Eindhoven University of Technology
Department of Mathematics and Computer Science
Formal system analysis

Verifying SPLs using parity games expressing variability

Sjef van Loo

Eindhoven, November 2019

Msc Thesis
Computer Science and Engineering

Supervised by T.A.C. Willemse

Abstract

SPL verification can be costly when all the software products of an SPL are verified independently. It is well known that parity games can be used to verify software products. We propose a generalization of parity games, named variability parity games (VPGs), that encode multiple parity games in a single game graph decorated with edge labels expressing variability between the parity games. We show that a VPG can be constructed from a modal μ-calculus formula and an FTS that models the behaviour of the different software products of an SPL. Solving the resulting VPG decides for which products in the SPL the formula is satisfied. We introduce several algorithms to efficiently solve VPGs and exploit commonalities between the different parity games encoded. We perform experiments on SPL models to demonstrate that the VPG algorithms indeed outperform independently verifying every product in an SPL.

Table of 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 and these models can then be checked to verify that they behave conforming to some formally specified requirement. These verification techniques are well-studied, specifically techniques to verify a single software product.

Software product lines (SPLs) are systems that can be configured to result in different variants of the same system [12, 32]. SPLs describe *families* of software products where the products originate from the same system and often times have a lot of commonalities. The difference between the products in a family is called the *variability* of the family [40]. A family of products can be verified by using traditional verification techniques to verify every single product independently. However, verifying models is expensive in term of computing power and the number of products in an SPL can grow large, therefore having to verify every single product independently is undesirable [11].

A common way of modelling the behaviour of software is by using *labelled transition systems* (LTSs) [22]. While LTSs can model behaviour well they cannot model variability. Efforts to also model variability include modal transition systems [14, 15, 38], I/O automata [26, 24] and *featured transition systems* (FTSs) [11, 8]. 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. FTSs use *features* to express variability; a feature is an option that can be turned on or off for the system. In the context of FTSs, a set of features is synonymous with a software product; an FTS describes the behaviour of a software product by enabling and disabling parts of the system based on the which features are enabled.

There are numerous temporal logics that can be used to formally express requirements. Examples include LTL, CTL, CTL* and modal μ-calculus [31, 1, 22]. Of the different temporal logics, the modal μ-calculus is the most expressive one; it subsumes the other temporal logics [28].

In this thesis we aim to verify the software products of an SPL in a collective manner that exploits commonalities between the different products. Given an FTS M, describing the set of products P, and a modal μ-calculus formula, we explore methods to find the largest set of products $P_s \subseteq P$ such that all products in P_s satisfy the formula. Specifically, we aim to verify SPLs more efficiently than verifying the products independently.

Parity games can be used to determine if an LTS behaves according to a modal μ-calculus formula. Parity games are directed graphs that express a game played by two players [3]. Every vertex in the graph is won by exactly one of the players and a parity game is globally *solved* when it is determined for every vertex who the winner is. A parity game can be constructed from an LTS and a modal μ-calculus formula such that solving the parity game provides the information needed to determine if the LTS behaves according to the formula [3].

We introduce a generalization of parity games, called *variability parity games* (VPGs). A VPG expresses variability similar to how an FTS expresses variability. However, instead of using features a VPG expresses variability through *configurations.* Parity games have a winner for every vertex, VPGs have a winner for every vertex configuration combination. A VPG is globally solved when it is determined for every vertex configuration combination who the winner is. We introduce a way of constructing a VPG from an FTS and a modal μ-calculus formula such that solving the VPG provides the information needed to determine which products, described by the FTS, behave according to the formula.

We introduce several algorithms to solve VPGs. We also introduce algorithms to partially solve

VPGs, in which case we only determine the winner of the vertex configuration combinations that are needed to determine which products, described by the FTS, behave according to the formula. This technique is called *locally* solving a VPG. We can also locally solve a parity game, where we only determine the winner of the vertex that is needed to determine if the LTS behaves according to the formula. Besides introducing local variants of the novel VPG algorithms we also introduce local variants of two well known parity game algorithms, namely the recursive algorithm [45, 29] and the fixed-point iteration algorithm [43, 4].

Finally we implement the algorithms and compare their performances. We use two SPL models to create a number of VPGs. We compare the time it takes the algorithms to solve the VPGs with the time it takes to verify every product in the SPLs independently. For the independent verification approach we create parity games for all the products and requirements; these parity games are solved using existing parity game algorithms.

