for game $\hat{G}_{\triangleright 1}$. In the next recursion the game is $\hat{G} \cap \mathfrak{X} \cap \mathfrak{X}'$ with P_0 being the winning set for player 0 in game $(\hat{G} \cap \mathfrak{X})_{\triangleright 0}$ and P_1 being the winning set for player 1 in game $(\hat{G} \cap \mathfrak{X})_{\triangleright 1}$. In general, after the kth recursion, with $k > 0$, the game is of the form $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1}) \cap \mathfrak{X}^k$. Furthermore P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ and P_1 is the winning set for player 1 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 1}$.

Next we inspect the three places Solve is invoked:

1. Consider the case where there is only one configuration in $\mathfrak{C}$ (line 1-5). Because P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ the vertices in P_0 are won by player 0 in game $G_{|c}$ for all $c \in \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1}$ (using theorem 6.7). This includes the one element in $\mathfrak{C}$. So we can conclude $P_0 \subseteq W_0$ where W_0 is the winning set for player 0 in game $G_{|c}$ where $\{c\} = \mathfrak{C}$.

   Similarly for player 1 we can conclude $P_1 \subseteq W_1$ and the theorem holds in this case.

2. On line 6 the game $G_{\triangleright 0}$ is solved with P_0 and P_1. Because $G = \hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1} \cap \mathfrak{X}^k$ and P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ and P_1 is the winning set for player 1 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 1}$ we can apply lemma 6.8 to conclude that the theorem holds in this case.

3. On line 7 we apply the same reasoning and lemma to conclude that the theorem holds in this case.

□

Next we prove the correctness of the algorithm, assuming the correctness of the Solve algorithm.

Theorem 6.10. *Given VPG* $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$ *and* $(W_0, W_1) = \text{IncPreSolve}(\hat{G}, \emptyset, \emptyset)$. *For every configuration* $c \in \mathfrak{C}$ *and winning sets* $\hat{W}_0^c, \hat{W}_1^c$ *for game* $\hat{G}$ *player for* c *it holds that:*

$$(c, v) \in W_0 \iff v \in \hat{W}_0^c$$

$$(c, v) \in W_1 \iff v \in \hat{W}_1^c$$

Proof. We prove the theorem by applying induction on $\mathfrak{C}$.

Base $|\mathfrak{C}| = 1$, when there is only one configuration, being c, then the algorithm solves game $G_{|c}$. The product of the winning sets and $\{c\}$ is returned, so the theorem holds.

Step Consider P_0' and P_1' as calculated in the algorithm (line 6-7). By theorem 6.7 all vertices in P_0' are won by player 0 in game $G_{|c}$ for any $c \in \mathfrak{C}$, similarly for P_1' and player 1.

If $P'_0 \cup P'_1 = V$ then the algorithm returns $(\mathfrak{C} \times P'_0, \mathfrak{C} \times P'_1)$. In which case the theorem holds because there are no configuration vertex combinations that are not in either winning set and theorem 6.7 proves the correctness.

If $P'_0 \cup P'_1 \neq V$ then we have winning sets (W_0^a, W_1^a) for which the theorem holds (by induction) for game $G \cap \mathfrak{C}^a$ and (W_0^b, W_1^b) for which the theorem holds (by induction) for game $G \cap \mathfrak{C}^b$. The algorithm returns $(W_0^a \cup W_0^b, W_1^a \cup W_1^b)$. Since $\mathfrak{C}^a \cup \mathfrak{C}^b = \mathfrak{C}$ all vertex configuration combinations are in the winning sets and the correctness follows from induction. □

6.2.3 A parity game algorithm using pre-solved vertices

The fixed-point iteration algorithm can be used to solve parity games using pre-solved vertices. First recall the fixed-point formula to calculate W_0:

$$S(G = (V, V_0, V_1, E, \Omega)) = \nu Z_{d-1}.\mu Z_{d-2}.\ldots.muX.f(x)nuZ_0.F_0(Z_{d-1}, \ldots, Z_0)$$

Also recall that we can calculate a least fixed-point as follows:

$$\mu X.f(X) = \bigcup_{i \geq 0} X^i$$

where $X^i = f(X^{i-1})$ for $i > 0$ and $X^0 \subseteq \mu X.f(X)$. So picking the smallest value possible for X_0 will always correctly calculate $\mu X.f(X)$. Similarly we can calculate fixed-point a greatest fixed-point as follows:

$$\nu X.f(X) = \bigcap_{i \geq 0} X^i$$

where $X^i = f(X^{i-1})$ for $i > 0$ and $X^0 \supseteq \nu X.f(X)$. So picking the largest value possible for X_0 will always correctly calculate $\nu X.f(X)$.

Let G be a PG and let sets P_0 and P_1 be such that vertices in P_0 are won by player 0 and vertices in P_1 are won by player 1. We can fixed-point iterate $S(G)$ to calculate W_0, we know that W_0 is bounded by P_0 and P_1, specifically we have

$$P_0 \subseteq W_0 \subseteq V \backslash P_1$$

This restriction can be used to efficiently iterate $S(G)$. If no bounds are know we would iterate fixed-point formula starting at V or $\emptyset$ which are the largest and smallest values possible, however given the bounds we can start our iterations of greatest fixed-point variables at $V \backslash P_1$ and start our iterations of least fixed-point variables at P_0. The following lemma's and theorem prove this.

First we show that when a bound on a fixed-point formula is known and we iterate starting at that bound then at no point during the iteration do we find a value outside of the bound. This is shown in the following two lemma's, which are the dual of each other.

Lemma 6.11. *Given*

- *A complete lattice* $\langle 2^A, \subseteq \rangle$,
- *monotonic function* $f : 2^A \to 2^A$ *and*
- $R^\perp \subseteq A$ *and* $R^\top \subseteq A$ *such that* $R^\perp \subseteq \nu X.f(X) \subseteq R^\top$

we iterate X *by starting with* $X^0 = R^\top$. *For any* $i \geq 0$ *it holds that*

$$R^\perp \subseteq f(X^i) \subseteq R^\top$$

Proof. Assume $R^\perp \supset f(X^i) = X^{i+1}$. By fixed-point iteration definition we have $\nu X.f(X) = \bigcap_{j \geq 0} X^j$, so we find $R^\perp \supset \nu X.f(x)$ which is a contradiction so $R^\perp \subseteq f(X^i)$.

Assume $X^{i+1} = f(X^i) \supset R^\top$. We start iterating at $X^0 = R^\top$ and have $X^{i+1} \supset X^0$. Consider $f(X^0)$, if $f(X^0) \subseteq X^0$ then $X^1 \subseteq X^0$ and therefore (because of monotonicity) $f(X^1) \subseteq f(X^0) = X^1$, this gives a decreasing series of X values never becoming larger than $R^\top$ and therefore we can't have $X^{i+1} \supset X^0$.

We can conclude that $X^1 = f(X^0) \supset X^0$. Next consider $f(X^1)$, we have $X^1 \supset X^0$ therefore (because of monotonicity) we get $f(X^1) \supseteq f(X^0) = X^1$. We have found $X^1 \supset R^\top$ and $f(X^1) \supseteq X^1$, by applying the Knaster-Tarski theorem (2.1) we conclude that the greatest fixed-point of f is at least $X^1 \supset R^\top$. This is a contradiction because $\nu X.f(X) \subseteq R^\top$ so the assumption doesn't hold and $f(X^i) \subseteq R^\top$. □

Lemma 6.12. *Given*

- *A complete lattice* $\langle 2^A, \subseteq \rangle$,
- *monotonic function* $f : 2^A \to 2^A$ *and*
- $R^\bot \subseteq A$ *and* $R^\top \subseteq A$ *such that* $R^\bot \subseteq \mu X.f(X) \subseteq R^\top$

we iterate X *by starting with* $X^0 = R^\bot$. *For any* $i \geq 0$ *it holds that*

$$R^\bot \subseteq f(X^i) \subseteq R^\top$$

Proof. Assume $R^\bot \supset f(X^i) = X^{i+1}$. We start iteration at $X^0 = R^\bot$ and have $X^{i+1} \subset X^0$. Consider $f(X^0)$, if $f(X^0) \supseteq X^0$ then $X^1 \supseteq X^0$ and therefore (because of monotonicity) $f(X^1) \supseteq f(X^0) = X^1$, this gives an increasing series of X values never becoming smaller than $R^\bot$ and therefore we can't have $X^{i+1} \subset X^0$. We can conclude that $X^1 = f(X^0) \subset X^0$. Next consider $f(X^1)$, we have $X^1 \subset X^0$ therefore (because of monotonicity) we get $f(X^1) \subseteq f(X^0) = X^1$. We have found $X^1 \subset R^\bot$ and $f(X^1) \subseteq X^1$, by applying the Knaster-Tarski theorem (2.1) we conclude that the least fixed-point of f is at most $X^1 \subset R^\bot$. This is a contradiction because $R^\bot \subseteq \mu X.f(X)$ so the assumption doesn't hold and $R^\bot \subseteq f(X^i)$.

Assume $X^{i+1} = f(X^i) \supset R^\top$. By fixed-point iteration definition we have $\mu X.f(X) = \bigcup_{j \geq 0} X^j$, so we find $\mu X.f(x) \supset R^\top$ which is a contradiction so $f(X^i) \subseteq R^\top$. □

Using this property of iterating bounded fixed-point variables we can prove that we can indeed start our iteration at the bounds when solving $S(G)$.

Theorem 6.13. *Given PG* $G = (V, V_0, V_1, E, \Omega)$ *with* P_0 *and* P_1 *such that vertices in* P_0 *are won by player* 0 *in game* G *and vertices in* P_1 *are won by player* 1 *in game* G *we can solve* $S(G)$ *by iterating the fixed-point variables starting at* P_0 *for least fixed-points and starting at* $V \backslash P_1$ *for greatest fixed-points.*

Proof. Let $f(Z_{d-1}) = \mu Z_{d-2} \ldots \nu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$. Because $\nu Z_{d-1}.f(Z_{d-1})$ calculates W_0 we know $P_0 \subseteq \nu Z_{d-1}.f(Z_{d-1}) \subseteq V \backslash P_1$ so we can start the fixed-point iteration at $Z^0_{d-1} = V \backslash P_1$. Using lemma 6.11 we find for any $i \geq 0$ we have $P_0 \subseteq f(Z^i_{d-1}) \subseteq V \backslash P_1$.

Let $g(Z_{d-2}) = \nu Z_{d-3} \ldots \nu Z_0.F_0(Z^i_{d-1}, Z_{d-2}, \ldots, Z_0)$. We found $P_0 \subseteq f(Z^i_{d-1}) \subseteq V \backslash P_1$, therefore we have $P_0 \subseteq \mu Z_{d-2}.g(Z_{d-2}) \subseteq V \backslash P_1$ so we can start the fixed-point iteration at $Z^0_{d-2} = P_0$. Using lemma 6.12 we find that for any $j \geq 0$ we have $P_0 \subseteq g(Z^j_{d-2}) \subseteq V \backslash P_1$.

We can repeat this logic up until Z_0 to conclude that the theorem holds. □

We take the fixed-point algorithm presented in [3] and modify it by starting at P_0 and $V \backslash P_1$. The pseudo code is presented in algorithm 7, its correctness follows from [3] and theorem 6.13.

This algorithm is appropriate to use as a Solve algorithm in IncPreSolve since it solves parity games that are not necessarily total and uses P_0 and P_1.

6.2.4 Running time

We consider the running time for solving VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ independently and collectively.

The fixed-point iteration algorithm without P_0 and P_1 runs in $O(e * n^d)$. We can use this algorithm to solve G independently, ie. solve all the projections of G. This gives a time complexity of $O(c * e * n^d)$.

Next consider the IncPreSolve algorithm for a collective approach, observe that in the worst case we have to split the set of configurations all the way down to individual configurations. We can consider the recursion as a tree where the leafs are individual configurations and at every internal node the set of configurations is split in two. Since in the worst case there are c leaves, there are at most $c - 1$ internal nodes. At every internal node the algorithm solves two games and at every leaf the algorithm solves 1 game, so we get $O(c + 2c - 2) = O(c)$ games that are being solved by IncPreSolve. In the worst case the values

Algorithm 7 Fixed-point iteration with P_0 and P_1

```
1: function FPIter(G = (V, V_0, V_1, E, Ω), P_0, P_1)
2:     for i ← d − 1, ..., 0 do
3:         Init(i)
4:     end for
5:     repeat
6:         Z'_0 ← Z_0
7:         Z_0 ← Diamond() ∪ Box()
8:         i ← 0
9:         while Z_i = Z'_i ∧ i < d − 1 do
10:            i ← i + 1
11:            Z'_i ← Z_i
12:            Z_i ← Z_{i−1}
13:            Init(i − 1)
14:        end while
15:    until i = d − 1 ∧ Z_{d−1} = Z'_{d−1}
16:    return (Z_{d−1}, V\Z_{d−1})
17: end function
```

```
1: function Init(i)
2:     Z_i ← P_0 if i is odd, V\P_1 otherwise
3: end function
```

```
1: function Diamond
2:     return {v ∈ V_0 | ∃_{w∈V}(v, w) ∈ E ∧ w ∈ Z_{Ω(w)}}
3: end function
```

```
1: function Box
2:     return {v ∈ V_1 | ∀_{w∈V}(v, w) ∈ E ⟹ w ∈ Z_{Ω(w)}}
3: end function
```

for P_0 and P_1 are empty. In this case the FPIte algorithm behaves the same as the original algorithm and has a time complexity of $O(e * n^d)$, this gives an overall time complexity of $O(c * e * n^d)$ which is equal to an independent solving approach.

7 Locally solving (variability) parity games

As discussed in the preliminaries PGs can be solved either globally or locally. Similar to PGs we can solve VPGs either globally or locally. When locally solving a VPG we determine for which configurations a certain vertex is won by player 0 and for which configurations the vertex is won by player 1. When globally solving a VPG we determine this for every vertex in the VPG.

When solving a VPG globally we might encounter significant differences in parts of the game or intermediate results between configurations that we perhaps don't encounter when solving it locally because we can terminate earlier. Therefore we hypothesize that the increase in performance between globally-collectively solving VPGs and locally-collectively solving VPGs is greater than the increase performance between globally-independently solving VPGs and locally-independently solving VPGs.

The algorithms we have seen thus far are global algorithms, in this section we introduce local variants for the PG algorithms we have seen: Zielonka's recursive algorithm and fixed-point iteration algorithm. Furthermore we introduce local variants for the VPG algorithms we have seen: the recursive algorithm for VPGs and the incremental pre-solve algorithm.

7.1 Locally solving parity games

The two parity game algorithms introduced in the preliminaries (Zielonka's recursive algorithm and the fixed-point iteration algorithm) can be turned into local variants. This local variant can be used to solve VPGs locally-independently.

7.1.1 Zielonka's recursive algorithm local

To make Zielonka's recursive algorithm a local algorithm, we use the property that vertices won by player $\overline{\alpha}$ in game $G \backslash A$ are also won by player $\overline{\alpha}$ in game G. This property is proven in the following lemma.

Lemma 7.1. *In* RecursivePG(G) *any vertex won by player* $\overline{\alpha}$ *in game* $G \backslash A$ *is also won by* $\overline{\alpha}$ *in game* G*.*

Proof. We simply observe the behaviour of the algorithm and note that the algorithm returns B as winning for player $\overline{\alpha}$, since $W'_{\overline{\alpha}} \subseteq B$ by definition of the attractor set we find that any vertex in $W'_{\overline{\alpha}}$ is in $W_{\overline{\alpha}}$. The correctness of RecursivePG follows from [23], therefore the lemma holds. □

We use this property to, in some cases, not go into the second recursion because we already found the vertex we are trying to solve locally. We extend the algorithm with two parameters, v_0 and Δ. Vertex v_0 represents the vertex that we try to solve. Parameter $\Delta \subseteq \{0, 1\}$ contains a set of players for which we try to find vertex v_0, if we find v_0 to be won by a player that is in Δ we are done for that recursion. However if we find vertex v_0 to be won by a player that is not in Δ we need to solve the entire game. The algorithm can make use of this to tell a recursive call it needs to find the vertex to be won by player $\overline{\alpha}$ in the subgame because then this vertex is also won by player $\overline{\alpha}$ in the game itself. If the vertex is not won by $\overline{\alpha}$ the entire game needs to be solved because the winning sets are required in the second recursion. Pseudo code for the algorithm is provided in algorithm 8.

To prove the correctness we show that winning sets resulting from RecursivePGLocal are mutually exclusive and that vertex v_0 is in the correct winning set.

The following theorem proves that the winning sets are mutually exclusive.

Theorem 7.2. *Given total parity game* $G = (V, V_0, V_1, E, \Omega)$*, vertex* v_0 *(which is not necessarily in* V*) and* $\Delta \subseteq \{0, 1\}$*. We have for sets* $(W_0, W_1) =$ RecursivePGLocal(G, v_0, Δ) *that* $W_0 \cap W_1 = \emptyset$*.*

Proof. Proof by induction on G with induction hypothesis:

$$W_0 \cap W_1 = \emptyset \text{ and } W_0 \cup W_1 \subseteq V$$

Base: When G is empty or there is only one priority then the algorithm returns V and $\emptyset$ for winning sets so the IH holds.

Step: For the first recursion we can apply induction to find that $W'_0 \cap W'_1 = \emptyset$ and $W'_0 \cup W'_1 \subseteq V \backslash A$. If $W'_{\overline{\alpha}} = \emptyset$ then the algorithm returns $A \cup W'_{\alpha}$ and $\emptyset$ which makes the IH true.