Through this experimental evaluation we show that we can indeed use a collective approach to more efficiently verify SPLs. The most efficient algorithm exploits commonalities between configuration by representing VPGs partially symbolic. This algorithm verifies the SPL properties 2 to 18 times quicker than an independent approach verifies them. Furthermore, we show that locally solving a VPG might improve the performance compared to globally solving a VPG; more so than locally solving a parity game improves the performance compared to globally solving a parity games.

Outline. First, we explore work related to model-checking SPLs in Chapter 2. Next, in Chapter 3, we introduce the following preliminary concepts: LTSs, μ-calculus, parity games, model-checking using parity games, two parity game algorithms and symbolically representing sets. In Chapter 4 we formally introduce FTSs and the problem statement. We introduce VPGs in Chapter 5 and show that they can be used to model-check FTSs. We introduce VPG solving algorithms in Chapter 6 and in Chapter 7 we present local variants. Finally, we discuss the implementation and experimental results in Chapter 8.

2. Related work

Prior work has been done to verify SPLs, we discuss four notable contributions.

In [11] a method is introduced to verify for which products in an FTS an LTL property holds. It does so by constructing a Büchi automaton representing the complement of the LTL property and checking if the synchronous product of the automaton and the transition system has an empty language [41]. When applying this method to verify LTSs the reachability of the synchronous product is explored. For FTSs the paper introduces a reachability definition that determines for every product if a state is reachable. It is observed that a symbolic representation of the sets of products is advantageous when keeping track of the reachability. The paper uses the minepump example [25] to perform an experimental evaluation and find a substantial gain using verifying a family of products collectively as opposed to independently.

This work is expanded upon in [8]. The performance of such an approach is further elaborated upon and it is confirmed that a collective approach indeed outperforms an independent approach. Furthermore an extension of LTL is presented, namely featured LTL (fLTL). fLTL parametrizes LTL to be able to express properties in terms of features. Using this language one can distinguish between products when expressing temporal requirements.

In [10] symbolic model-checking is used to verify SPLs. fCTL is introduced as an extension of CTL that is able to reason about features. Verification of a CTL property can be done by expressing the CTL property as a tree, its parse tree, and doing a bottom-up traversal of it, deciding at every node what states satisfy the subformula. The paper proposes a way to symbolically represents FTSs, introduces a parse tree definition for the fCTL language and introduces an algorithm to do a bottom-up traversal of a fCTL parse tree, deciding at every node which state-product pairs satisfy it. These concepts are put in practice using the symbolic model checking toolset NuSMV [7] as a basis and extend the language to express variability. The paper uses the elevator example [30], modified to have 9 features, to show that symbolically model checking an SPL collectively using the methods proposed can significantly improve the performance compared to symbolically model checking all products independently.

Finally, in [36] an extension of the modal μ-calculus, namely μL_f, is proposed that can reason about features. In [37] it is shown how properties expressed in μL_f can be embedded in first order μ-calculus and how the mCRL2 toolset [6] can be put to work to verify these properties. An algorithm is proposed that recursively partitions the products based on their features. After every partitioning the algorithm checks if the remaining set of products all satisfy the requirement or none of them satisfy the requirement. If either is true then that recursion is done, otherwise the algorithm continues. It is observed that the performance of this approach depends largely on deciding how to partition the sets of products. What would be a good heuristic/approach to splitting products is left unanswered in the paper.

3. Preliminaries

3.1 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.

3.1.1 Lattices

We introduce definitions for ordering and lattices taken from [2].

Definition 3.1 ([2])**.** *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 3.2 ([2])**.** *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 3.3 ([2])**.** *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 that every other upper bound is larger or equal to a.*

The term least upper bound is synonymous with the term supremum, we write $\sup\{S\}$ to denote the supremum of set S.

Definition 3.4 ([2])**.** *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, we write $\inf\{S\}$ to denote the infimum of set S.

Definition 3.5 ([2])**.** *A lattice is a partially ordered set where any two of its elements have a supremum and an infimum.*

Definition 3.6 ([2])**.** *A complete lattice is a partially ordered set in which every subset has a supremum and an infimum.*

Definition 3.7 ([2])**.** *Given a lattice $\langle D, \leq \rangle$, function $f : D \to D$ is monotonic if and only if for all $x \in D$ and $y \in D$ it holds that if $x \leq y$ then $f(x) \leq f(y)$.*

3.1.2 Fixed-points

Fixed-points are formally defined as follows:

Definition 3.8. *Given function $f : D \to D$ the value $x \in D$ is a fixed-point for f if and only if $f(x) = x$. Furthermore x is the least fixed-point for f if every other fixed-point for f is greater or equal to x and dually x is the greatest fixed-point for f if every other fixed-point 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.

Theorem 3.1 (Knaster-Tarski[35])**.** *Let*

- *$\langle A, \leq \rangle$ be a complete lattice,*
- *f be a monotonic function on A to A,*
- *P be the set of all fixed-points 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

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

3.2 Model verification

It is difficult to develop correct software, one way to improve reliability of software is through model verification; the behaviour of the 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 changes of the system. Transitions are labelled with actions that indicate what kind of change is happening. Formally we define an LTS as follows.

Definition 3.9 ([22])**.** *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 state indicates the initial state. Such a representation is depicted in the example below.

Example 3.1 ([37])**.** *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 3.1.

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

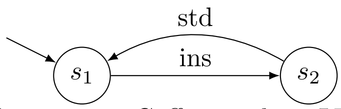


Figure 3.1: Coffee machine LTS

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 3.2. *We extend the coffee machine example such that at some point the coffee machine can be empty and needs to be filled before the system is ready to receive a coin again. This LTS is depicted in Figure 3.2. When the std transition is taken from state s_2 it is non-determined in which states the system ends up.*

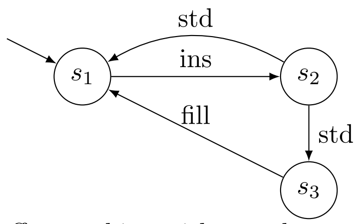


Figure 3.2: Coffee machine with non-deterministic behaviour

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, i.e. no negations.

Definition 3.10 ([22])**.** *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 fixed-point 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 $\eta[X := S']$, i.e. $(\eta[X := S'])(X) = S'$.

Definition 3.11 ([22])**.** *For LTS $(S, Act, trans, s_0)$ we inductively define the interpretation of a modal μ-calculus formula φ, notation $[\![\varphi]\!]^\eta$, where $\eta : \mathcal{X} \to 2^S$ 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, i.e. 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 3.1) 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 3.3 ([37])**.** *Consider the coffee machine example from Figure 3.1, which we call C, and formula $\varphi = \nu X.\mu Y([ins]Y \wedge [std]X)$ which states that action std must occur infinitely often over all infinite runs. Obviously this holds for the coffee machine, therefore we have $C \models \varphi$.*

3.3 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, i.e. $\overline{0} = 1$ 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 3.12 ([3]). *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 3.4. *Figure 3.3 shows the parity game:*

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

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

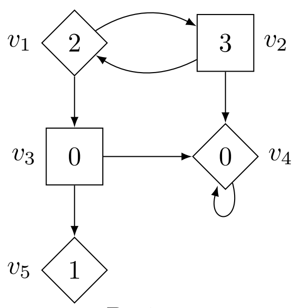


Figure 3.3: Parity game example

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, i.e. 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 occurring infinitely often 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 3.5. *Again consider the example in Figure 3.3. 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* $w_1 \ldots w_m$*.*

- $\pi = v_1v_3v_5$ *is won by player* 1 *since player* 0 *cannot move at* v_5*.*
- $\pi = (v_1v_2)^\omega$ *is won by player* 1 *since the highest priority occurring infinitely often is 3.*
- $\pi = v_1v_3(v_4)^\omega$ *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 3.6. *In the parity game seen in Figure 3.3 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 [3]. Meaning that every vertex in a parity game is winning for exactly 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 previously visited vertices. Therefore we can consider a strategy for player α as a function $\sigma_\alpha : V_\alpha \to V$. Finally, it is decidable for each of the vertices in a parity game who the winner is [3].

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. Solving parity games is in complexity class UP $\cap$ co-UP and NP $\cap$ co-NP [23]. No polynomial algorithms are known, however finding a polynomial algorithm does not prove P=NP.

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.