Algorithm 8 RecursivePGLocal($PG\ G = (V, V_0, V_1, E, \Omega), v_0, \Delta$)

$m \leftarrow \min\{\Omega(v) \mid v \in V\}$
$h \leftarrow \max\{\Omega(v) \mid v \in V\}$
if $h = m$ or $V = \emptyset$ **then**
    if h is even or $V = \emptyset$ **then**
        return $(V, \emptyset)$
    else
        return $(\emptyset, V)$
    end if
end if
$\alpha \leftarrow 0$ if h is even and 1 otherwise
$U \leftarrow \{v \in V \mid \Omega(v) = h\}$
$A \leftarrow \alpha$-$Attr(G, U)$
if $\overline{\alpha} \in \Delta$ **then**
    $(W'_0, W'_1) \leftarrow$ RecursivePGLocal$(G \backslash A, v_0, \{\overline{\alpha}\})$
else
    $(W'_0, W'_1) \leftarrow$ RecursivePGLocal$(G \backslash A, v_0, \emptyset)$
end if
if $W'_{\overline{\alpha}} = \emptyset$ **then**
    $W_\alpha \leftarrow A \cup W'_\alpha$
    $W_{\overline{\alpha}} \leftarrow \emptyset$
else
    if $\overline{\alpha} \in \Delta \wedge v_0 \in W'_{\overline{\alpha}}$ **then**
        $W_\alpha \leftarrow \emptyset$
        $W_{\overline{\alpha}} \leftarrow W'_{\overline{\alpha}}$
    else
        $B \leftarrow \overline{\alpha}$-$Attr(G, W'_{\overline{\alpha}})$
        if $\overline{\alpha} \in \Delta \wedge v_0 \in B$ **then**
            $W_\alpha \leftarrow \emptyset$
            $W_{\overline{\alpha}} \leftarrow B$
        else
            $(W''_0, W''_1) \leftarrow$ RecursivePGLocal$(G \backslash B, v_0, \Delta)$
            $W_\alpha \leftarrow W''_\alpha$
            $W_{\overline{\alpha}} \leftarrow W''_{\overline{\alpha}} \cup B$
        end if
    end if
end if
return (W_0, W_1)

If $W'_{\overline{\alpha}} \neq \emptyset$ then we consider two cases. First consider $\overline{\alpha} \in \Delta \wedge v_0 \in W'_{\overline{\alpha}}$, then the algorithm returns $\emptyset$ and $W'_{\overline{\alpha}}$ as winning sets, since $W'_{\overline{\alpha}} \subseteq V \backslash A \subseteq V$ the IH holds. Otherwise we calculate a value for B, by definition of the attractor set we get $B \subseteq V$, if $\overline{\alpha} \in \Delta \wedge v_0 \in B$ we return $\emptyset$ and B so the IH holds. Otherwise we go into the second recursion where we find by induction that $W''_0 \cap W''_1 = \emptyset$ and $W''_0 \cup W''_1 \subseteq V \backslash B$. The algorithm returns W''_α and $W''_{\overline{\alpha}} \cup B$ as winning sets. Because $B \subseteq V$ we find $W''_0 \cup W''_1 \cup B \subseteq V$, we can also conclude that $W''_\alpha \cap B = \emptyset$ so the IH holds. □

The next theorem proves that vertex v_0 is always in the correct winning set.

Theorem 7.3. *Given total parity game $G = (V, V_0, V_1, E, \Omega)$, vertex v_0 (which is not necessarily in V), $\Delta \subseteq \{0, 1\}$ and winning sets Q_0 and Q_1 for game G. We have for sets $(W_0, W_1) =$ RecursivePGLocal(G, v_0, Δ) that either or both of the following statements hold:*

(I) For some $\beta \in \{0, 1\}$ we have $v_0 \in W_\beta$, $v_0 \in Q_\beta$ and $\beta \in \Delta$.

(II) $W_0 = Q_0$ and $W_1 = Q_1$.

Proof. To prove the correctness we compare the behaviour of RecursivePGLocal with the behaviour of the original algorithm RecursivePG, which we know solves a parity game globally. Note that solving G globally is equal to statement (II).

Prove by induction on G.

Base: When there are no vertices in V or there is only one priority then the local algorithm behaves the same as the original algorithm so statement (II) holds.

Step: We start by distinguishing two cases, first assume that $v_0 \notin V$. Vertex v_0 is also not in any subgame of G. In the first recursion we know by induction that either or both of the statements are true for $G \backslash A$, since v_0 is not in game $G \backslash A$ only statement (II) can be true. So the first recursion returns the same result as the original algorithm would. The if statements guarding the second recursion are always false because $v_0 \notin B$ and $v_0 \notin W'_{\overline{\alpha}}$ and similarly as with game $G \backslash A$ the vertex v_0 is not in $G \backslash B$. So the second recursion also returns the same values as the original algorithm would. We can conclude that when $v_0 \notin V$ the local algorithm behaves the same as the original and therefore statement (II) holds.

Now consider $v_0 \in V$. Again we distinguish two cases, starting with $\overline{\alpha} \notin \Delta$. In the first recursive call we get $\Delta = \emptyset$, by induction either or both statements hold for $G \backslash A$, however with $\Delta = \emptyset$ statement (II) must be true. Therefore the first recursive call behaves the same as the original algorithm. If $W'_{\overline{\alpha}} = \emptyset$ then the parity game is solved and statement (II) holds for game G. If $W'_{\overline{\alpha}} \neq \emptyset$ then the if statements guarding the second recursion are always false since $\overline{\alpha} \notin \Delta$ so the second recursive call is always performed. Up until this second recursive call the local algorithm behaves the same as the original algorithm. From the correctness of the original algorithm we can conclude that vertices won by some player in game $G \backslash B$ are also won by that player in game G. This holds true for the local algorithm in this case since it has behaved identical up to this point. By induction we know either or both statements are true, if statement (II) is true for $G \backslash B$ then the algorithm behaves the same as the original and statement (II) is also true for game G. If statement (I) is true for $G \backslash B$ and v_0 is won by player β in $G \backslash B$ then we know v_0 is also won by player β in game G and thus statement (I) holds for G.

Next consider $\overline{\alpha} \in \Delta$ (and $v_0 \in V$). The first recursive call is made with $\Delta = \{\overline{\alpha}\}$. If $W'_{\overline{\alpha}} = \emptyset$ then statement (II) must be true for game $G \backslash A$ and the algorithm behaves the same as the original in which case statement (II) is true for G. If $W'_{\overline{\alpha}} \neq \emptyset$ then we consider v_0 being in $W'_{\overline{\alpha}}$, by induction we can conclude that v_0 is indeed won by player $\overline{\alpha}$ in game $G \backslash A$. Furthermore from the correctness of the original algorithm we can conclude that any vertex won by player $\overline{\alpha}$ in game $G \backslash A$ is also won by player $\overline{\alpha}$ in game G. So returning winning sets $\emptyset$ and $W'_{\overline{\alpha}}$ make statement (I) true for game G.

Consider v_0 not being in $W'_{\overline{\alpha}}$ but in B. We can conclude that statement (II) is true for subgame $G \backslash A$ so the algorithm behaves the same as the original algorithm up until this point. Using the original algorithm we can see that any vertex in B is won by player $\overline{\alpha}$ in game G so if we find v_0 in B we can return B for player $\overline{\alpha}$ which includes v_0 and $\emptyset$ for player α to make statement (I) true for game G.

If v_0 is also not in B then we still conclude that statement (II) is true for subgame $G \backslash A$ and that the algorithm has behaved the same as the original algorithm up until this point. From the correctness of the original algorithm we can conclude that vertices won by some player in game $G \backslash B$ are also won by that player in game G. This holds true in the local algorithm since it has behaved identical up to this point. By

induction we know either or both statements are true, if statement (II) is true for $G \backslash B$ then the algorithm behaves the same as the original and statement (II) is also true for game G. If statement (I) is true for $G \backslash B$ and v_0 is won by player β in $G \backslash B$ then we know v_0 is also won by player β in game G and thus statement (I) holds for G. □

Calling RecursivePGLocal$(G, v_0, \{0, 1\})$ with v_0 in G either solves the full game (statement (II)) or correctly puts v_0 in either winning set (statement (I)). In both cases v_0 is in the correct winning set and therefore it is not in the other winning set, so the game is solved locally.

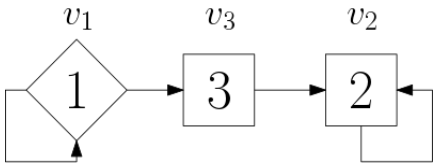
The time complexity of the local variant is the same as the original algorithm: $O(e * n^d)$. If vertex v_0 is not winning for a player in Δ than the algorithm behaves the same as the original so its worst-case complexity is the same.

A simpler idea A simpler version of the algorithm is where there is no Δ variable and the algorithm simply terminates when v_0 is found to be won for player $\overline{\alpha}$. After all, vertices won by player $\overline{\alpha}$ in the subgame are also won by player $\overline{\alpha}$ in the game itself. However this idea is flawed because this property only extends one recursion deep. So during some recursion it is possible that the algorithm finds a vertex to be won by player $\overline{\alpha}$ for the subgame $G \backslash A$ and therefore also for game G, however game G might be a subgame itself of some game H. The vertex found to be won by player $\overline{\alpha}$ in $G \backslash A$ and G might not be won by player $\overline{\alpha}$ in game H.

We disprove the following conjecture to show that there indeed can be a vertex that is won by player $\overline{\alpha}$ at some point but be won by player α in a higher recursion level.

Conjecture 7.4 (Disproven)**.** *For any* RecursivePG(G') *that is invoked during* RecursivePG(G) *it holds that any vertex* $v \in W'_{\overline{\alpha}}$ *is won by player* $\overline{\alpha}$ *in game* G*.*

Counterexample. Consider the following parity game G:



All vertices are won by player 0 (v_1 plays to v_3, v_3 must play to v_2 and v_2 must play to itself therefore we always get an infinite path of v_2's).

We solve this game using RecursivePG and write down the values of relevant variables below. We use the tilde decoration to indicate values for variables in the first recursion:

RecursivePG(G):
$h = 3, \alpha = 1$
$A = \{v_3\}$
  RecursivePG$(G \backslash A)$:
  $\tilde{h} = 2, \tilde{\alpha} = 0$
  $\tilde{A} = \{v_2\}$
    RecursivePG$(G \backslash A \backslash \tilde{A})$
    $\tilde{W}'_0 = \emptyset$
    $\tilde{W}'_1 = \{v_1\} = \tilde{W}'_{\overline{\tilde{\alpha}}}$
    Vertex v_1 **is in** $\tilde{W}'_{\overline{\tilde{\alpha}}}$ **however in** G **the vertex is won by player** $\tilde{\alpha}$.
  $\tilde{B} = \{v_1\}$
    RecursivePG$(G \backslash A \backslash \tilde{B})$
    $\tilde{W}''_0 = \{v_2\}, \tilde{W}''_1 = \emptyset$
$W'_0 = W'_{\overline{\alpha}} = \{v_2\}$
$W'_1 = W'_{\alpha} = \{v_1\}$
$B = V$
$W_0 = V$

This counterexample disproves the conjecture. □

An algorithm that simply stops when v_0 is found to be won for player $\overline{\alpha}$ requires the above conjecture to be true, since this isn't the case such an algorithm is flawed. This problem is solved by using the Δ variable, as is used in the RecursivePGLocal algorithm.

7.1.2 Fixed-point iteration local

The fixed-point iteration algorithm can be modified to locally solve a game for vertex v_0 by distinguishing two cases:

1. If $d-1$ is even then the outermost fixed-point variable is a greatest fixed-point variable. When at some point $v_0 \notin Z_{d-1}$ then we know v_0 is never won by player 0 and we are done.
2. If $d-1$ is odd then the outermost fixed-point variable is a least fixed-point variable. When at some point $v_0 \in Z_{d-1}$ then we know v_0 is won by player 0 and we are done.

If vertex v_0 is won by player 0 in the first case or won by player 1 in the second case then the algorithm never terminates early. So in the worst-case the local algorithm behaves the same as the global algorithm, therefore we have an identical time complexity, ie. $O(e * n^d)$.

7.2 Locally solving variability parity games

To solve a VPG locally we try to find out for which configurations vertex $\hat{v}_0$ is won by player 1 and for which configurations it is won by player 0. We consider the two collective VPG algorithms we have seen thus far and create local variants of them.

7.2.1 Local recursive algorithm for variability parity games

We can modify the recursive algorithm for VPGs into a local variant. In section 7.1.1 we have seen a local variant of the Zielonka's recursive algorithm for PGs that uses $\Delta \subseteq \{0, 1\}$ to indicate for which player we are trying to find the specific vertex.

When solving a VPG we have vertex $\hat{v}_0$, we try to find for every configuration c which player wins $\hat{v}_0$. Consider VPG G with configuration set $\mathfrak{C}$. Unified PG $G_\downarrow$ contains vertices $\mathfrak{C} \times \{\hat{v}_0\}$, if we find out the winning player for each of these vertices we have solved the VPG locally. So we are going to solve a unified PG locally, however instead of finding the winner of a single vertex we are finding the winners for a set of vertices, specifically vertices: $\mathfrak{C} \times \{\hat{v}_0\}$.

When we locally solve a parity game using the recursive algorithm we can sometimes not go into the second recursion because we already found the $\hat{v}_0$, however when locally solving a unified PG we might find $\hat{v}_0$ for some configuration but not for all. So when we find $\hat{v}_0$ to be won by player $\overline{\alpha} \in \Delta$ for configurations $C \subseteq \mathfrak{C}$ then we remove all vertices with configurations C from the game, ie. we remove vertices $C \times \hat{V}$. For the remaining vertices we do go into the second recursion. Pseudo code is presented in algorithm 9, we introduce function LocalConfs which find the configurations for which we have found the local solution.

The algorithm uses definitions to reason about projections of games and sets to configuration(s). Previously we introduced a simple projection definition that projects a unified PG to a configuration. This is possible because vertices in a unified PG consist of pairs of configurations and origin vertices. We define a similar projection for sets of vertices consisting of pairs of configurations and origin vertices.

Definition 7.1. *Given set $X \subseteq (\mathfrak{C} \times \hat{V})$ we define the projection of X to $c \in \mathfrak{C}$, denoted by $X_{|c}$, as*

$$X_{|c} = \{\hat{v} \mid (c, \hat{v}) \in X\}$$

Furthermore we need to be able to reason about projections not only to a single vertex but to a group of vertices for which we introduce the following two definitions.

Definition 7.2. *Given set $X \subseteq (\mathfrak{C} \times \hat{V})$ we define the projection of X to $C \subseteq \mathfrak{C}$, denoted by $X_{||C}$, as*

$$X_{||C} = X \cap (C \times \hat{V})$$

Definition 7.3. *Given set $X \subseteq (\mathfrak{C} \times \hat{V})$ we define the complementary projection of X to $C \subseteq \mathfrak{C}$, denoted by $X_{|\backslash C}$, as*

$$X_{|\backslash C} = X \backslash (C \times \hat{V})$$

In order to prove the correctness of the algorithm we repeat parts of the original proof given in [23]. We first introduce a few auxiliary definitions and lemma's from [23].

Consider total PG (V, V_0, V_1, E, Ω) in the next definition and lemma's.

Definition 7.4. *[23] Set $X \subseteq V$ is an α-trap in G if and only if player $\overline{\alpha}$ can keep the play inside X.*

Lemma 7.5. *[23] Set $V \backslash \alpha\text{-}Attr(G, X)$ is an α-trap in G for any non-empty $X \subseteq V$.*

Lemma 7.6. *[23] Let $X \subseteq V$ be an α-trap in G. Then $\overline{\alpha}\text{-}Attr(G, X)$ is also an α-trap in G.*

Finally we observe that a winning set W_α of PG G is an $\overline{\alpha}$-trap in G, since if $\overline{\alpha}$ could play to $W_{\overline{\alpha}}$ from a vertex $v \in W_\alpha$ then v would be winning for $\overline{\alpha}$.

We prove the correctness of the algorithm by first showing that the winning sets are mutually exclusive. Next we prove that vertex $(c, \hat{v}_0)$ is in the correct winning set for every configuration.

The following theorem shows that the winning sets are mutually exclusive.

Theorem 7.7. *Given:*

- *VPG $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$,*
- *origin vertex $\hat{v}_0 \in \hat{V}$,*
- *total unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ that is a subgame of, or equal to, unified PG $\hat{G}_\downarrow$,*
- *a set of players $\Delta \subseteq \{0, 1\}$ and*
- *winning sets $(W_0, W_1) = \textsc{RecursiveUPGLocal}(G, \hat{v}_0, \Delta)$*

we have $W_0 \cap W_1 = \emptyset$.

Proof. Proof by induction on G, with induction hypothesis:

$$W_0 \cap W_1 = \emptyset \text{ and } W_0 \cup W_1 \subseteq V$$

Base: If there is only one priority or there are no vertices than one of the winning sets is empty and the other is equal to V so the IH holds.

Step: In the first recursion step we find by induction that W_0' and W_1' are mutually exclusive and that $W_0' \cup W_1' \subseteq V \backslash A$. We find that $W_0' \cup W_1' \cup A \subseteq V$. If $W_{\overline{\alpha}}' = \emptyset$ then we return $A \cup W_{\overline{\alpha}}'$ and $\emptyset$ as winning sets in which case the IH holds.

If $W_{\overline{\alpha}}' \neq \emptyset$ then we find a value for C_W. If $(W_{\overline{\alpha}}')_{|\backslash C_W} = \emptyset$ then $W_{\overline{\alpha}}'$ only contains vertices with a configuration that is in C_W. Winning set W_α gets vertices only with a configuration that is not in C_W therefore the two are mutually exclusive. Furthermore since $W_0' \cup W_1' \cup A \subseteq V$ we find $W_{\overline{\alpha}} \cup W_\alpha \subseteq V$ and the IH holds in this case.

If $(W_{\overline{\alpha}}')_{|\backslash C_W} \neq \emptyset$ then we find a value for B. Since B is the attractor set of $(W_{\overline{\alpha}}')_{|\backslash C_W}$ and edges don't cross configurations the set B contains no vertices with a configuration that is in C_W. Clearly $B \subseteq V$. The recursion step gives W_0'' and W_1'' which are mutually and have $W_0'' \cup W_1'' \subseteq G \backslash B \backslash V_{||C_W \cup C_B}$ by induction. The algorithm returns these winning sets with B and $(W_{\overline{\alpha}}')_{||C_W}$. Both B and $(W_{\overline{\alpha}}')_{||C_W}$ don't share any vertices with each other nor with the winning sets W_0'' and W_1''. Moreover all B and $(W_{\overline{\alpha}}')_{||C_W}$ are subsets of V so we can conclude that the IH holds. □

The next theorem proves that vertex $(c, \hat{v}_0)$ is in the correct winning set for every configuration $c \in \mathfrak{C}$.