3.3.1 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 3.4.

$$\begin{tikzcd}
\text{LTS } M \arrow[dr] & & \text{Modal } \mu\text{-calculus formula } \varphi \arrow[dl] \\
& \text{PG} \arrow[d] & \\
& M \overset{?}{\models} \varphi &
\end{tikzcd}$$

Figure 3.4: LTS verification using parity games

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 fixed-point 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 fixed-point formula results in the same set as interpreting its unfolding as shown in [3]; i.e. $[\![\mu X.\varphi]\!]^\eta = [\![\varphi[X := \mu X.\varphi]]\!]^\eta$. The same holds for the fixed-point operator ν.

Next we define the Fischer-Ladner closure for a closed μ-calculus formula [34, 16]. 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 3.13. *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 3.14 ([3]). *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 fixed-points.*

Example 3.7. *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 [3], it holds that formula $\mu X.\psi$ has the same alternation depth as its unfolding $\psi[X := \mu X.\psi]$. Similarly for the greatest fixed-point.

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

Definition 3.15 ([3]). *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 represented 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 owners, successors and priorities:

Vertex	*Owner*	*Successor(s)*	*Priority*
$(s, \bot)$	*0*		*0*
$(s, \top)$	*1*		*0*
$(s, \psi_1 \vee \psi_2)$	*0*	(s, ψ_1) *and* (s, ψ_2)	*0*
$(s, \psi_1 \wedge \psi_2)$	*1*	(s, ψ_1) *and* (s, ψ_2)	*0*
$(s, \langle a\rangle\psi)$	*0*	(s', ψ) *for every* $s \xrightarrow{a} s'$	*0*
$(s, [a]\psi)$	*1*	(s', ψ) *for every* $s \xrightarrow{a} s'$	*0*
$(s, \mu X.\psi)$	*1*	$(s, \psi[X := \mu X.\psi])$	$2\lfloor adepth(X)/2\rfloor + 1$
$(s, \nu X.\psi)$	*1*	$(s, \psi[X := \nu X.\psi])$	$2\lfloor adepth(X)/2\rfloor$

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

Example 3.8. *Consider LTS M in Figure 3.5 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.*

The resulting parity game is depicted in Figure 3.6. Let V denote the set of vertices of this parity game. There are two vertices with more than one outgoing edge. From vertex $(s_1, [a](\mu X.\phi) \vee$

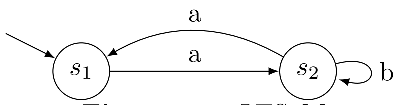


Figure 3.5: LTS M

$\langle b \rangle \top$) 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 will lose 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):

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

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.

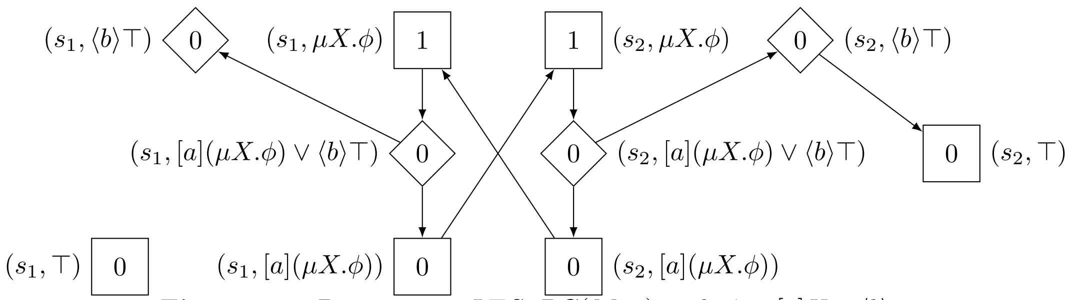


Figure 3.6: Parity game $LTS2PG(M, \varphi)$ with $\phi = [a]X \vee \langle b \rangle \top$

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 and proven in [3].

Theorem 3.2 ([3])**.** *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 $LTS2PG(M, \varphi)$.*

3.3.2 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.

3.3.3 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 [39, 18]. We also inspect the fixed-point iteration algorithm which tends to perform well for model-checking problems with a low number of distinct priorities [33].

Zielonka's recursive algorithm

First we consider Zielonka's recursive algorithm [45, 29], 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 [17].