Algorithm 9 RecursiveUPGLocal($PG\ G = (V \subseteq \mathfrak{C} \times \hat{V}, \hat{V}_0 \subseteq \hat{V}, \hat{V}_1 \subseteq \hat{V}, E \subseteq (\mathfrak{C} \times \hat{V}) \times (\mathfrak{C} \times \hat{V}), \hat{\Omega} : \hat{V} \to \mathbb{N}), \hat{v}_0 \in \hat{V}, \Delta \subseteq \{0,1\}$)

$m \leftarrow \min\{\hat{\Omega}(\hat{v}) \mid (c,\hat{v}) \in V\}$
$h \leftarrow \max\{\hat{\Omega}(\hat{v}) \mid (c,\hat{v}) \in V\}$
if $h = m$ or $V = \emptyset$ **then**
  if h is even or $V = \emptyset$ **then**
    return $(V, \emptyset)$
  else
    return $(\emptyset, V)$
  end if
end if
$\alpha \leftarrow 0$ if h is even and 1 otherwise
$U \leftarrow \{(c,\hat{v}) \in V \mid \hat{\Omega}(\hat{v}) = h\}$
$A \leftarrow \alpha\text{-}Attr(G, U)$
if $\overline{\alpha} \in \Delta$ **then**
  $(W_0', W_1') \leftarrow$ RecursiveUPGLocal$(G \backslash A, \hat{v}_0, \{\overline{\alpha}\})$
else
  $(W_0', W_1') \leftarrow$ RecursiveUPGLocal$(G \backslash A, \hat{v}_0, \emptyset)$
end if
if $W'_{\overline{\alpha}} = \emptyset$ **then**
  $W_\alpha \leftarrow A \cup W'_\alpha$
  $W_{\overline{\alpha}} \leftarrow \emptyset$
else
  $C_W \leftarrow$ LocalConfs$(W'_{\overline{\alpha}})$
  if $(W'_{\overline{\alpha}})_{|\backslash C_W} = \emptyset$ **then**
    $W_{\overline{\alpha}} \leftarrow W'_{\overline{\alpha}}$
    $W_\alpha \leftarrow (W'_\alpha \cup A)_{|\backslash C_W}$
  else
    $B \leftarrow \overline{\alpha}\text{-}Attr(G, (W'_{\overline{\alpha}})_{|\backslash C_W})$
    $C_B \leftarrow$ LocalConfs(B)
    $(W_0'', W_1'') \leftarrow$ RecursiveUPGLocal$(G \backslash B \backslash V_{||C_W \cup C_B}, \hat{v}_0, \Delta)$
    $W_\alpha \leftarrow W''_\alpha$
    $W_{\overline{\alpha}} \leftarrow W''_{\overline{\alpha}} \cup B \cup (W'_{\overline{\alpha}})_{||C_W}$
  end if
end if
return (W_0, W_1)

function LocalConfs$(X \subseteq V)$
  if $\overline{\alpha} \in \Delta$ **then**
    return $\{c \in \mathfrak{C} \mid (c, \hat{v}_0) \in X\}$
  else
    return $\emptyset$
  end if
end function

Theorem 7.8. *Given:*

- *VPG* $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$,
- *origin vertex* $\hat{v}_0 \in \hat{V}$,
- *total unified PG* $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ *that is a subgame of, or equal to, unified PG* $\hat{G}_{\downarrow}$,
- *configuration* $c \in \mathfrak{C}$,
- *winning sets* Q_0 *and* Q_1 *for game* $G_{|c}$,
- *a set of players* $\Delta \subseteq \{0, 1\}$ *and*
- *winning sets* $(W_0, W_1) = \textsc{RecursiveUPGLocal}(G, \hat{v}_0, \Delta)$

either or both of the following statements hold:

(I) For some $\beta \in \Delta$ *we have* $(c, \hat{v}_0) \in W_\beta$ *and* $\hat{v}_0 \in Q_\beta$.

(II) $(W_0)_{|c} = Q_0$ *and* $(W_1)_{|c} = Q_1$.

Proof. First observe that statement (II) means that the projection of G to c is solved correctly, if G is solved globally then statement (II) holds for any c.

We prove the theorem by comparing the behaviour of RecursiveUPGLocal with the behaviour of the original algorithm (RecursivePG) which solves a unified PG globally.

Prove by induction on G.

Base: If G has only one priority or no vertices then RecursiveUPGLocal behaves the same as the original algorithm so statement (II) holds.

Step: The algorithm makes a number of case distinction. In order to prove the correctness we inspect all the different paths through the algorithm. We use a Fitch-like notation to keep an overview of the different case distinctions and improve readability.

Case $G_{\|c}$ is empty.
Winning sets $(W_0)_{\|c}$ and $(W_1)_{\|c}$ are empty since the winning sets will be subsets of V and $V_{\|c}$ is empty. Clearly Q_0 and Q_1 are empty so the theorem holds.

Case $G_{\|c}$ is not empty.

Case $(c, \hat{v}_0) \notin V$

Vertex $(c, \hat{v}_0)$ is also not in any subgame of G. By induction we can conclude that either or both statements are true for subgame $G \backslash A$ with configuration c and sets (W'_0, W'_1). Since $(c, \hat{v}_0)$ is not in the subgame statement (II) must hold.

Case $W'_{\overline{\alpha}} = \emptyset$

In this case the algorithm returns all vertices to be winning for player α. By induction we found that indeed all vertices in game $(G \backslash A)_{|c}$ are won by player α. This means that when we consider game $G_{|c}$ player $\overline{\alpha}$ must play the token away from vertices $V_{|c} \backslash A_{|c}$ to win. When the play goes to vertices in $A_{|c}$ then player α can make sure that the play goes to one of the vertices with the highest priority. After that the play can either go to vertices in $A_{|c}$ or vertices in $V_{|c} \backslash A_{|c}$. In the former case play will again go to a vertex with the highest priority. In the latter case player $\overline{\alpha}$ must play back to $A_{|c}$ or loose the game, however playing to $A_{|c}$ causes the play to go to the highest vertex again. Since the highest vertex has a parity equal to α every play is won by player α and we can conclude that indeed every vertex in $G_{|c}$ is won by player α and that statement (II) holds.

Case $W'_{\overline{\alpha}} \neq \emptyset$

Since $(c, \hat{v}_0) \notin V$ we conclude $c \notin C_W$ and $((W'_{\overline{\alpha}})_{|\backslash C_W})_{|c} = (W'_{\overline{\alpha}})_{|c}$.

Case $(W'_{\overline{\alpha}})_{|\backslash C} = \emptyset$

All vertices in $V_{|c}$ are returned to be winning for player α. We argued that the C_W operations don't have an effect on the value of $(W'_{\overline{\alpha}})_{|c}$ therefore we can use the same argument as in case $W'_{\overline{\alpha}} = \emptyset$ to show that indeed all vertices in $V_{|c}$ are won by player α and that statement (II) holds.

Case $(W'_{\overline{\alpha}})_{|\backslash C} \neq \emptyset$

Again we find that $c \notin C_B$. Set $V \backslash A$ is an α-trap in G (using lemma 7.5), therefore $(V \backslash A)_{|c}$ is an α-trap in $G_{|c}$. Set $(W'_{\overline{\alpha}})_{|c}$ is an α-trap in $(G \backslash A)_{|c}$ (because it is a winning set). We can conclude that $(W'_{\overline{\alpha}})_{|c}$ is an α-trap in $G_{|c}$. Using lemma 7.6 we find that $B_{|c}$ is an α-trap in $G_{|c}$, meaning that player $\overline{\alpha}$ can keep the play in $B_{|c}$, moreover player $\overline{\alpha}$ can play towards $(W'_{\overline{\alpha}})_{|c}$ where this player wins. We can conclude that any vertex in $B_{|c}$ is won by player $\overline{\alpha}$ and there are no edges that go from a vertex owned by player $\overline{\alpha}$ in $V_{|c} \backslash B_{|c}$ to $B_{|c}$. There might be such an edge owned by player α, however if player α chooses plays along such an edge play ends up in $B_{|c}$ and player $\overline{\alpha}$ wins. Therefore we can simply solve game $G_{|c} \backslash B_{|c}$, the winning sets in this subgame are also winning in game $G_{|c}$.

The subgame created projected to c is $G_{|c} \backslash B_{|c}$, since $\hat{v}_0$ is not in $G_{|c}$ it is also not in $G_{|c} \backslash B_{|c}$. By induction we find that statement (II) holds for $G_{|c} \backslash B_{|c}$ with sets W''_0, W''_1. The algorithm returns $(W''_\alpha)_{|c}$ for player α and $(W''_{\overline{\alpha}})_{|c} \cup B_{|c}$ for player $\overline{\alpha}$, this correctly solves game $G_{|c}$ therefore statement (II) holds.

Case $(c, \hat{v}_0) \in V$

Case $\overline{\alpha} \notin \Delta$

In the first recursion we get $\Delta = \emptyset$, therefore by induction we conclude that statement (II) holds for game $G \backslash A$ with configuration c and sets (W_0', W_1').

Case $W'_{\overline{\alpha}} = \emptyset$

As argued before if $(G \backslash A)_{|c}$ is entirely won by player α then $G_{|c}$ is entirely won by player α. The algorithm returns all vertices to be winning for player α so statement (II) holds in this case.

Case $W'_{\overline{\alpha}} \neq \emptyset$

Since $\overline{\alpha} \notin \Delta$ LocalConfs will always return $\emptyset$. So $c \notin C_W$.

Case $(W'_{\overline{\alpha}})_{|\backslash C_W} = \emptyset$

Similarly, in this case $(G \backslash A)_{|c}$ is entirely won by player α and therefore statement (II) holds.

Case $(W'_{\overline{\alpha}})_{|\backslash C_W} \neq \emptyset$

As argued before when $c \notin C_W$ and $c \notin C_B$ then if a vertex is won by a certain player in subgame $(G \backslash B)_{|c}$ it is also won by that player in game $G_{|c}$.
Therefore we can conclude that if statement (I) holds for the subgame it holds for the game, similarly if statement (II) holds for the subgame it holds for the game. By induction at least one holds so the theorem holds in this case.

Case $\overline{\alpha} \in \Delta$

The first recursion is called with $\Delta = \{\overline{\alpha}\}$. By induction we find that either or both statements hold for game $G \backslash A$, configuration c and sets (W_0', W_1').

Case $W'_{\overline{\alpha}} = \emptyset$

In this case statement (II) must hold. And similar as before we find that indeed all vertices in $G_{|c}$ are won by player α.

Case $W'_{\overline{\alpha}} \neq \emptyset$

Case $c \in C_W$

Using lemma 7.1 we find that anything won in subgame $G \backslash A$ by player $\overline{\alpha}$ is also won by that player in game G. We find $(c, \hat{v}_0)$ to be won by player $\overline{\alpha}$ in game $G \backslash A$ so this vertex is also won by player $\overline{\alpha}$ in game G. The algorithm returns $W'_{\overline{\alpha}}$ or $(W'_{\overline{\alpha}})_{||C_W}$ to be won by player $\overline{\alpha}$, this includes $(c, \hat{v}_0)$ so statement (I) holds.

Case $c \notin C_W$

Statement (II) must hold for game $G \backslash A$ with configuration c and sets (W_0', W_1') because if statement (I) would hold we would have had $c \in C_W$.

Case $(W'_{\overline{\alpha}})_{|\backslash C_W} = \emptyset$

We find that game $(G \backslash A)_{|c}$ is entirely won by player α and therefore game $G_{|c}$ is entirely won by player α so returning all vertices with configuration c to be winning for player α makes statement (II) true.

Case $(W'_{\overline{\alpha}})_{|\backslash C_W} \neq \emptyset$

Case $c \in C_B$

Anything won by player $\overline{\alpha}$ in game $G \backslash A$ is also won by that player in game G. Set B contains vertices such that player $\overline{\alpha}$ can force the play to a vertex that is won by player $\overline{\alpha}$ in game G, therefore anything in B is also won by player $\overline{\alpha}$ in game G. The algorithm returns everything in B to be winning for player $\overline{\alpha}$, so statement (I) holds.

Case $c \notin C_B$

As argued before when $c \notin C_W$ and $c \notin C_B$ then if a vertex is won by a certain player in subgame $(G \backslash B)_{|c}$ it is also won by that player in game $G_{|c}$.
Therefore we can conclude that if statement (I) holds for the subgame it holds for the game, similarly if statement (II) holds for the subgame it holds for the game. By induction at least one holds so the theorem holds in this case.

□

Using these two theorems we can conclude that RecursiveUPGLocal($\hat{G}_{\downarrow}, \hat{v}_0, \{0,1\}$) correctly locally solves VPG $\hat{G}$ for vertex $\hat{v}_0$.

The pseudo code presented for the algorithm uses a set-wise representation of unified PGs. As we have seen previously in the RecursiveUPG algorithm we can modify the recursive algorithm to use a function-wise representation with a function-wise attractor set calculation. The RecursiveUPGLocal algorithm can be transformed in the same way. The RecursiveUPGLocal algorithm introduces defintions for projections to sets of configurations as well as the LocalConfs subroutine, we introduce function-wise variants for these definitions and subroutine.

Definition 7.5. *Given function $X : \hat{V} \to 2^{\mathfrak{C}}$ we define the projection of X to $C \subseteq \mathfrak{C}$, denoted by $X_{||C}$, as*

$$X_{||C}(\hat{v}) = X(\hat{v}) \cap C$$

Definition 7.6. *Given function $X : \hat{V} \to 2^{\mathfrak{C}}$ we define the complementary projection of X to $C \subseteq \mathfrak{C}$, denoted by $X_{|\setminus C}$, as*

$$X_{||C}(\hat{v}) = X(\hat{v}) \setminus C$$

Algorithm 10 shows a function wise implementation of the LocalConfs subroutine. It is trivial to see that this algorithm and the projection definitions are equal under the $=_\lambda$ operator to their set-wise counterparts.

Algorithm 10 Function-wise LocalConfs subroutine

```
1: function FLocalConfs(X : V̂ → 2^𝔈)
2:     if ᾱ ∈ Δ then
3:         return X(v̂_0)
4:     else
5:         return ∅
6:     end if
7: end function
```

We can solve a VPG locally using this local recursive algorithm for unified parity games. We can either represent the PGs set-wise or we can present them function-wise in which case we can represent the sets of configurations explicit or symbolic. In all three cases the time complexity is identical to their global counterparts because in the worst case the vertex we are searching for is never won by player $\overline{\alpha}$ at any recursion level so we still have to solve the game globally. Furthermore the added projection operations are subsumed in time complexity by the attractor set calculation so the time complexity argumentation presented for the global variants is also valid for the local variants.

7.2.2 Local incremental pre-solve algorithm

The incremental pre-solve algorithm is particularly appropriate for local solving because if we find v_0 in P_α then we know that v_0 is won by player α for every configuration, therefore we are done for that particular recursion. This can potentially reduce the recursion depth of the algorithm and therefore reduce the number of (pessimistic) games solved.

Furthermore when there is only one configuration the incremental pre-solve algorithm solves the corresponding parity game. When taking a local approach it is sufficient to solve this PG locally. Note that the pessimistic games still must be solved globally to find as much assistance as possible for further recursions.

Clearly it could be the case that v_0 is not found in either P_0 or P_1 and is only solved when there is one configuration left. In such a case the local algorithm behaves the same as the global algorithm so we have an identical time complexity, ie. $O(c * e * n^d)$.

8 Experimental evaluation

The algorithms proposed to solve VPGs collectively have the same or a worse time complexity than the independent approach. The aim of the collective algorithms is to solve VPGs effectively when there are a lot of commonalities between configurations. A worst-case time complexity analyses doesn't say much about the performance in case there are many commonalities. In order to evaluate actual running time on different types of VPGs the algorithms are implemented and a number of test VPGs are created to test the performance on. In this next few sections we discuss the implementation and look at the results.

8.1 Implementation

The algorithms are implemented in C++ version 14 and use BuDDy[1] as a BDD library. The complete source is hosted on github[2].

The implementation is split in three phases: parsing, solving and solution printing. The solving part contains the implementations of the algorithms presented. The parsing and solution printing parts are implemented trivially and hardly optimized, their running times are not considered in the experimental evaluation.

The parsing phase of the algorithm creates BDDs from the input file, in doing so parts of the BDD cache gets filled. After parsing the BDD cache is cleared to make sure that the work done in the solving phase corresponds with the algorithms presented and no work to assist it has been done prior to this phase.

8.1.1 Game representation

The graph is represented by using adjacency lists for incoming and outgoing edges, furthermore every edge maps to a set of configurations representing the θ value for the edge (either explicitly or symbolically). For product based algorithms the edges are not mapped to sets of configurations. Finally sets of vertices are represented using bit-vectors.

Note that only the representation of the games used during the algorithm is relevant, since we don't evaluate the parsing phase it is not relevant how the games are stored in a file.

8.1.2 Product based algorithms

Three product based algorithms are implemented, ie. algorithms that solve parity games:

- Zielonka's recursive algorithm.
- Fixed-point iteration algorithm, global variant.
- Fixed-point iteration algorithm, local variant.

A few optimizations are applied to the fixed-point iteration algorithm. The following three are described in [3]:

- For fixed-point variable Z_i its value is only ever used to check if a vertex with priority i is in Z_i. So instead of storing all vertices in Z_i we only have to store the vertices that have priority i. We can store all fixed-point variables in a single bit-vector, named Z, of size n.
- The algorithm only reinitializes a certain range of fixed-point variables. So the diamond and box operations can use the previous result and only reconsider vertices that have an edge to a vertex that has a priority for which its fixed-point variable is reset.
- The algorithm updates variables Z_0 to Z_m and reinitializes Z_0 to Z_{m-1}, however if Z_m is a least fixed-point variable then Z_m has just increased and due to monotonicity the other least fixed-point formula's, ie. $Z_{m-2}, Z_{m-4}, \ldots$, will also increase so there is no need to reset them. Similarly for greatest fixed-point variables. So we only to reset half of the variables instead of all of them.

[1] https://sourceforge.net/projects/buddy/

[2] https://github.com/SjefvanLoo/VariabilityParityGames/tree/master/implementation/VPGSolver

Finally the vertices in the game are reordered such that they are sorted by parity first and by priority second. Using the above optimizations the algorithm needs to reset variables $Z_m, Z_{m-2}, \ldots$, these variables are stored in a single bit-vector Z. By reordering the variables to be sorted by parity and priority these vertices that need to be reset are always consecutively stored in Z, so resetting this sequence can be done by a memory copy instead of iterating all the different vertices. Note that when the algorithm is used by the pre-solve algorithm the variables are not reset to simply $\emptyset$ and V but are reset to two specific bit-vectors that are given by the pre-solve algorithm. These bit-vectors have the same order and resetting can be done by copying a part of them into Z.

The advantage of using a memory copy as opposed to iteration all the different vertices is due to the fact that a bit vector uses integers to store its boolean values. A 32-bit integer can store 32 boolean values. Iterating and writing every boolean value individually causes the integer to be written 32 times. However with a memory copy we can simply copy the entire integer value and the integer is only written once.

8.1.3 Family based algorithms

Four family based algorithms are implemented:

- Zielonka's recursive family based algorithm with explicit configuration set representation.
- Zielonka's recursive family based algorithm with symbolic configuration set representation.
- Incremental pre-solve algorithm, global variant.
- Incremental pre-solve algorithm, local variant.

The incremental pre-solve algorithms use the fixed-point iteration algorithm as described above to solve the (pessimistic) PGs.

8.1.4 Random verification

In order to prevent implementation mistakes 200 VPGs are created randomly, every VPG is projected to all its configurations to get a set of PGs. These PGs are solved using the PGSolver tool ([10]). All algorithms implemented are used to solve the 200 VPGs and the solutions are compared to the solutions created by the PGSolver.

8.2 Test cases

The algorithms are used to solve VPGs created from different test cases. We have two model checking problems as well as random VPGs. The model checking VPGs are created as described in part 5, with the exception that only vertices are added when they are reachable from the initial vertex. So these games are never disjointed, random games can be disjointed.

8.2.1 Model checking games

We use two software product line examples, first the minepump example as described in [12] and implemented in the mCRL2 toolset as described in [18]. The minepump example models the behaviour of controllers for a pump that pumps water out of a mineshaft. There are 10 different features that change the way the sensors/actors behave. In total there are 128 valid feature assignments, ie. products.

The mCRL2 implementation creates an LTS with parametrized actions, the parameters describe the boolean formula's guarding the transitions, effectively making it an FTS. This FTS is interpreted in combination with 9 different μ-calculus formula's to create 9 VPGs. The FTS consists of 582 states and 1376 transitions. The resulting VPGs consist 128 configurations and on average 6600 vertices per game.

Next we have the elevator example, described in [14]. This example models the behaviour of an elevator where 5 different features modify the behaviour of the model. All feature assignments are valid therefore we have $2^5 = 32$ feature assignments, ie. products. Again an mCRL2 implementation[3] (created by T.A.C. Willemse) is used to create 7 VPGs. The FTS consists 95591 states and 622265 transitions. The resulting VPGs consist of 32 configurations and on average TODO vertices per game.

[3] https://github.com/SjefvanLoo/VariabilityParityGames/blob/master/implementation/Elevator.tar.gz

8.2.2 Random games

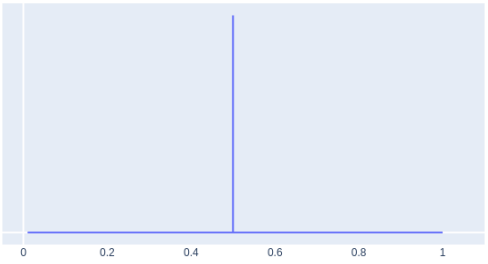
Random VPGs can be created by creating a random parity game and creating sets of configurations that guard the edges. For these sets we need to consider two factors: how large are the sets guarding the edges and how are they created.

The guard sets in the minepump and elevator games have a very specific distribution where of the sets admit either 100% or 50% of the configurations. An edge requiring the presence or absence of one specific feature results in a set admitting 50%, this explains the distribution. The average edge in the examples admits 92% of the configurations.

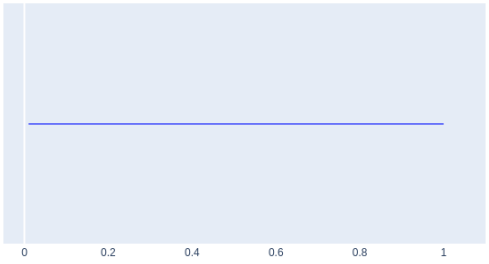
We use λ to denote the average relative size of guard sets in a VPG. We create a random game with a specific λ by using a probabilistic distribution ranging from 0 to 1 to determine the size of an individual guard set. Such a distribution must have a mean equal to λ. We consider two distributions:

- A modified Bernoulli distribution; in a Bernoulli distribution there is a probability of p to get an outcome of 1 and a probability of $1-p$ to get an outcome of 0. We modify this such that there is a probability of p to get 1 and a probability of $1-p$ to get 0.5. This gives a mean of $1p + 0.5(1-p) = 0.5p + 0.5$. So to get a mean of λ we choose $p = 2\lambda - 1$. Note that we can't use this distribution when $\lambda < 0.5$ because p becomes less than 0.
- A beta distribution; a beta distribution ranges from 0 to 1 and is curved such that it has a specific mean. The beta distribution has two parameters: α and β and a mean of $\frac{\alpha}{\alpha+\beta}$. We pick $\beta = 1$ and $\alpha = \frac{\lambda\beta}{1-\lambda}$ to get a mean of λ.

Figures 17, 18 and 19 show the shapes of the distribution for different values for λ.

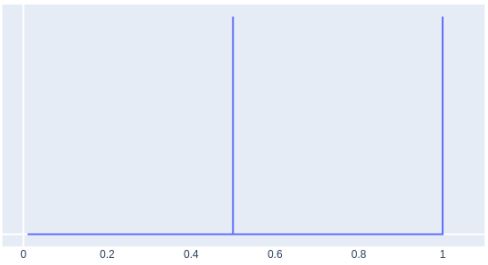


(a) Modified Bernoulli distribution with $p = 0$

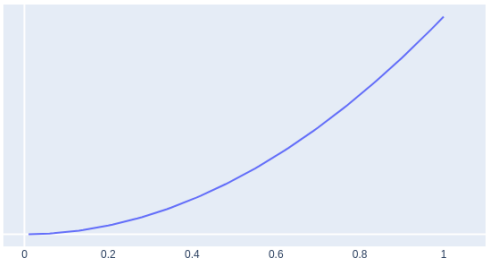


(b) Beta distribution with $\beta = 1$ and $\alpha = 1$

Figure 17: Edge guard size distribution for $\lambda = 0.5$

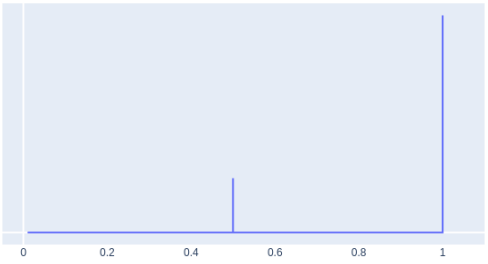


(a) Modified Bernoulli distribution with $p = 0.5$

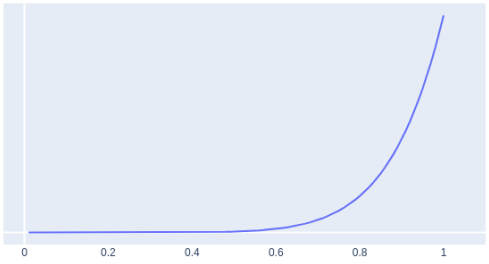


(b) Beta distribution with $\beta = 1$ and $\alpha = 3$

Figure 18: Edge guard size distribution for $\lambda = 0.75$



(a) Modified Bernoulli distribution with $p = 0.8$



(b) Beta distribution with $\beta = 1$ and $\alpha = 9$

Figure 19: Edge guard size distribution for $\lambda = 0.9$

Once we have determined the size for a configuration set we need to consider how the set is created. We can simply create a random set of configurations without any notion of features, we call this a *configuration based* approach. Alternatively we can use a *feature based* approach where we create sets by looking at features. Consider features $f_0, \ldots, f_m$, we can create a boolean function that is the conjunction of k features where every feature in the conjunction has probability 0.5 of being negated. For example when using $k = 3$ and $m = 5$ we might get boolean formula $f_1 \wedge \neg f_2 \wedge \neg f_4$. Such a boolean formula corresponds to a set of configurations of size 2^{m-k} and a relative size $\frac{2^{m-k}}{2^m} = 2^{-k}$, so when creating a set with relative size r we choose $k = \min(m, \lfloor -\log_2 r \rfloor)$. When using a feature based approach we can only create sets that have a relative size of $\frac{1}{2^i}$ for some $i \in \mathbb{N}$.

We can create 4 types of games:

1. Bernoulli distributed and feature based. These games are most similar to the model verification games.
2. Bernoulli distributed and configuration based. These games do have the characteristics of a model verification game in terms of set size but have unstructured sets guarding the edges. Furthermore with a configuration based approach less guard sets will be identical than with a feature based approach.
3. Beta distributed and configuration based. These games are most different from the model verification games.

4. Beta distributed and feature based. Games created in this way don't have the average relative set size of λ because the feature based approach can only create sets of size $\frac{1}{2^i}$ for any $\lambda \geq \frac{1}{2}$ almost all the sets have either relative size $\frac{1}{2}$ or 1, so this creates almost the exact same games as using the Bernoulli distribution only with an incorrect average relative set size. Therefore we will not consider this category of games.

For random games of type 1,2 and 3 we create 50 games: game 50 to game 99, where game i has $\lambda = \frac{i}{100}$ and a random number of features, nodes, edges and maximum priority.

Furthermore we create 48 games to evaluate how the algorithm scales when the number of features becomes larger. For every $i \in [2, 12]$ we create random games ia, ib, ic and id of type 1 with $\lambda = 0.92$, i features and a random number of nodes, edges and maximum priority.

8.3 Results

In this section the experimental results are presented. The following algorithms are used to solve the 6 categories of games:

1. Zielonka's recursive algorithm, product based
2. Fixed-point iteration, product based
3. Fixed-point iteration, local product based
4. Zielonka's recursive algorithm, family based with explicit configuration representation
5. Zielonka's recursive algorithm, family based with symbolic configuration representation
6. Incremental pre-solve algorithm
7. Incremental pre-solve algorithm, local

The numbers presented are the times it took to solve the games; times for parsing, projecting (for product based approaches) and printing solutions are excluded. So the product based results are the sum of the solve times of the projections, parsing and projecting are not included in the result.

The exact times can be found in appendix A, in this section the results are visualized and presented to be easily interpreted. In some cases the results in a graph are normalized meaning that the running times are divided by the running times of the first algorithm in the graph. Specifically for the random games the running times vary a lot so normalizing is required to properly visualize the results. The times are presented on a logarithmic scale, games that were unable to be completed (due to memory) are marked with a grey bar.

All the experiments are ran on a Linux x64 operating system with an Intel i5-4570 processor and 8GB of RAM.

8.3.1 Zielonka's family based

First we compare the running times of the Zielonka's family based approaches with the Zielonka's product based approach. First we look at the model verification games.

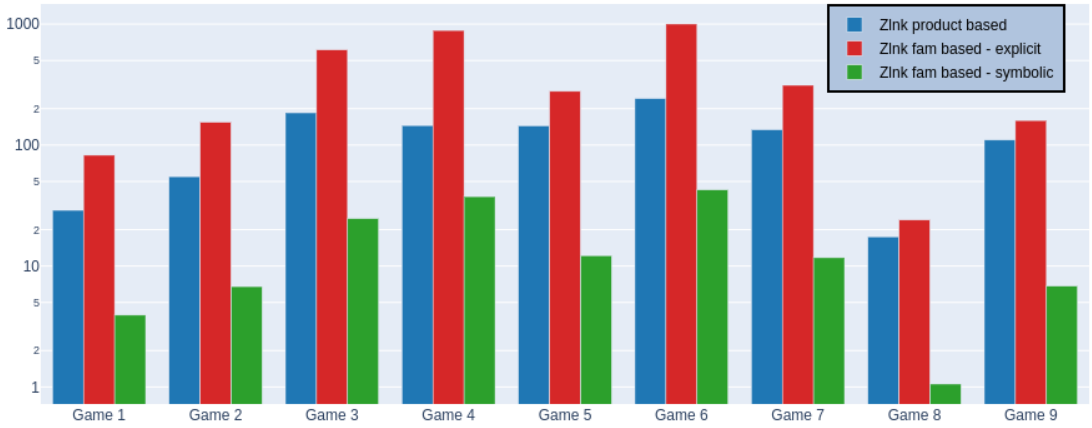


Figure 20: Running time of Zielonka's algorithms on the minepump problem

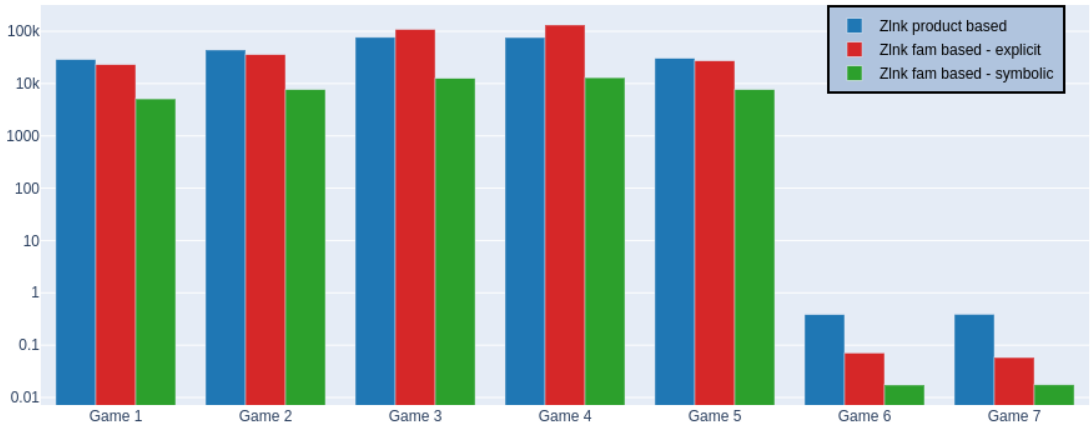


Figure 21: Running time of Zielonka's algorithms on the elevator problem

We can see that the performance of the explicit variant varies a lot between games. The symbolic variant greatly outperforms the product based approach for every problem.

Next we inspect the random games, first we look at the games with a variable λ and a random number of features. The graphs are normalized.

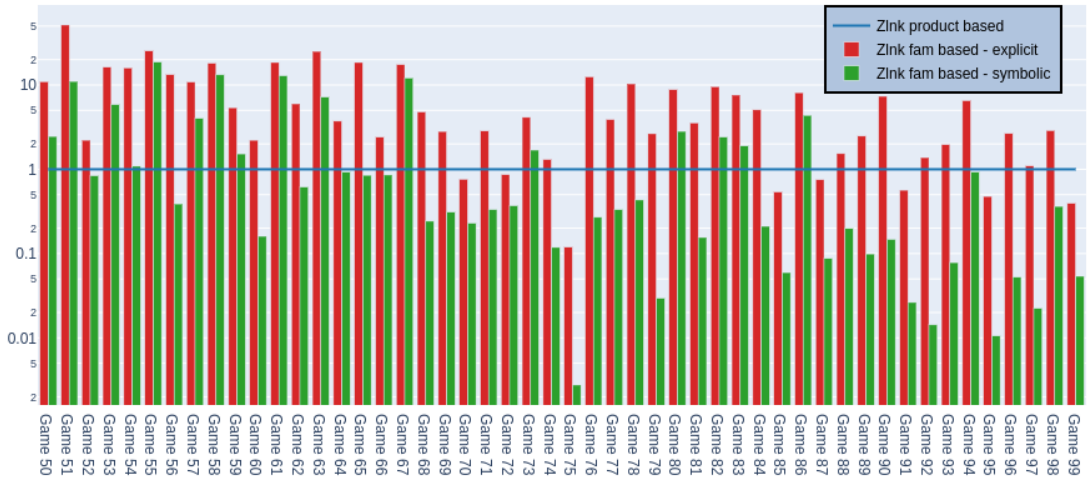


Figure 22: Running time of Zielonka's algorithms on randomgames of type 1 with $\lambda = \frac{game\ nr}{100}$, times are normalized

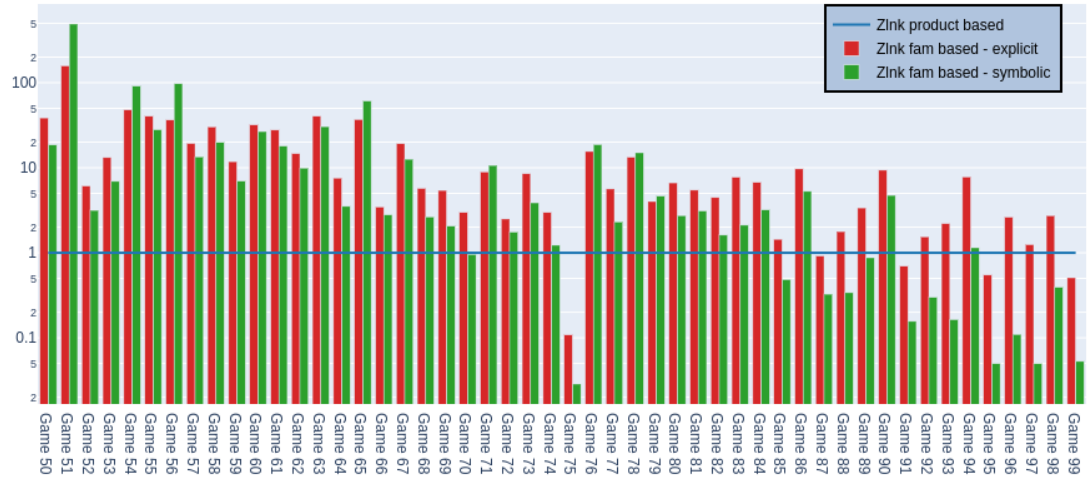


Figure 23: Running time of Zielonka's algorithms on randomgames of type 2 with $\lambda = \frac{game\ nr}{100}$, times are normalized

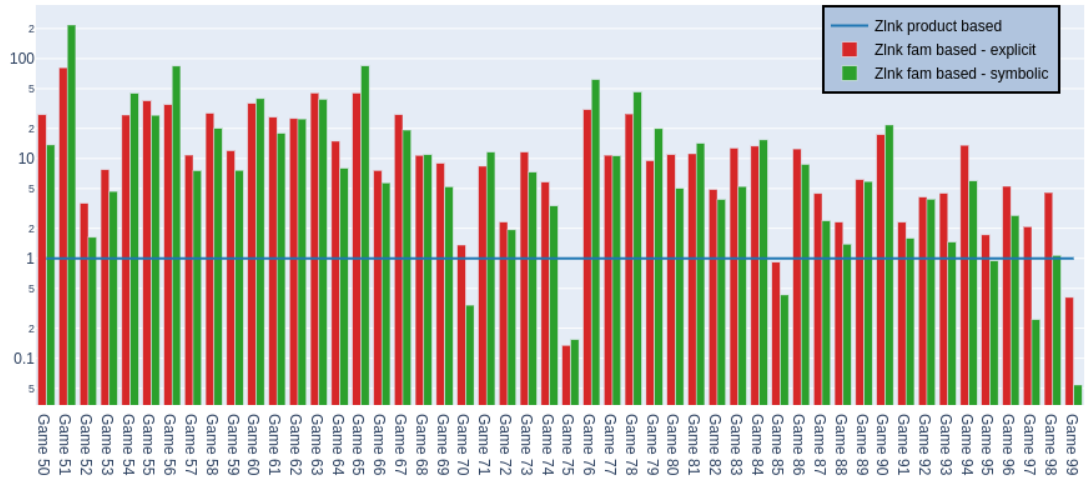


Figure 24: Running time of Zielonka's algorithms on randomgames of type 3 with $\lambda = \frac{game\ nr}{100}$, times are normalized

For type 1 games we see that when λ gets bigger the family based symbolic approach starts winning from the product based approach. There are a few exceptions to this, games 80, 82 and 86, all three of these games have only 4 features. As we will see later the less features there are in a game the worse the family based approaches perform.

For type 2 the explicit variant performs very similar to the type 1 games, however the symbolic approach performs much worse. This is due to the unstructured nature of the configuration sets which negatively influences bdd performance but has no effect on the explicit set operations. We also see the explicit algorithm outperforming the symbolic algorithm in some cases.

For type 3 games the product based approach performs generally better than the family based approaches unless λ becomes very high.

Next we inspect how the algorithms scales in terms of number of features

Figure 25: Running time of Zielonka's algorithms on randomgames of type 1 with $\lambda = 0.92$ and the number of features equal to the *game nr*, times are normalized

We can clearly conclude that as the number of features increases the family based symbolic approach performs better compared to the product based approach.

Overall we can conclude that the explicit algorithm performs somewhat arbitrary, however the symbolic algorithm performs really well for model checking problems and random games that have similar properties. Also we can conclude that the algorithms scales well in terms of number of features.

8.3.2 Incremental pre-solve algorithm

We compare the running times of the incremental pre-solve approaches with the fixed-point iteration product based algorithm. First we look at the model verification games.