Algorithm 1 RecursivePG(*parity game* $G = (V, V_0, V_1, E, \Omega)$)

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

The algorithm solves G by taking the set of vertices with the highest priority and choosing 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, 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 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 remainder of the game player α can keep the play from $W'_{\overline{\alpha}}$ so the algorithm solves the remainder of the game recursively to find the complete winning sets for game G.

A complete explanation of the algorithm can be found in [45], 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 3.16 ([45])**.** *Given parity game $G = (V, V_0, V_1, E, \Omega)$ and a non-empty set $U \subseteq V$ we inductively define α-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:

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

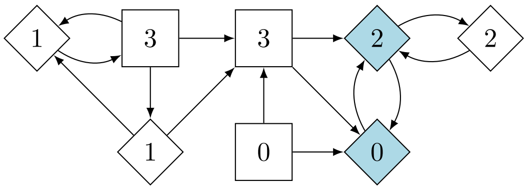
Example 3.9. *Figure 3.7 shows an example parity game in which an attractor set is calculated for player 0. For set U_2 no more vertices can be attracted so we found the complete attractor set.*

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

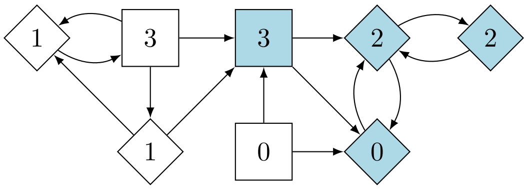
Definition 3.17 ([45])**.** *Given a parity game $G = (V, V_0, V_1, E, \Omega)$ and $U \subseteq V$ we define the subgame $G \backslash U$ to be the game $(V', V'_0, V'_1, E', \Omega)$ with:*

- $V' = V \backslash U$,
- $V'_0 = V_0 \cap V'$,
- $V'_1 = V_1 \cap V'$ *and*
- $E' = E \cap (V' \times V')$.

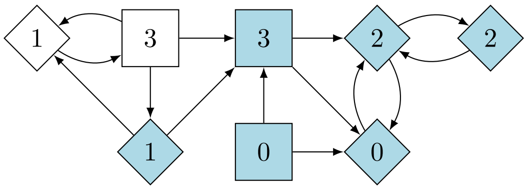
Note that a subgame is not necessarily total, however the recursive algorithm always creates subgames that are total [45].



(a) Set $U = U_0$ highlighted



(b) Set U_1 highlighted



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

Figure 3.7: Game G showing the attractor calculation for $0\text{-}Attr(G, U)$

Fixed-point iteration algorithm

Parity games can be solved by solving an alternating fixed-point formula [43]. Consider parity game $G = (V, V_0, V_1, E, \Omega)$ 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 $\{1, \ldots, d\}$ [19, 4]. 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

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

with

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

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 [43] formula $S(G)$ 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 [43].

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

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

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

$$\dots$$

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

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, i.e. if $S^i(Z_i) \geq S^i(Z_i')$ then $Z_i \geq Z_i'$.

Fixed-point formulas can be solved by *fixed-point iteration*. As shown in [13] 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 subformulas, 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 [4] 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

function FPITER($G = (V, V_0, V_1, E, \Omega)$)
  for $i \leftarrow d-1, \ldots, 0$ **do**
    INIT(i)
  end for
  repeat
    $Z'_0 \leftarrow Z_0$
    $Z_0 \leftarrow$ DIAMOND() $\cup$ BOX()
    $i \leftarrow 0$
    while $Z_i = Z'_i \wedge i < d-1$ **do**
      $i \leftarrow i+1$
      $Z'_i \leftarrow Z_i$
      $Z_i \leftarrow Z_{i-1}$
      INIT($i-1$)
    end while
  until $i = d-1 \wedge Z_{d-1} = Z'_{d-1}$
  return $(Z_{d-1}, V \backslash Z_{d-1})$
end function

function INIT(i)
  $Z_i \leftarrow \emptyset$ if i is odd, V otherwise
end function

function DIAMOND
  return $\{v \in V_0 \mid \exists_{w \in V}(v,w) \in E \wedge w \in Z_{\Omega(w)}\}$
end function

function BOX
  return $\{v \in V_1 \mid \forall_{w \in V}(v,w) \in E \implies w \in Z_{\Omega(w)}\}$
end function

3.4 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* [44, 5].

Example 3.10. *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, represented in binary format.

We can perform set operations on sets represented as boolean functions by performing logical operations on the functions. For example, given boolean formulas 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 formulas 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'.

3.4.1 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 [44, 5].

BDDs represent boolean formulas 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 it 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 3.11. *Consider the boolean formula in Example 3.10. This formula can be represented as the BDD shown in Figure 3.8. 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 that has no origin vertex.*

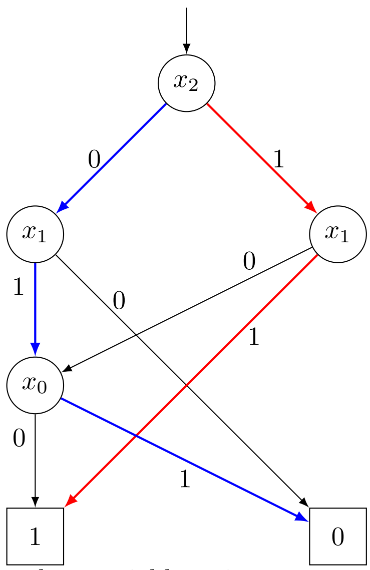


Figure 3.8: BDD highlighting boolean variable assignment $x_2x_1x_0 = 011$ in blue and $x_2x_1 = 11$ in red

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 3.10.*

Given n boolean variables and two boolean functions encoded as BDDs we can perform binary operations $\vee, \wedge$ on the BDDs in $O(N_a * N_b)$, where N_a and N_b are the number of nodes in the decision diagrams of the two functions. A decision diagram is a tree with n levels, so $N_a = O(2^n)$ and $N_b = O(2^n)$. Therefore with n boolean variables we can perform binary operations $\vee$ and $\wedge$ on them in $O(2^{2n}) = O(m^2)$ where $m = 2^n$ is the maximum set size that can be represented using n variables [44, 5]. 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.

4. 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 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 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 for example. To express the relation between features one can use feature diagrams as explained in [9]. 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 4.1 ([9])**.** *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 feature expression.*

A transition $s \xrightarrow{a} s'$ with $\gamma(s, a, s') = f$ 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 4.1 ([37])**.** *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 4.1.*

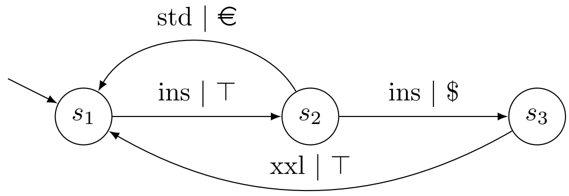
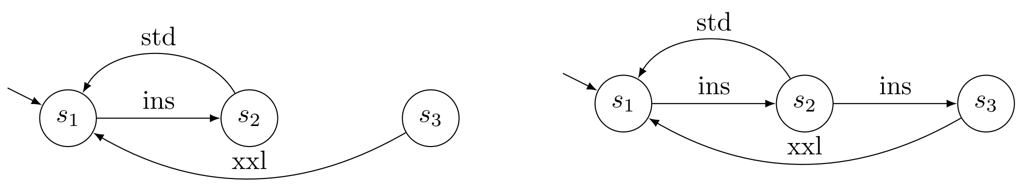


Figure 4.1: Coffee machine FTS C

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 4.2 ([9]). *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 4.2 ([37]). *The coffee machine example can be projected to its two products, which results in the LTSs in Figure 4.2.*



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

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

Figure 4.2: Projections of the coffee machine FTS

Problem statement Given an FTS M, that models the behaviour of an SPL, with products P and modal μ-calculus formula φ we want to find the products in P that satisfy φ. 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 $M_{|p} \not\models \varphi$.

We aim to find P_s in a way that utilizes the commonalities in behaviour between the different products.

5. Variability parity games

In the preliminaries we have seen how parity games can be used to check if a modal μ-calculus formula is satisfied by an LTS. A parity game can be constructed such that it contains the information needed to determine 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 determine 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 a parity game,*
- *$\mathfrak{C}$ 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 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 5.1 shows an example of a total VPG with configuration $\mathfrak{C} = \{c_1, c_2, c_3\}$.*

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 cannot 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 $i > 0$ such that π_i exists we have $(\pi_{i-1}, \pi_i) \in E$ and $c \in \theta(\pi_{i-1}, \pi_i)$.