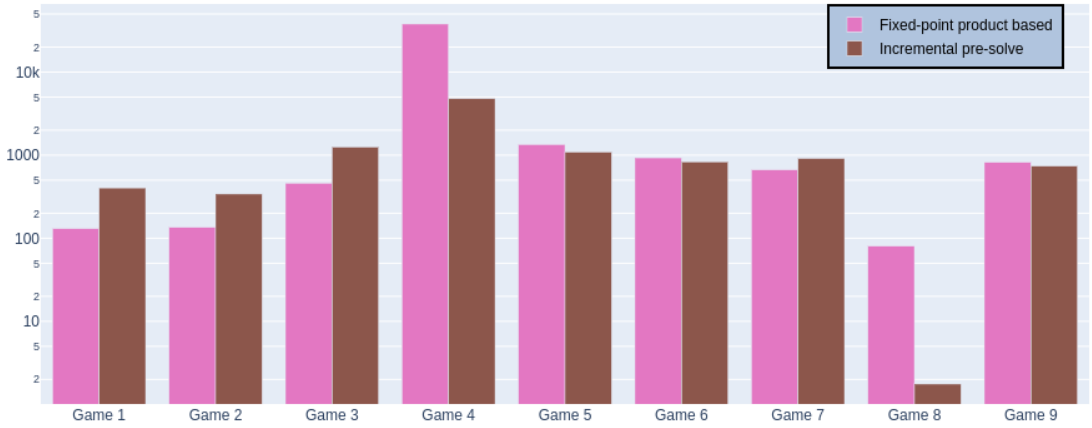


Figure 26: Running time of the incremental pre-solve algorithms on the minepump problem

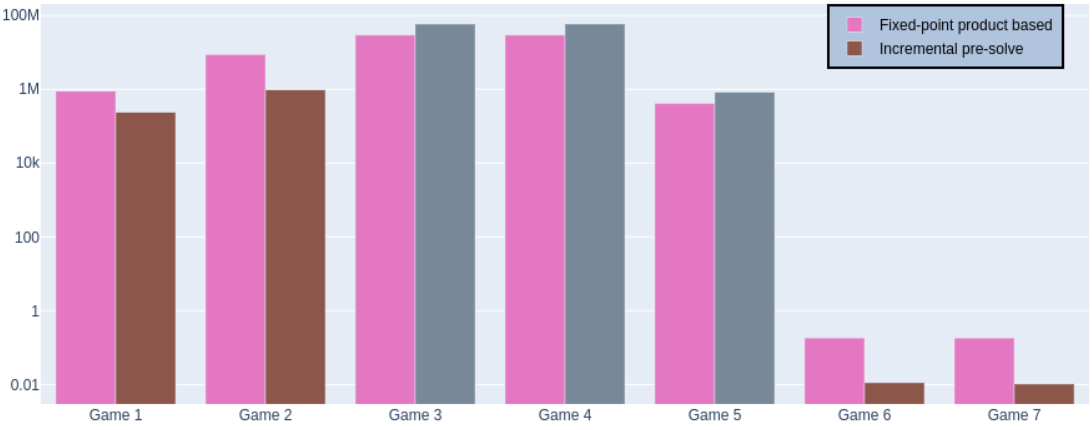


Figure 27: Running time of the incremental pre-solve algorithms on the elevator problem. Game 3, 4 and 5 were not able to be finished

The incremental pre-solve algorithms doesn't show any significant improvements over the product based fixed-point algorithm.

Next we inspect the random games, first we look at the games with a variable λ and a random number of features. The graphs are normalized.

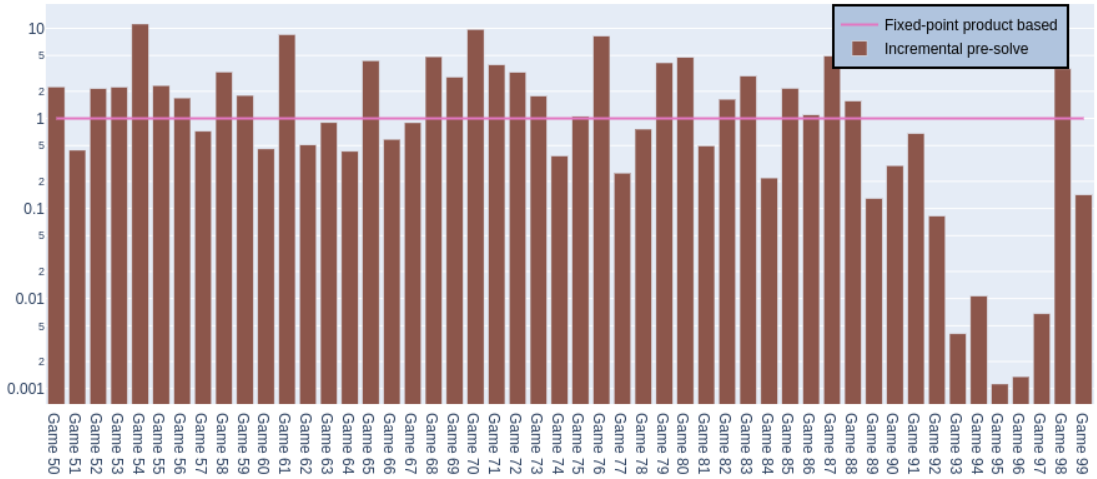


Figure 28: Running time of the incremental pre-solve algorithms on randomgames of type 1 with $\lambda = \frac{game\ nr}{100}$, times are normalized

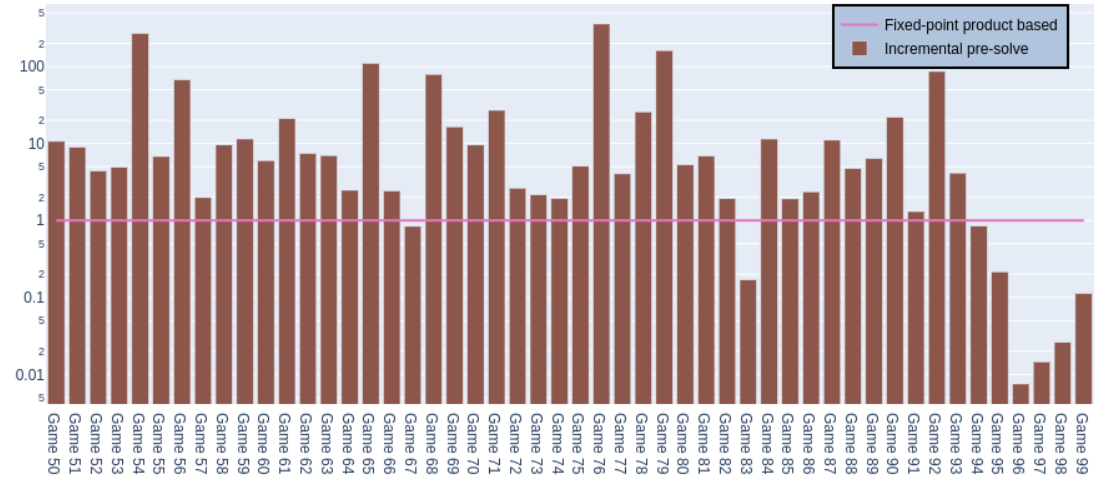


Figure 29: Running time of incremental pre-solve algorithms on randomgames of type 2 with $\lambda = \frac{game\ nr}{100}$, times are normalized

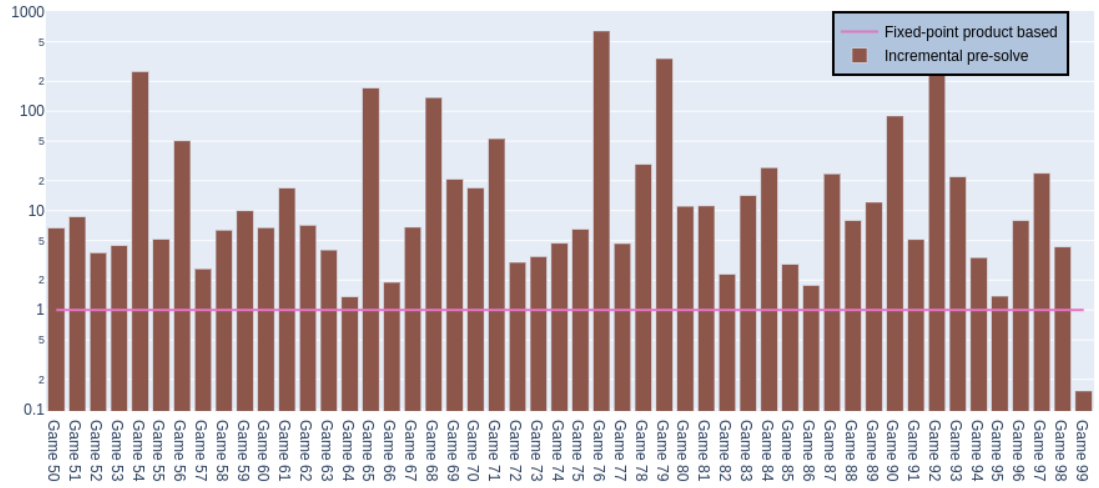


Figure 30: Running time of incremental pre-solve algorithms on randomgames of type 3 with $\lambda = \frac{game\ nr}{100}$, times are normalized

For type 1 and type 2 games we see that when λ gets bigger the incremental pre-solve start performing better, however it this trends only appears when λ get larger than about 0.95.

Again we see that for type 3 games the family based algorithm performs worse.

Next we inspect how the algorithms scales in terms of number of features

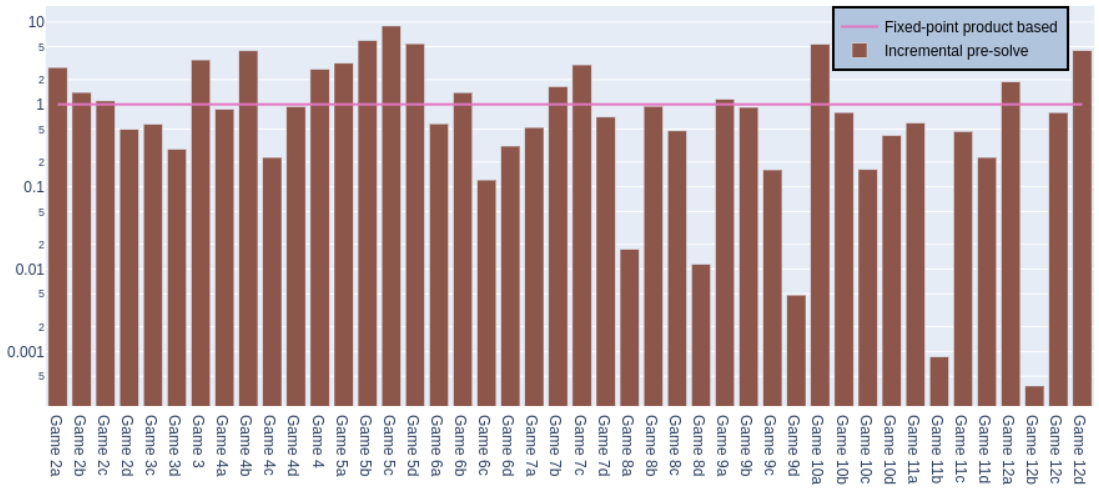


Figure 31: Running time of incremental pre-solve algorithms on randomgames of type 1 with $\lambda = 0.92$ and the number of features equal to the *game nr*, times are normalized

There is a downwards trend when it comes to the number of features, however it is quite inconsistent. Note that these games have $\lambda = 0.92$. Which is lower than the threshold observed in figure 28.

Overall the performance is not a significant improvement on the product based approach.

8.3.3 Incremental pre-solve local algorithm

Next we again compare the incremental pre-solver with the fixed-point iteration product based approach, however this time we compare the local variants. We start with the model-checking games.

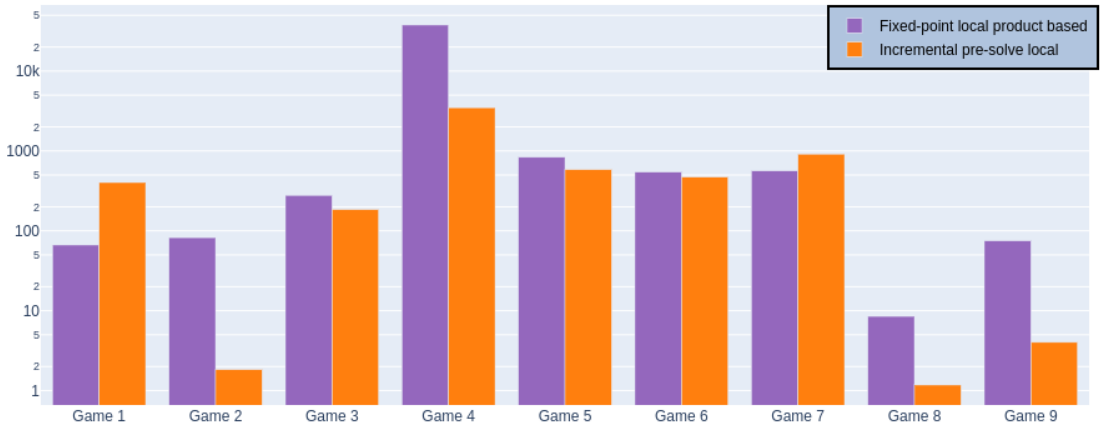


Figure 32: Running time of the incremental pre-solve local algorithms on the minepump problem

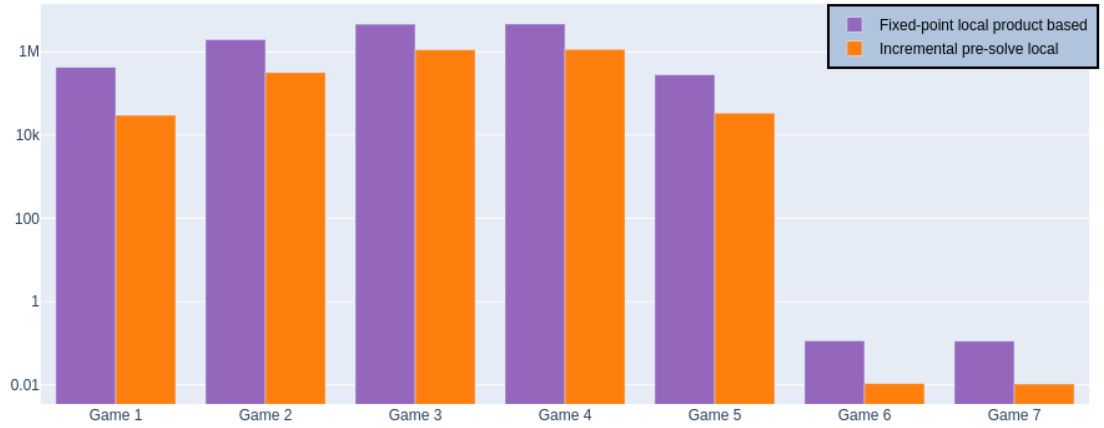


Figure 33: Running time of the incremental pre-solve local algorithms on the elevator problem

Generally the performance of the incremental pre-solver local algorithm is better, also games 3, 4 and 5 for the elevator problem were solved which was not the case for the global algorithm.

Next we inspect the random games, first we look at the games with a variable λ and a random number of features. The graphs are normalized.

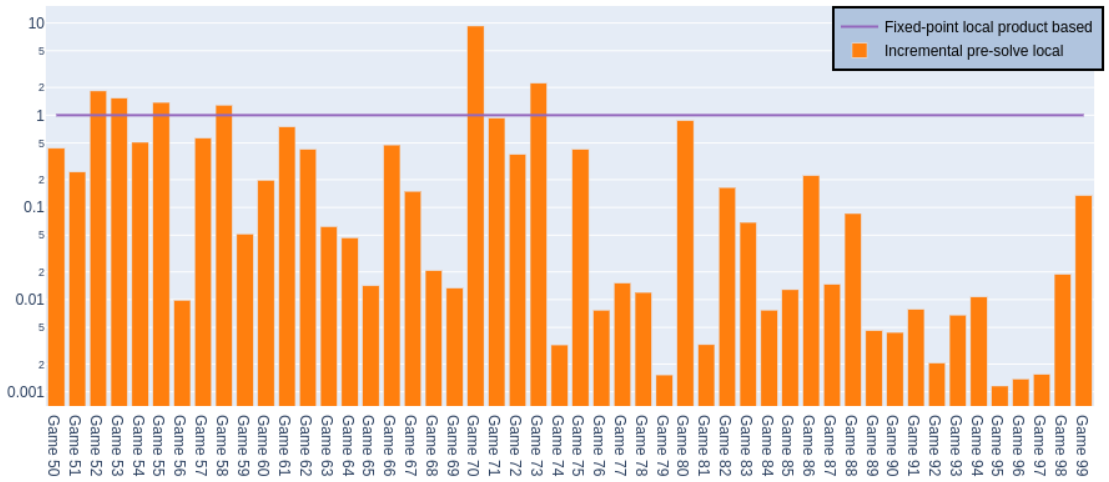


Figure 34: Running time of the incremental pre-solve local algorithms on randomgames of type 1 with $\lambda = \frac{game\ nr}{100}$, times are normalized

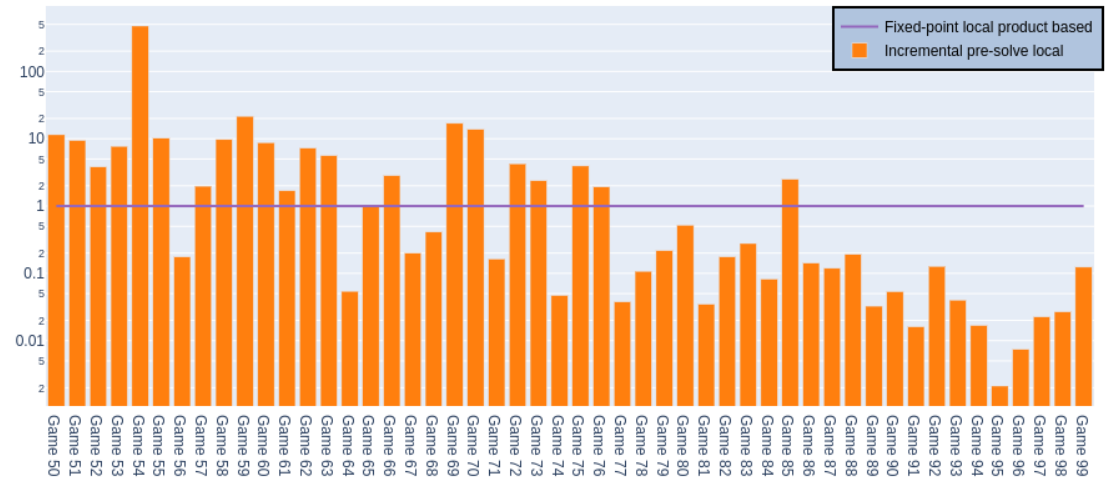


Figure 35: Running time of incremental pre-solve local algorithms on randomgames of type 2 with $\lambda = \frac{game\ nr}{100}$, times are normalized

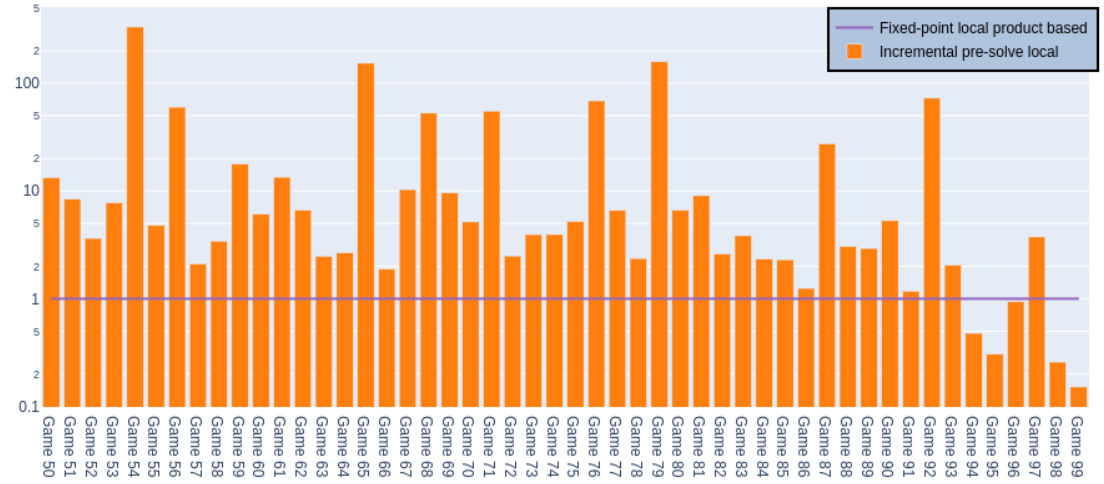


Figure 36: Running time of incremental pre-solve local algorithms on randomgames of type 3 with $\lambda = \frac{game\ nr}{100}$, times are normalized

For type 1 games the local incremental pre-solve algorithm greatly outperforms the product based approach, for type 2 games this is still the case but we can see differences becoming a bit smaller. Finally for type 3 games it only outperforms the product based approach when λ gets sufficiently high.

Next we inspect how the algorithms scales in terms of number of features

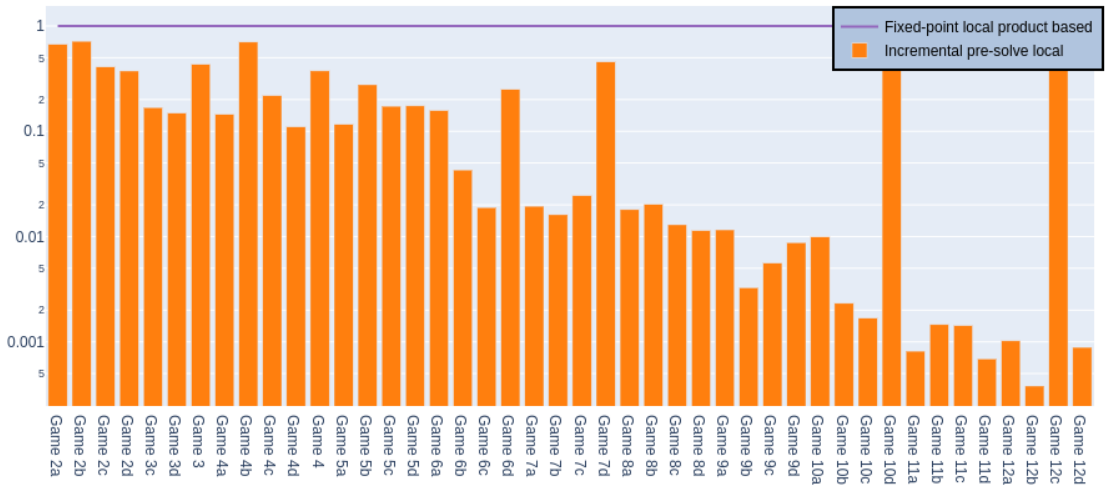


Figure 37: Running time of incremental pre-solve local algorithms on randomgames of type 1 with $\lambda = 0.92$ and the number of features equal to the *game nr*, times are normalized

Clearly the performance improves when the number of features becomes larger.

We can conclude that for model-checking games the algorithm performs decent, however the biggest

improvements are seen on random games.

8.3.4 Comparison

After comparing family based and product based variants of the same algorithm we compare the algorithms overall. We do so by comparing the mean solve time of the for every set of games. The incremental pre-solve global algorithm didn't complete 3 games, for these games we use the fixed-point product based global times to calculate the mean.

First we compare the product based algorithms:

	Zlnk product based	Fixed-point product based global	Fixed-point product based local
Minepump	118 ms	4719 ms	4469 ms
Elevator	36540 ms	9842835 ms	1713073 ms
Random games type 1	55 ms	413 ms	324 ms
Random games type 2	54 ms	413 ms	381 ms
Random games type 3	55 ms	416 ms	351 ms
Random scale games	108 ms	408 ms	381 ms

Next we compare the family based algorithms:

	Zlnk fam based explicit	Zlnk fam based symbolic	Incremental pre-solve global	Incremental pre-solve local
Minepump	391 ms	16 ms	1154 ms	672 ms
Elevator	46775 ms	6592 ms	8612914 ms	381041 ms
Random games type 1	513 ms	59 ms	274 ms	40 ms
Random games type 2	1149 ms	2569 ms	8283 ms	1507 ms
Random games type 3	1117 ms	2114 ms	12733 ms	4365 ms
Random scale games	397 ms	9 ms	205 ms	24 ms

9 Conclusion

Appendices

A Running time results

A.1 minepump

	Zlnk product based
Game 1	28.882998 ms
Game 2	54.796994 ms
Game 3	184.780354 ms
Game 4	145.049639 ms
Game 5	144.539523 ms
Game 6	242.921996 ms
Game 7	134.345216 ms
Game 8	17.444557 ms
Game 9	110.081099 ms

	Fixed-point product based
Game 1	130.587291 ms
Game 2	135.382598 ms
Game 3	460.184224 ms
Game 4	37898.67001 ms
Game 5	1346.308334 ms
Game 6	928.083238 ms
Game 7	669.829938 ms
Game 8	80.328725 ms
Game 9	823.229266 ms

	Fixed-point local product based
Game 1	66.477775 ms
Game 2	82.189287 ms
Game 3	278.245894 ms
Game 4	37764.888965 ms
Game 5	833.712816 ms
Game 6	544.744721 ms
Game 7	562.618467 ms
Game 8	8.503434 ms
Game 9	75.415805 ms

	Zlnk fam based - explicit
Game 1	82.447263 ms
Game 2	154.849897 ms
Game 3	615.161096 ms
Game 4	886.476284 ms
Game 5	278.163607 ms
Game 6	1001.927374 ms
Game 7	312.183059 ms
Game 8	24.019659 ms
Game 9	159.633291 ms

	Zlnk fam based - symbolic
Game 1	3.92863 ms
Game 2	6.760552 ms
Game 3	24.704502 ms
Game 4	37.467636 ms
Game 5	12.192468 ms
Game 6	42.792651 ms
Game 7	11.718343 ms
Game 8	1.058332 ms
Game 9	6.853061 ms

	Incremental pre-solve
Game 1	402.898316 ms
Game 2	341.198461 ms
Game 3	1255.949764 ms
Game 4	4803.836362 ms
Game 5	1092.277622 ms
Game 6	831.198424 ms
Game 7	917.083359 ms
Game 8	1.761178 ms
Game 9	742.183235 ms

	Incremental pre-solve local
Game 1	402.728056 ms
Game 2	1.827667 ms
Game 3	184.726048 ms
Game 4	3480.033761 ms
Game 5	583.474132 ms
Game 6	473.593019 ms
Game 7	914.134888 ms
Game 8	1.174847 ms
Game 9	4.019017 ms

A.2 elevator

	Zlnk product based
Game 1	29073.014986 ms
Game 2	43881.36677 ms
Game 3	76465.44404 ms
Game 4	75902.19748 ms
Game 5	30456.61109 ms
Game 6	0.386628 ms
Game 7	0.390919 ms

	Fixed-point product based
Game 1	880895.014563 ms
Game 2	8790000.10349 ms
Game 3	29387446.6626 ms
Game 4	29423372.371 ms
Game 5	418128.687347 ms
Game 6	0.182554 ms
Game 7	0.182125 ms

	Fixed-point local product based
Game 1	423068.957211 ms
Game 2	1983814.2006 ms
Game 3	4635988.08945 ms
Game 4	4666038.77282 ms
Game 5	282602.487182 ms
Game 6	0.11491 ms
Game 7	0.111502 ms

	Zlnk fam based - explicit
Game 1	23043.749564 ms
Game 2	35840.520344 ms
Game 3	109270.750667 ms
Game 4	131809.825388 ms
Game 5	27461.829276 ms
Game 6	0.071153 ms
Game 7	0.058017 ms

	Zlnk fam based - symbolic
Game 1	5105.955006 ms
Game 2	7716.33222 ms
Game 3	12591.04199 ms
Game 4	12980.005372 ms
Game 5	7751.122504 ms
Game 6	0.017326 ms
Game 7	0.017462 ms

	Incremental pre-solve
Game 1	236171.44533 ms
Game 2	960813.557869 ms
Game 3	†
Game 4	†
Game 5	†
Game 6	0.011157 ms
Game 7	0.0103 ms

	Incremental pre-solve local
Game 1	29941.840644 ms
Game 2	317217.608246 ms
Game 3	1133904.4743 ms
Game 4	1152374.57835 ms
Game 5	33845.973375 ms
Game 6	0.010736 ms
Game 7	0.010461 ms

A.3 Randomgames type 1

	Zlnk product based
Game 50	12.228259 ms
Game 51	181.379715 ms
Game 52	2.14455 ms
Game 53	1.439006 ms
Game 54	37.311392 ms
Game 55	2.712425 ms
Game 56	195.442821 ms
Game 57	7.462226 ms
Game 58	2.260288 ms
Game 59	9.033436 ms
Game 60	39.138796 ms
Game 61	2.380189 ms
Game 62	46.281275 ms
Game 63	17.259779 ms
Game 64	6.796232 ms
Game 65	140.116512 ms
Game 66	12.213586 ms
Game 67	2.650634 ms
Game 68	33.353115 ms
Game 69	15.443378 ms
Game 70	0.909655 ms
Game 71	23.199077 ms
Game 72	0.44719 ms
Game 73	2.605452 ms
Game 74	21.374703 ms
Game 75	82.953938 ms
Game 76	185.460826 ms
Game 77	36.707668 ms
Game 78	142.722988 ms
Game 79	313.657386 ms
Game 80	1.210273 ms
Game 81	74.671357 ms
Game 82	0.706624 ms
Game 83	8.991237 ms
Game 84	73.511991 ms
Game 85	7.586758 ms
Game 86	2.602492 ms
Game 87	7.513792 ms
Game 88	3.337442 ms
Game 89	101.685549 ms
Game 90	259.567841 ms
Game 91	51.463351 ms
Game 92	187.563902 ms
Game 93	49.047239 ms
Game 94	38.267368 ms
Game 95	92.163047 ms
Game 96	148.701171 ms
Game 97	57.192324 ms
Game 98	14.124868 ms
Game 99	0.593981 ms

	Fixed-point product based
Game 50	20.566329 ms
Game 51	6744.284861 ms
Game 52	1.47327 ms
Game 53	4.295792 ms
Game 54	29.14741 ms
Game 55	14.425234 ms
Game 56	912.128577 ms
Game 57	199.442715 ms
Game 58	6.650932 ms
Game 59	30.331524 ms
Game 60	266.762206 ms
Game 61	2.756214 ms
Game 62	215.868882 ms
Game 63	226.614402 ms
Game 64	83.090986 ms
Game 65	266.050094 ms
Game 66	121.173001 ms
Game 67	30.477553 ms
Game 68	43.78565 ms
Game 69	28.032054 ms
Game 70	0.396067 ms
Game 71	28.951018 ms
Game 72	0.200488 ms
Game 73	10.418019 ms
Game 74	143.527764 ms
Game 75	46.377756 ms
Game 76	173.248219 ms
Game 77	1023.572304 ms
Game 78	1500.871069 ms
Game 79	473.520308 ms
Game 80	1.31479 ms
Game 81	1054.141901 ms
Game 82	1.317754 ms
Game 83	24.022014 ms
Game 84	947.266091 ms
Game 85	9.411088 ms
Game 86	17.27655 ms
Game 87	5.299216 ms
Game 88	5.561753 ms
Game 89	1103.733905 ms
Game 90	642.737385 ms
Game 91	210.113964 ms
Game 92	140.735555 ms
Game 93	124.311004 ms
Game 94	465.467471 ms
Game 95	1940.174842 ms
Game 96	1066.142203 ms
Game 97	202.031576 ms
Game 98	24.381076 ms
Game 99	0.153665 ms

	Fixed-point local product based
Game 50	20.595845 ms
Game 51	6292.631423 ms
Game 52	1.398527 ms
Game 53	4.302088 ms
Game 54	29.24289 ms
Game 55	7.193027 ms
Game 56	472.727188 ms
Game 57	169.646163 ms
Game 58	6.664684 ms
Game 59	10.358405 ms
Game 60	267.307192 ms
Game 61	1.508336 ms
Game 62	205.976637 ms
Game 63	130.156094 ms
Game 64	29.198676 ms
Game 65	197.182221 ms
Game 66	120.811351 ms
Game 67	20.417376 ms
Game 68	18.646781 ms
Game 69	28.136298 ms
Game 70	0.381035 ms
Game 71	27.067918 ms
Game 72	0.204019 ms
Game 73	3.462291 ms
Game 74	143.903704 ms
Game 75	36.805055 ms
Game 76	167.128798 ms
Game 77	383.250688 ms
Game 78	564.751313 ms
Game 79	473.792807 ms
Game 80	1.086947 ms
Game 81	1053.657412 ms
Game 82	1.324338 ms
Game 83	14.837088 ms
Game 84	357.204857 ms
Game 85	9.476771 ms
Game 86	7.262061 ms
Game 87	5.352538 ms
Game 88	2.610671 ms
Game 89	543.43552 ms
Game 90	367.617943 ms
Game 91	87.154413 ms
Game 92	142.924215 ms
Game 93	74.032315 ms
Game 94	465.642971 ms
Game 95	1939.324955 ms
Game 96	1068.923695 ms
Game 97	202.097029 ms
Game 98	24.263109 ms
Game 99	0.154271 ms

	Zlnk fam based - explicit
Game 50	134.115221 ms
Game 51	9343.895693 ms
Game 52	4.751699 ms
Game 53	23.414928 ms
Game 54	596.248278 ms
Game 55	69.013627 ms
Game 56	2601.566566 ms
Game 57	81.407788 ms
Game 58	41.16784 ms
Game 59	48.588163 ms
Game 60	86.673604 ms
Game 61	44.042146 ms
Game 62	276.921593 ms
Game 63	432.521275 ms
Game 64	25.353502 ms
Game 65	2592.790921 ms
Game 66	29.396552 ms
Game 67	46.417311 ms
Game 68	160.569425 ms
Game 69	43.433305 ms
Game 70	0.694694 ms
Game 71	66.114718 ms
Game 72	0.387431 ms
Game 73	10.769351 ms
Game 74	28.102874 ms
Game 75	9.883799 ms
Game 76	2326.323546 ms
Game 77	143.74654 ms
Game 78	1473.697901 ms
Game 79	829.334874 ms
Game 80	10.683794 ms
Game 81	265.53113 ms
Game 82	6.763731 ms
Game 83	68.374198 ms
Game 84	373.221211 ms
Game 85	4.104353 ms
Game 86	21.095341 ms
Game 87	5.654448 ms
Game 88	5.123875 ms
Game 89	252.185215 ms
Game 90	1906.86293 ms
Game 91	29.10855 ms
Game 92	258.287784 ms
Game 93	96.508631 ms
Game 94	250.738963 ms
Game 95	43.95263 ms
Game 96	395.296555 ms
Game 97	62.937372 ms
Game 98	40.836075 ms
Game 99	0.235165 ms

	Zlnk fam based - symbolic
Game 50	30.008469 ms
Game 51	1999.800971 ms
Game 52	1.808211 ms
Game 53	8.461786 ms
Game 54	40.608858 ms
Game 55	50.971201 ms
Game 56	75.923961 ms
Game 57	29.997034 ms
Game 58	29.874202 ms
Game 59	13.7632 ms
Game 60	6.277646 ms
Game 61	30.812104 ms
Game 62	28.648873 ms
Game 63	124.170511 ms
Game 64	6.285064 ms
Game 65	119.129832 ms
Game 66	10.465418 ms
Game 67	32.170605 ms
Game 68	8.085773 ms
Game 69	4.788345 ms
Game 70	0.209629 ms
Game 71	7.7442 ms
Game 72	0.16619 ms
Game 73	4.405239 ms
Game 74	2.530264 ms
Game 75	0.229024 ms
Game 76	50.011567 ms
Game 77	12.211849 ms
Game 78	61.78517 ms
Game 79	9.310871 ms
Game 80	3.41224 ms
Game 81	11.595323 ms
Game 82	1.704003 ms
Game 83	17.046606 ms
Game 84	15.492898 ms
Game 85	0.451486 ms
Game 86	11.369413 ms
Game 87	0.657659 ms
Game 88	0.662482 ms
Game 89	10.069447 ms
Game 90	38.409335 ms
Game 91	1.355012 ms
Game 92	2.680355 ms
Game 93	3.834215 ms
Game 94	35.697677 ms
Game 95	0.972757 ms
Game 96	7.797964 ms
Game 97	1.282742 ms
Game 98	5.126716 ms
Game 99	0.031976 ms

	Incremental pre-solve
Game 50	46.080898 ms
Game 51	2992.042268 ms
Game 52	3.151228 ms
Game 53	9.583558 ms
Game 54	325.711561 ms
Game 55	33.458447 ms
Game 56	1539.005235 ms
Game 57	143.532033 ms
Game 58	21.824657 ms
Game 59	54.624111 ms
Game 60	122.370786 ms
Game 61	23.554931 ms
Game 62	109.501093 ms
Game 63	203.673008 ms
Game 64	35.808988 ms
Game 65	1165.840461 ms
Game 66	71.074647 ms
Game 67	27.291058 ms
Game 68	212.068661 ms
Game 69	80.825415 ms
Game 70	3.859968 ms
Game 71	114.442028 ms
Game 72	0.649724 ms
Game 73	18.543163 ms
Game 74	55.220584 ms
Game 75	48.855116 ms
Game 76	1419.192729 ms
Game 77	253.275833 ms
Game 78	1142.785118 ms
Game 79	1962.348341 ms
Game 80	6.247378 ms
Game 81	524.071394 ms
Game 82	2.152315 ms
Game 83	71.00827 ms
Game 84	206.907479 ms
Game 85	20.351748 ms
Game 86	18.925971 ms
Game 87	26.118424 ms
Game 88	8.694553 ms
Game 89	143.203366 ms
Game 90	192.609233 ms
Game 91	143.36905 ms
Game 92	11.604601 ms
Game 93	0.505494 ms
Game 94	4.989098 ms
Game 95	2.18487 ms
Game 96	1.447915 ms
Game 97	1.369827 ms
Game 98	86.552119 ms
Game 99	0.02177 ms