Moves made by the players are again 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 $i > 0$ such that π_i exists and $\pi_{i-1} \in V_\alpha$ we have $\pi_i = \sigma^c_\alpha(\pi_0\pi_1 \dots \pi_{i-1})$.

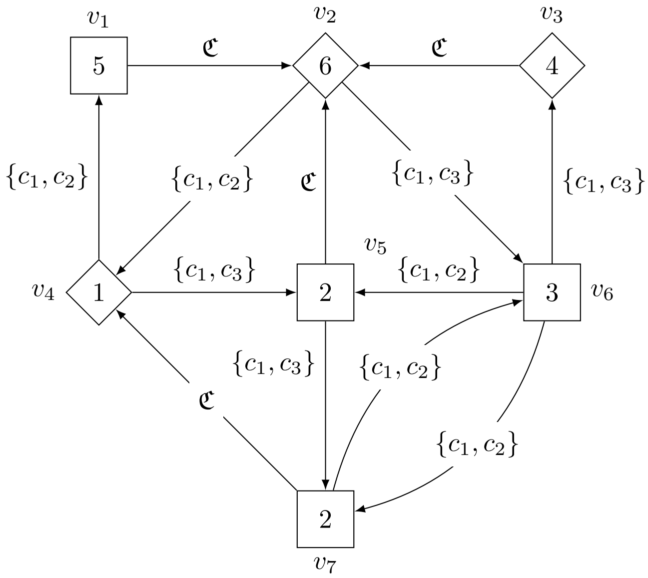


Figure 5.1: VPG with configurations $\mathfrak{C} = \{c_1, c_2, c_3\}$

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

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. $\square$

Lemma 5.2. *Any strategy σ_α^c 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 same reasoning as in Lemma 5.1 can be applied to prove this lemma. $\square$

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 and conforming to σ_α^c 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 and conforming to σ_α^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 and conforming to σ_α 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 $G_{|c}$ and conforming to σ_α 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 $v \in W_\alpha^c$. $\square$

Parity games have a unique winner for every vertex, from Theorem 5.3 we can conclude that a VPG played 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 function $\sigma_\alpha^c : V_\alpha \to V$.

Example 5.2. *Consider the VPG in Figure 5.1. When playing the game for vertex v_5 and configuration c_1 we can define strategy*

$$\sigma_1^{c_1} = \{v_5 \mapsto v_7, v_7 \mapsto v_6, v_6 \mapsto v_7, \dots\}$$

This always results in the path $v_5(v_7v_6)^\omega$ 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_1.

If the game is played for vertex v_5 and configuration c_2 the strategy $\sigma_1^{c_1}$ is not valid because the edge (v_5, v_7) is not enabled. For player 0 we can define strategy

$$\sigma_0^{c_2} = \{v_2 \mapsto v_4, v_4 \mapsto v_1, \dots\}$$

Player 1 can only play from v_5 to v_2 so the only path that conforms to $\sigma_0^{c_2}$ is $v_5(v_2v_4v_1)^\omega$, which is winning for player 0. So vertex v_5 is won by player 0 in configuration c_2.

If the game is played for vertex v_5 and configuration c_3 then player 0 can win v_5 using the strategy

$$\sigma_0^{c_3} = \{v_2 \mapsto v_6, v_3 \mapsto v_2, v_4 \mapsto v_5\}$$

Player 1 can play to v_2 or v_7. If the play goes to v_2 then player 0 plays to v_6 from where play can only go to v_3 and v_2 next. We get path $v_5(v_2v_6v_3)^\omega$, which is won by player 0. If player 1 plays to v_7 then play can only go to v_4 where player 0 plays to v_5. If play stays in this loop then player 0 wins because the highest priority occurring infinitely often is 2. If play eventually goes to v_2 then player 0 wins as well.

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 transition guards from the FTS are translated into guard sets for the VPG.

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, i.e. $\mathfrak{C} = P$.