	Incremental pre-solve local
Game 50	9.018887 ms
Game 51	1523.362457 ms
Game 52	2.56276 ms
Game 53	6.587165 ms
Game 54	14.87686 ms
Game 55	9.903866 ms
Game 56	4.660675 ms
Game 57	96.08021 ms
Game 58	8.569419 ms
Game 59	0.534212 ms
Game 60	52.722595 ms
Game 61	1.132779 ms
Game 62	88.117407 ms
Game 63	8.078743 ms
Game 64	1.370403 ms
Game 65	2.78775 ms
Game 66	57.445895 ms
Game 67	3.053301 ms
Game 68	0.384612 ms
Game 69	0.376439 ms
Game 70	3.533979 ms
Game 71	25.211715 ms
Game 72	0.076533 ms
Game 73	7.696058 ms
Game 74	0.464487 ms
Game 75	15.709783 ms
Game 76	1.285309 ms
Game 77	5.801185 ms
Game 78	6.765665 ms
Game 79	0.722754 ms
Game 80	0.953798 ms
Game 81	3.425409 ms
Game 82	0.217681 ms
Game 83	1.021352 ms
Game 84	2.749428 ms
Game 85	0.121502 ms
Game 86	1.616307 ms
Game 87	0.078561 ms
Game 88	0.224175 ms
Game 89	2.50544 ms
Game 90	1.621967 ms
Game 91	0.683825 ms
Game 92	0.294685 ms
Game 93	0.503414 ms
Game 94	4.977672 ms
Game 95	2.243018 ms
Game 96	1.475905 ms
Game 97	0.312385 ms
Game 98	0.45833 ms
Game 99	0.020862 ms

A.4 Random games type 2

	Zlnk product based
Game 50	10.918245 ms
Game 51	173.983561 ms
Game 52	1.38404 ms
Game 53	1.563996 ms
Game 54	37.693475 ms
Game 55	2.884703 ms
Game 56	186.982175 ms
Game 57	7.849308 ms
Game 58	2.402196 ms
Game 59	9.357524 ms
Game 60	40.283558 ms
Game 61	2.453017 ms
Game 62	37.682566 ms
Game 63	18.900662 ms
Game 64	7.005823 ms
Game 65	151.08974 ms
Game 66	10.608501 ms
Game 67	2.668589 ms
Game 68	33.591869 ms
Game 69	16.203081 ms
Game 70	0.951235 ms
Game 71	23.205072 ms
Game 72	0.578534 ms
Game 73	2.070027 ms
Game 74	22.614131 ms
Game 75	47.03979 ms
Game 76	181.067279 ms
Game 77	39.142096 ms
Game 78	155.973436 ms
Game 79	296.588672 ms
Game 80	1.168711 ms
Game 81	73.937408 ms
Game 82	0.708429 ms
Game 83	8.857286 ms
Game 84	82.055301 ms
Game 85	9.470796 ms
Game 86	2.697405 ms
Game 87	7.927669 ms
Game 88	3.402374 ms
Game 89	93.767298 ms
Game 90	247.483212 ms
Game 91	51.416592 ms
Game 92	194.94486 ms
Game 93	50.602247 ms
Game 94	40.277615 ms
Game 95	93.780179 ms
Game 96	148.719081 ms
Game 97	61.45583 ms
Game 98	14.088651 ms
Game 99	0.614478 ms

	Fixed-point product based
Game 50	25.788145 ms
Game 51	6841.202002 ms
Game 52	1.142889 ms
Game 53	3.264741 ms
Game 54	30.670339 ms
Game 55	10.21778 ms
Game 56	1312.727673 ms
Game 57	336.312221 ms
Game 58	3.993694 ms
Game 59	19.763114 ms
Game 60	419.528973 ms
Game 61	1.847166 ms
Game 62	157.99114 ms
Game 63	123.032105 ms
Game 64	50.739421 ms
Game 65	339.604908 ms
Game 66	55.454246 ms
Game 67	56.577501 ms
Game 68	35.266073 ms
Game 69	31.136878 ms
Game 70	0.623003 ms
Game 71	28.043222 ms
Game 72	0.720565 ms
Game 73	14.39249 ms
Game 74	233.078601 ms
Game 75	20.684427 ms
Game 76	156.408315 ms
Game 77	411.702241 ms
Game 78	798.928246 ms
Game 79	463.166073 ms
Game 80	1.714026 ms
Game 81	735.675399 ms
Game 82	1.853504 ms
Game 83	14.961404 ms
Game 84	462.353075 ms
Game 85	18.109145 ms
Game 86	10.22887 ms
Game 87	4.766618 ms
Game 88	4.285175 ms
Game 89	569.014841 ms
Game 90	1137.612069 ms
Game 91	380.726518 ms
Game 92	159.622577 ms
Game 93	190.273961 ms
Game 94	595.608117 ms
Game 95	3054.323181 ms
Game 96	1172.194676 ms
Game 97	120.681324 ms
Game 98	27.819839 ms
Game 99	0.184036 ms

	Fixed-point local product based
Game 50	23.491823 ms
Game 51	6358.738676 ms
Game 52	1.120596 ms
Game 53	1.841316 ms
Game 54	16.842579 ms
Game 55	5.598839 ms
Game 56	1316.381878 ms
Game 57	323.38378 ms
Game 58	2.154177 ms
Game 59	8.593163 ms
Game 60	266.971185 ms
Game 61	0.974743 ms
Game 62	153.270569 ms
Game 63	123.573552 ms
Game 64	50.76468 ms
Game 65	151.33958 ms
Game 66	43.175666 ms
Game 67	56.524643 ms
Game 68	35.68747 ms
Game 69	27.232735 ms
Game 70	0.42265 ms
Game 71	28.190491 ms
Game 72	0.384138 ms
Game 73	12.369724 ms
Game 74	86.760695 ms
Game 75	20.558411 ms
Game 76	71.018835 ms
Game 77	412.161454 ms
Game 78	800.264776 ms
Game 79	462.523326 ms
Game 80	0.923882 ms
Game 81	735.152819 ms
Game 82	1.870616 ms
Game 83	9.113067 ms
Game 84	295.219217 ms
Game 85	11.739621 ms
Game 86	10.344082 ms
Game 87	4.326818 ms
Game 88	2.003342 ms
Game 89	569.378285 ms
Game 90	1133.058396 ms
Game 91	225.493779 ms
Game 92	160.648632 ms
Game 93	107.219647 ms
Game 94	596.556856 ms
Game 95	3053.310575 ms
Game 96	1175.237651 ms
Game 97	76.270106 ms
Game 98	27.717006 ms
Game 99	0.168606 ms

	Zlnk fam based - explicit
Game 50	420.892527 ms
Game 51	27572.326974 ms
Game 52	8.493117 ms
Game 53	20.603836 ms
Game 54	1820.06217 ms
Game 55	116.424002 ms
Game 56	6818.05413 ms
Game 57	152.332163 ms
Game 58	72.735493 ms
Game 59	110.421616 ms
Game 60	1293.173118 ms
Game 61	68.153455 ms
Game 62	554.753817 ms
Game 63	762.954768 ms
Game 64	53.042922 ms
Game 65	5594.930519 ms
Game 66	36.767411 ms
Game 67	51.269817 ms
Game 68	192.865339 ms
Game 69	87.789411 ms
Game 70	2.835724 ms
Game 71	207.624583 ms
Game 72	1.446822 ms
Game 73	17.618806 ms
Game 74	67.377396 ms
Game 75	5.127006 ms
Game 76	2816.136694 ms
Game 77	220.696102 ms
Game 78	2085.753805 ms
Game 79	1189.430678 ms
Game 80	7.765884 ms
Game 81	403.953604 ms
Game 82	3.16458 ms
Game 83	68.881758 ms
Game 84	552.718294 ms
Game 85	13.66698 ms
Game 86	26.188807 ms
Game 87	7.311722 ms
Game 88	6.040884 ms
Game 89	316.303413 ms
Game 90	2319.834539 ms
Game 91	35.78069 ms
Game 92	300.939324 ms
Game 93	112.089537 ms
Game 94	314.73919 ms
Game 95	51.41534 ms
Game 96	392.095848 ms
Game 97	77.036153 ms
Game 98	38.558349 ms
Game 99	0.312622 ms

	Zlnk fam based - symbolic
Game 50	202.77158 ms
Game 51	85292.257449 ms
Game 52	4.367472 ms
Game 53	10.823653 ms
Game 54	3445.363545 ms
Game 55	81.243074 ms
Game 56	18201.386708 ms
Game 57	105.554749 ms
Game 58	47.737789 ms
Game 59	65.431177 ms
Game 60	1073.714993 ms
Game 61	44.252954 ms
Game 62	373.83758 ms
Game 63	575.551737 ms
Game 64	24.641367 ms
Game 65	9258.326986 ms
Game 66	29.829226 ms
Game 67	33.649774 ms
Game 68	88.905642 ms
Game 69	33.595287 ms
Game 70	0.912352 ms
Game 71	247.337206 ms
Game 72	1.017398 ms
Game 73	8.016097 ms
Game 74	27.786453 ms
Game 75	1.348222 ms
Game 76	3402.734031 ms
Game 77	90.493295 ms
Game 78	2341.770188 ms
Game 79	1382.17279 ms
Game 80	3.201824 ms
Game 81	230.685978 ms
Game 82	1.145025 ms
Game 83	18.733033 ms
Game 84	263.509541 ms
Game 85	4.604088 ms
Game 86	14.244522 ms
Game 87	2.588723 ms
Game 88	1.156411 ms
Game 89	82.005214 ms
Game 90	1172.033005 ms
Game 91	8.036287 ms
Game 92	58.713405 ms
Game 93	8.327577 ms
Game 94	46.196091 ms
Game 95	4.695648 ms
Game 96	16.346571 ms
Game 97	3.059106 ms
Game 98	5.570903 ms
Game 99	0.032517 ms

	Incremental pre-solve
Game 50	274.513552 ms
Game 51	61404.869851 ms
Game 52	5.041807 ms
Game 53	16.068485 ms
Game 54	8252.836326 ms
Game 55	69.251691 ms
Game 56	88278.874709 ms
Game 57	666.755859 ms
Game 58	38.229257 ms
Game 59	226.141247 ms
Game 60	2514.769282 ms
Game 61	38.780713 ms
Game 62	1176.272233 ms
Game 63	858.516357 ms
Game 64	125.34107 ms
Game 65	37422.726912 ms
Game 66	133.928952 ms
Game 67	47.395902 ms
Game 68	2787.521288 ms
Game 69	512.772417 ms
Game 70	5.953089 ms
Game 71	757.297336 ms
Game 72	1.893045 ms
Game 73	30.999393 ms
Game 74	452.229702 ms
Game 75	104.657565 ms
Game 76	55820.453259 ms
Game 77	1661.170935 ms
Game 78	20559.171143 ms
Game 79	74687.630715 ms
Game 80	9.092573 ms
Game 81	5052.554785 ms
Game 82	3.555746 ms
Game 83	2.529063 ms
Game 84	5286.51502 ms
Game 85	34.527788 ms
Game 86	24.111287 ms
Game 87	52.726666 ms
Game 88	20.382232 ms
Game 89	3639.537318 ms
Game 90	24874.107362 ms
Game 91	496.947157 ms
Game 92	13755.969387 ms
Game 93	785.924095 ms
Game 94	502.19285 ms
Game 95	656.998381 ms
Game 96	8.802605 ms
Game 97	1.757358 ms
Game 98	0.72427 ms
Game 99	0.020685 ms

	Incremental pre-solve local
Game 50	269.718821 ms
Game 51	60355.20469 ms
Game 52	4.278918 ms
Game 53	14.222552 ms
Game 54	7967.297797 ms
Game 55	57.062669 ms
Game 56	231.673885 ms
Game 57	638.551196 ms
Game 58	20.990244 ms
Game 59	185.289698 ms
Game 60	2338.294662 ms
Game 61	1.645522 ms
Game 62	1120.42437 ms
Game 63	697.99699 ms
Game 64	2.75111 ms
Game 65	149.314436 ms
Game 66	122.997515 ms
Game 67	11.312594 ms
Game 68	14.694391 ms
Game 69	461.680203 ms
Game 70	5.849632 ms
Game 71	4.622708 ms
Game 72	1.631972 ms
Game 73	29.434576 ms
Game 74	4.10057 ms
Game 75	81.1602 ms
Game 76	137.661231 ms
Game 77	15.629775 ms
Game 78	85.947227 ms
Game 79	101.015369 ms
Game 80	0.48114 ms
Game 81	25.734733 ms
Game 82	0.330751 ms
Game 83	2.53277 ms
Game 84	24.373237 ms
Game 85	29.590797 ms
Game 86	1.480927 ms
Game 87	0.514867 ms
Game 88	0.38692 ms
Game 89	18.561496 ms
Game 90	60.996587 ms
Game 91	3.65302 ms
Game 92	20.437045 ms
Game 93	4.298427 ms
Game 94	10.100405 ms
Game 95	6.47525 ms
Game 96	8.752235 ms
Game 97	1.736223 ms
Game 98	0.749523 ms
Game 99	0.020884 ms

A.5 Random games type 3

	Zlnk product based
Game 50	9.976095 ms
Game 51	168.106822 ms
Game 52	2.037131 ms
Game 53	1.455536 ms
Game 54	38.714198 ms
Game 55	2.791138 ms
Game 56	187.030532 ms
Game 57	7.542981 ms
Game 58	2.237842 ms
Game 59	9.146537 ms
Game 60	45.055167 ms
Game 61	2.406704 ms
Game 62	35.729423 ms
Game 63	17.61155 ms
Game 64	7.421523 ms
Game 65	146.089908 ms
Game 66	16.845018 ms
Game 67	2.781907 ms
Game 68	35.396676 ms
Game 69	16.607279 ms
Game 70	0.967567 ms
Game 71	23.912201 ms
Game 72	0.51349 ms
Game 73	2.033038 ms
Game 74	22.84673 ms
Game 75	62.986938 ms
Game 76	187.89891 ms
Game 77	38.757606 ms
Game 78	137.862444 ms
Game 79	297.547201 ms
Game 80	1.183585 ms
Game 81	73.797893 ms
Game 82	0.79472 ms
Game 83	8.998882 ms
Game 84	77.547555 ms
Game 85	7.718849 ms
Game 86	2.819101 ms
Game 87	7.591827 ms
Game 88	3.20876 ms
Game 89	98.268788 ms
Game 90	250.712267 ms
Game 91	49.413216 ms
Game 92	189.968957 ms
Game 93	50.349972 ms
Game 94	38.956918 ms
Game 95	103.698611 ms
Game 96	159.487239 ms
Game 97	59.231515 ms
Game 98	14.638905 ms
Game 99	0.576372 ms

	Fixed-point product based
Game 50	33.39416 ms
Game 51	5534.871778 ms
Game 52	1.26391 ms
Game 53	3.170443 ms
Game 54	26.331521 ms
Game 55	11.795348 ms
Game 56	1504.589399 ms
Game 57	191.552096 ms
Game 58	5.875631 ms
Game 59	22.604068 ms
Game 60	420.111746 ms
Game 61	2.346122 ms
Game 62	210.076795 ms
Game 63	268.083218 ms
Game 64	188.800481 ms
Game 65	246.453054 ms
Game 66	109.679181 ms
Game 67	24.159304 ms
Game 68	32.304285 ms
Game 69	35.825679 ms
Game 70	0.344868 ms
Game 71	21.781779 ms
Game 72	0.602684 ms
Game 73	7.749658 ms
Game 74	212.354638 ms
Game 75	25.740094 ms
Game 76	154.448659 ms
Game 77	756.884077 ms
Game 78	1174.479264 ms
Game 79	422.807031 ms
Game 80	1.094506 ms
Game 81	934.68797 ms
Game 82	2.723153 ms
Game 83	18.059568 ms
Game 84	414.709718 ms
Game 85	20.150961 ms
Game 86	20.67089 ms
Game 87	5.526844 ms
Game 88	4.486084 ms
Game 89	792.944337 ms
Game 90	755.171693 ms
Game 91	380.795967 ms
Game 92	162.856618 ms
Game 93	131.000885 ms
Game 94	402.117363 ms
Game 95	3293.720234 ms
Game 96	1700.935701 ms
Game 97	80.308982 ms
Game 98	40.934591 ms
Game 99	0.13178 ms

	Fixed-point local product based
Game 50	13.808627 ms
Game 51	5326.973496 ms
Game 52	1.276611 ms
Game 53	1.64824 ms
Game 54	12.136227 ms
Game 55	6.576087 ms
Game 56	944.411779 ms
Game 57	191.884138 ms
Game 58	5.840379 ms
Game 59	10.144003 ms
Game 60	410.714964 ms
Game 61	1.14639 ms
Game 62	205.434014 ms
Game 63	267.867202 ms
Game 64	77.285434 ms
Game 65	223.400633 ms
Game 66	106.379365 ms
Game 67	13.235959 ms
Game 68	32.293533 ms
Game 69	35.856809 ms
Game 70	0.342402 ms
Game 71	20.189885 ms
Game 72	0.587294 ms
Game 73	6.687538 ms
Game 74	209.573048 ms
Game 75	24.833401 ms
Game 76	154.408502 ms
Game 77	301.337573 ms
Game 78	1173.799697 ms
Game 79	220.204762 ms
Game 80	1.11143 ms
Game 81	414.742317 ms
Game 82	2.324309 ms
Game 83	10.075222 ms
Game 84	415.978428 ms
Game 85	15.497647 ms
Game 86	8.184581 ms
Game 87	4.216409 ms
Game 88	4.532044 ms
Game 89	793.009891 ms
Game 90	448.992638 ms
Game 91	381.016753 ms
Game 92	77.706188 ms
Game 93	131.040994 ms
Game 94	402.395126 ms
Game 95	3293.137131 ms
Game 96	1060.421164 ms
Game 97	80.576536 ms
Game 98	23.666594 ms
Game 99	0.130765 ms

	Zlnk fam based - explicit
Game 50	273.792074 ms
Game 51	13646.451841 ms
Game 52	7.25798 ms
Game 53	11.243911 ms
Game 54	1055.519768 ms
Game 55	106.227548 ms
Game 56	6484.58168 ms
Game 57	81.66421 ms
Game 58	63.750863 ms
Game 59	109.994729 ms
Game 60	1603.233246 ms
Game 61	62.352276 ms
Game 62	898.041632 ms
Game 63	798.875525 ms
Game 64	110.70652 ms
Game 65	6624.857714 ms
Game 66	128.211484 ms
Game 67	76.392448 ms
Game 68	381.266139 ms
Game 69	149.083001 ms
Game 70	1.31729 ms
Game 71	200.448827 ms
Game 72	1.191567 ms
Game 73	23.675776 ms
Game 74	133.06584 ms
Game 75	8.497955 ms
Game 76	5812.819166 ms
Game 77	418.636531 ms
Game 78	3857.542835 ms
Game 79	2842.369859 ms
Game 80	13.008812 ms
Game 81	823.914959 ms
Game 82	3.90781 ms
Game 83	114.703845 ms
Game 84	1032.435205 ms
Game 85	7.073889 ms
Game 86	35.204818 ms
Game 87	33.873518 ms
Game 88	7.400586 ms
Game 89	606.091825 ms
Game 90	4348.115576 ms
Game 91	113.76588 ms
Game 92	783.015408 ms
Game 93	226.238564 ms
Game 94	529.478261 ms
Game 95	178.521841 ms
Game 96	841.505137 ms
Game 97	123.269255 ms
Game 98	66.753929 ms
Game 99	0.234607 ms

	Zlnk fam based - symbolic
Game 50	136.556033 ms
Game 51	36507.220998 ms
Game 52	3.313925 ms
Game 53	6.821058 ms
Game 54	1742.951911 ms
Game 55	75.711339 ms
Game 56	15763.37959 ms
Game 57	57.252336 ms
Game 58	44.878245 ms
Game 59	69.754642 ms
Game 60	1792.61042 ms
Game 61	43.232763 ms
Game 62	889.723099 ms
Game 63	691.620014 ms
Game 64	59.568202 ms
Game 65	12383.166294 ms
Game 66	96.249389 ms
Game 67	53.606178 ms
Game 68	386.686147 ms
Game 69	86.865845 ms
Game 70	0.329505 ms
Game 71	277.223349 ms
Game 72	0.992797 ms
Game 73	14.837991 ms
Game 74	76.712306 ms
Game 75	9.734143 ms
Game 76	11627.779415 ms
Game 77	414.721386 ms
Game 78	6413.826884 ms
Game 79	5931.527685 ms
Game 80	5.962704 ms
Game 81	1046.228509 ms
Game 82	3.105238 ms
Game 83	47.268526 ms
Game 84	1192.05896 ms
Game 85	3.327481 ms
Game 86	24.712157 ms
Game 87	18.135895 ms
Game 88	4.466311 ms
Game 89	579.281233 ms
Game 90	5432.643684 ms
Game 91	78.93278 ms
Game 92	744.988497 ms
Game 93	73.713132 ms
Game 94	232.290929 ms
Game 95	98.273438 ms
Game 96	429.448153 ms
Game 97	14.449629 ms
Game 98	15.714123 ms
Game 99	0.031156 ms

	Incremental pre-solve
Game 50	223.466021 ms
Game 51	47937.178942 ms
Game 52	4.769585 ms
Game 53	14.069063 ms
Game 54	6622.38747 ms
Game 55	60.795483 ms
Game 56	75834.067772 ms
Game 57	497.534014 ms
Game 58	37.426839 ms
Game 59	225.083612 ms
Game 60	2836.00766 ms
Game 61	39.508389 ms
Game 62	1499.171081 ms
Game 63	1077.431545 ms
Game 64	256.577525 ms
Game 65	42208.617471 ms
Game 66	208.42148 ms
Game 67	164.479026 ms
Game 68	4443.06214 ms
Game 69	742.156512 ms
Game 70	5.855942 ms
Game 71	1155.989937 ms
Game 72	1.819665 ms
Game 73	26.794277 ms
Game 74	996.17373 ms
Game 75	167.774753 ms
Game 76	99258.828158 ms
Game 77	3530.63729 ms
Game 78	34238.537422 ms
Game 79	143423.447357 ms
Game 80	12.132405 ms
Game 81	10427.95421 ms
Game 82	6.241549 ms
Game 83	255.292567 ms
Game 84	11249.112673 ms
Game 85	57.947425 ms
Game 86	36.634892 ms
Game 87	129.742853 ms
Game 88	35.592823 ms
Game 89	9632.501725 ms
Game 90	67689.612809 ms
Game 91	1946.877179 ms
Game 92	43094.739497 ms
Game 93	2865.112811 ms
Game 94	1356.038341 ms
Game 95	4543.837972 ms
Game 96	13465.080253 ms
Game 97	1906.935486 ms
Game 98	176.16669 ms
Game 99	0.020168 ms

	Incremental pre-solve local
Game 50	182.975723 ms
Game 51	44671.815935 ms
Game 52	4.637662 ms
Game 53	12.782434 ms
Game 54	4032.451405 ms
Game 55	31.408177 ms
Game 56	56605.457611 ms
Game 57	404.737995 ms
Game 58	19.957296 ms
Game 59	180.302945 ms
Game 60	2498.227687 ms
Game 61	15.375766 ms
Game 62	1364.469331 ms
Game 63	664.211163 ms
Game 64	205.724963 ms
Game 65	34218.480539 ms
Game 66	199.78587 ms
Game 67	136.205044 ms
Game 68	1704.763075 ms
Game 69	345.406447 ms
Game 70	1.766606 ms
Game 71	1111.678948 ms
Game 72	1.462348 ms
Game 73	26.320828 ms
Game 74	826.088687 ms
Game 75	129.308793 ms
Game 76	10589.753997 ms
Game 77	1986.122688 ms
Game 78	2772.260825 ms
Game 79	34775.330723 ms
Game 80	7.359347 ms
Game 81	3749.058872 ms
Game 82	6.064545 ms
Game 83	38.694408 ms
Game 84	974.61078 ms
Game 85	35.633124 ms
Game 86	10.211757 ms
Game 87	115.535395 ms
Game 88	13.844376 ms
Game 89	2323.685826 ms
Game 90	2379.568429 ms
Game 91	447.783042 ms
Game 92	5649.308915 ms
Game 93	268.755977 ms
Game 94	192.249596 ms
Game 95	1006.087455 ms
Game 96	993.745517 ms
Game 97	301.626611 ms
Game 98	6.109814 ms
Game 99	0.019839 ms

A.6 Random scale games

	Zlnk product based
Game 2a	0.218619 ms
Game 2b	0.254285 ms
Game 2c	0.685084 ms
Game 2d	0.202601 ms
Game 3c	2.169628 ms
Game 3d	0.641398 ms
Game 3	1.642876 ms
Game 4a	1.382486 ms
Game 4b	0.761629 ms
Game 4c	0.387448 ms
Game 4d	1.558909 ms
Game 4	0.471874 ms
Game 5a	2.155584 ms
Game 5b	7.940156 ms
Game 5c	1.077075 ms
Game 5d	12.452995 ms
Game 6a	4.941249 ms
Game 6b	7.448886 ms
Game 6c	8.854734 ms
Game 6d	3.303204 ms
Game 7a	15.773086 ms
Game 7b	8.192256 ms
Game 7c	4.651644 ms
Game 7d	4.684234 ms
Game 8a	42.609582 ms
Game 8b	17.150868 ms
Game 8c	13.403176 ms
Game 8d	34.365665 ms
Game 9a	170.513824 ms
Game 9b	24.262356 ms
Game 9c	18.42389 ms
Game 9d	66.014055 ms
Game 10a	85.933823 ms
Game 10b	46.613012 ms
Game 10c	248.446715 ms
Game 10d	115.753201 ms
Game 11a	151.191006 ms
Game 11b	337.831038 ms
Game 11c	74.969157 ms
Game 11d	96.280452 ms
Game 12a	937.437731 ms
Game 12b	1191.637453 ms
Game 12c	110.383665 ms
Game 12d	870.642396 ms

	Fixed-point product based
Game 2a	0.289869 ms
Game 2b	0.823527 ms
Game 2c	3.852203 ms
Game 2d	4.197048 ms
Game 3c	37.01689 ms
Game 3d	16.07463 ms
Game 3	2.356911 ms
Game 4a	2.746074 ms
Game 4b	0.547347 ms
Game 4c	0.126752 ms
Game 4d	5.695754 ms
Game 4	0.462942 ms
Game 5a	1.702403 ms
Game 5b	7.33214 ms
Game 5c	0.51776 ms
Game 5d	20.973557 ms
Game 6a	29.071679 ms
Game 6b	12.778665 ms
Game 6c	196.926396 ms
Game 6d	40.164514 ms
Game 7a	103.270116 ms
Game 7b	18.017859 ms
Game 7c	3.16192 ms
Game 7d	12.278696 ms
Game 8a	45.395534 ms
Game 8b	67.554086 ms
Game 8c	96.495009 ms
Game 8d	54.088157 ms
Game 9a	215.733053 ms
Game 9b	92.432614 ms
Game 9c	26.96885 ms
Game 9d	115.984155 ms
Game 10a	54.51221 ms
Game 10b	61.192662 ms
Game 10c	1047.345171 ms
Game 10d	2488.498826 ms
Game 11a	899.060691 ms
Game 11b	1336.42121 ms
Game 11c	104.520117 ms
Game 11d	615.371781 ms
Game 12a	922.246974 ms
Game 12b	8159.747723 ms
Game 12c	132.80073 ms
Game 12d	906.988802 ms

	Fixed-point local product based
Game 2a	0.275598 ms
Game 2b	0.493619 ms
Game 2c	3.795691 ms
Game 2d	4.231017 ms
Game 3c	36.927433 ms
Game 3d	16.312832 ms
Game 3	2.346755 ms
Game 4a	2.793239 ms
Game 4b	0.243132 ms
Game 4c	0.125161 ms
Game 4d	5.710197 ms
Game 4	0.453259 ms
Game 5a	1.713909 ms
Game 5b	3.857951 ms
Game 5c	0.52551 ms
Game 5d	11.633533 ms
Game 6a	5.333565 ms
Game 6b	12.767724 ms
Game 6c	198.298965 ms
Game 6d	37.896252 ms
Game 7a	66.951652 ms
Game 7b	18.030418 ms
Game 7c	3.183493 ms
Game 7d	12.293851 ms
Game 8a	44.225411 ms
Game 8b	22.873975 ms
Game 8c	49.403749 ms
Game 8d	54.40273 ms
Game 9a	130.934114 ms
Game 9b	92.897594 ms
Game 9c	18.349991 ms
Game 9d	64.62795 ms
Game 10a	28.647424 ms
Game 10b	61.268989 ms
Game 10c	1049.365977 ms
Game 10d	2165.865324 ms
Game 11a	898.492652 ms
Game 11b	797.898309 ms
Game 11c	85.833675 ms
Game 11d	614.757797 ms
Game 12a	927.441418 ms
Game 12b	8156.109629 ms
Game 12c	132.008086 ms
Game 12d	913.086655 ms

	Zlnk fam based - explicit
Game 2a	1.573788 ms
Game 2b	2.325468 ms
Game 2c	8.002301 ms
Game 2d	0.493537 ms
Game 3c	36.545318 ms
Game 3d	1.472726 ms
Game 3	29.143848 ms
Game 4a	6.431603 ms
Game 4b	2.338128 ms
Game 4c	0.225128 ms
Game 4d	6.151446 ms
Game 4	1.695825 ms
Game 5a	7.05817 ms
Game 5b	84.048738 ms
Game 5c	1.787185 ms
Game 5d	251.057677 ms
Game 6a	7.081844 ms
Game 6b	32.570335 ms
Game 6c	7.962399 ms
Game 6d	3.237656 ms
Game 7a	19.884909 ms
Game 7b	11.873086 ms
Game 7c	2.195627 ms
Game 7d	6.091123 ms
Game 8a	243.813714 ms
Game 8b	17.348234 ms
Game 8c	8.437889 ms
Game 8d	133.539403 ms
Game 9a	1344.320405 ms
Game 9b	19.089173 ms
Game 9c	7.582653 ms
Game 9d	207.958891 ms
Game 10a	204.418578 ms
Game 10b	10.469885 ms
Game 10c	1253.036232 ms
Game 10d	78.559265 ms
Game 11a	76.917888 ms
Game 11b	1055.452837 ms
Game 11c	57.274867 ms
Game 11d	87.216235 ms
Game 12a	3368.892219 ms
Game 12b	6525.556889 ms
Game 12c	32.864634 ms
Game 12d	2224.366375 ms

	Zlnk fam based - symbolic
Game 2a	1.041346 ms
Game 2b	1.543646 ms
Game 2c	6.373157 ms
Game 2d	0.325945 ms
Game 3c	35.437264 ms
Game 3d	1.125151 ms
Game 3	27.267114 ms
Game 4a	2.663953 ms
Game 4b	0.688344 ms
Game 4c	0.046223 ms
Game 4d	2.337379 ms
Game 4	1.430764 ms
Game 5a	1.450956 ms
Game 5b	28.215689 ms
Game 5c	0.576556 ms
Game 5d	83.525374 ms
Game 6a	1.562035 ms
Game 6b	7.174414 ms
Game 6c	1.408088 ms
Game 6d	0.621669 ms
Game 7a	2.678508 ms
Game 7b	1.082131 ms
Game 7c	0.223138 ms
Game 7d	0.60704 ms
Game 8a	17.830424 ms
Game 8b	0.987863 ms
Game 8c	0.63709 ms
Game 8d	9.918959 ms
Game 9a	51.749732 ms
Game 9b	0.79506 ms
Game 9c	0.286808 ms
Game 9d	8.256069 ms
Game 10a	4.034914 ms
Game 10b	0.248462 ms
Game 10c	24.639604 ms
Game 10d	8.788762 ms
Game 11a	0.962753 ms
Game 11b	10.355926 ms
Game 11c	0.654151 ms
Game 11d	1.07811 ms
Game 12a	17.111062 ms
Game 12b	32.849372 ms
Game 12c	0.346623 ms
Game 12d	11.343972 ms

	Incremental pre-solve
Game 2a	0.811764 ms
Game 2b	1.148929 ms
Game 2c	4.239638 ms
Game 2d	2.081608 ms
Game 3c	21.286546 ms
Game 3d	4.600061 ms
Game 3	8.164722 ms
Game 4a	2.394831 ms
Game 4b	2.44522 ms
Game 4c	0.028754 ms
Game 4d	5.364946 ms
Game 4	1.237886 ms
Game 5a	5.368428 ms
Game 5b	43.541702 ms
Game 5c	4.628053 ms
Game 5d	114.214773 ms
Game 6a	16.892639 ms
Game 6b	17.772473 ms
Game 6c	23.784703 ms
Game 6d	12.506237 ms
Game 7a	54.140063 ms
Game 7b	29.67486 ms
Game 7c	9.529289 ms
Game 7d	8.606841 ms
Game 8a	0.786969 ms
Game 8b	64.062096 ms
Game 8c	46.006536 ms
Game 8d	0.619124 ms
Game 9a	248.430049 ms
Game 9b	85.026739 ms
Game 9c	4.321948 ms
Game 9d	0.561722 ms
Game 10a	292.491686 ms
Game 10b	48.732983 ms
Game 10c	169.264165 ms
Game 10d	1042.297044 ms
Game 11a	536.153076 ms
Game 11b	1.147536 ms
Game 11c	49.02516 ms
Game 11d	139.684456 ms
Game 12a	1728.508003 ms
Game 12b	3.109087 ms
Game 12c	105.602598 ms
Game 12d	4076.344935 ms

	Incremental pre-solve local
Game 2a	0.184919 ms
Game 2b	0.353704 ms
Game 2c	1.550435 ms
Game 2d	1.58245 ms
Game 3c	6.207901 ms
Game 3d	2.427378 ms
Game 3	1.015665 ms
Game 4a	0.404769 ms
Game 4b	0.171154 ms
Game 4c	0.027441 ms
Game 4d	0.629948 ms
Game 4	0.169857 ms
Game 5a	0.199643 ms
Game 5b	1.068162 ms
Game 5c	0.090949 ms
Game 5d	2.029073 ms
Game 6a	0.843247 ms
Game 6b	0.547847 ms
Game 6c	3.744065 ms
Game 6d	9.462903 ms
Game 7a	1.300875 ms
Game 7b	0.291092 ms
Game 7c	0.077919 ms
Game 7d	5.607273 ms
Game 8a	0.797932 ms
Game 8b	0.463523 ms
Game 8c	0.640107 ms
Game 8d	0.62179 ms
Game 9a	1.517523 ms
Game 9b	0.304071 ms
Game 9c	0.103037 ms
Game 9d	0.566604 ms
Game 10a	0.284593 ms
Game 10b	0.142399 ms
Game 10c	1.76313 ms
Game 10d	918.460802 ms
Game 11a	0.733337 ms
Game 11b	1.165118 ms
Game 11c	0.122856 ms
Game 11d	0.424249 ms
Game 12a	0.948749 ms
Game 12b	3.106324 ms
Game 12c	104.64924 ms
Game 12d	0.809707 ms

References

[1] G. Birkhoff. *Lattice Theory*. Number v. 25,dl. 2 in American Mathematical Society colloquium publications. American Mathematical Society, 1940.

[2] J. Bradfield and I. Walukiewicz. *The mu-calculus and Model Checking*, pages 871–919. Springer International Publishing, Cham, 2018.

[3] F. Bruse, M. Falk, and M. Lange. The fixpoint-iteration algorithm for parity games. *Electronic Proceedings in Theoretical Computer Science*, 161, 08 2014.

[4] R. E. Bryant. *Binary Decision Diagrams*, pages 191–217. Springer International Publishing, Cham, 2018.

[5] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *IEEE Transactions on Software Engineering*, 39:1069–1089, 2013.

[6] E. A. Emerson and C. Lei. Model checking in the propositional mu-calculus. Technical report, Austin, TX, USA, 1986.

[7] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194 – 211, 1979.

[8] O. Friedmann and M. Lange. Solving parity games in practice. In Z. Liu and A. P. Ravn, editors, *Automated Technology for Verification and Analysis*, pages 182–196, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[9] O. Friedmann and M. Lange. Solving parity games in practice. In Z. Liu and A. P. Ravn, editors, *Automated Technology for Verification and Analysis*, pages 182–196, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[10] O. Friedmann and M. Lange. The pgsolver collection of parity game solvers version 3. 2010.

[11] J. F. Groote and M. R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[12] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. 1983.

[13] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149 – 184, 1993.

[14] M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53 – 84, 2001.

[15] L. Sanchez, J. Wesselink, and T. Willemse. *BDD-based parity game solving: a comparison of Zielonka's recursive algorithm, priority promotion and fixpoint iteration*. Computer science reports. Technische Universiteit Eindhoven, 2018.

[16] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249 – 264, 1989.

[17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[18] M. ter Beek, E. de Vink, and T. Willemse. Family-based model checking with mcrl2. In M. Huisman and J. Rubin, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 387–405, Germany, 2017. Springer.

[19] T. van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 291–308, Cham, 2018. Springer International Publishing.

[20] I. Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275(1):311 – 346, 2002.

[21] I. Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics, 2000.

[22] I. Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics, 2000.

[23] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1):135 – 183, 1998.