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 owners, successors with edge guards and priorities of vertices. 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*	*Priority*
$(s, \bot)$	*0*		*0*
$(s, \top)$	*1*		*0*
$(s, \psi_1 \vee \psi_2)$	*0*	$(s, \psi_1) \mid \mathfrak{C}$ *and* $(s, \psi_2) \mid \mathfrak{C}$	*0*
$(s, \psi_1 \wedge \psi_2)$	*1*	$(s, \psi_1) \mid \mathfrak{C}$ *and* $(s, \psi_2) \mid \mathfrak{C}$	*0*
$(s, \langle a \rangle \psi)$	*0*	$(s', \psi) \mid \{c \in \mathfrak{C} \mid c \models g\}$ *for every* $s \xrightarrow{a \mid g} s'$	*0*
$(s, [a]\psi)$	*1*	$(s', \psi) \mid \{c \in \mathfrak{C} \mid c \models g\}$ *for every* $s \xrightarrow{a \mid g} s'$	*0*
$(s, \mu X.\psi)$	*1*	$(s, \psi[X := \mu X.\psi]) \mid \mathfrak{C}$	$2\lfloor adepth(X)/2 \rfloor + 1$
$(s, \nu X.\psi)$	*1*	$(s, \psi[X := \nu X.\psi]) \mid \mathfrak{C}$	$2\lfloor adepth(X)/2 \rfloor$

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

Similar to a parity game, 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 $\mathfrak{C} \backslash \bigcup\{\theta(v,w) \mid (v,w) \in E\}$. 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 5.2, 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 does not hold because once in s_1 it is possible to stay in s_1 indefinitely through an a transition. For product $\{f,g\}$ 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 5.3 shows the VPG resulting from FTS2VPG(M,φ) made total using sink vertices l_0 and l_1. Products $\{\emptyset\}, \{f\}, \{f,g\}$ are depicted as configurations c_1, c_2, c_3 respectively.

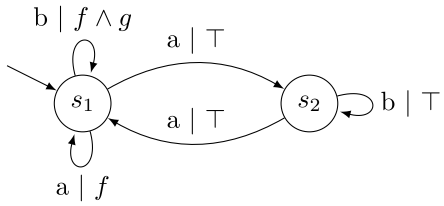


Figure 5.2: FTS M

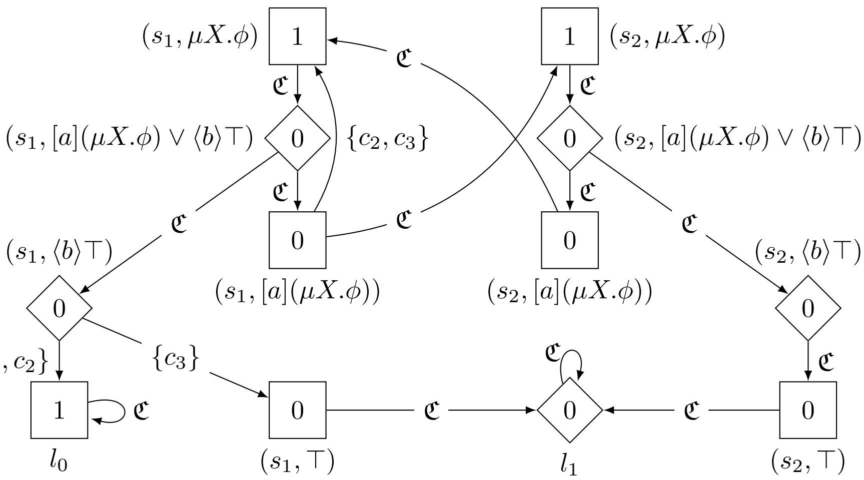


Figure 5.3: Total VPG created by *FTS2VPG* with $\phi = [a]X \vee \langle b\rangle\top$

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:

$$\begin{tikzcd}
\text{FTS } M \arrow[rr, "\mathit{FTS2VPG}(M,\varphi)"] \arrow[d, dashed, "\Pi_p"'] & & \text{VPG } \hat{G} \arrow[d, dashed, "\Pi_p"] \\
\text{LTS } M_{|p} \arrow[r, "\mathit{LTS2PG}(M_{|p},\varphi)"] & \text{PG } G & \text{PG } \hat{G}_{|p}
\end{tikzcd}$$

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

First observe that when an FTS is projected onto a product (using Definition 4.2) the FTS has the same states as the projection, we find that 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.

$$\begin{tikzcd}
\text{FTS } M \arrow[r, "FTS2VPG(M{,}\varphi)"] \arrow[d, dashed, "\Pi_p"'] & \text{VPG } \hat{G} \arrow[d, dashed, "\Pi_p"] \\
\text{LTS } M_{|p} \arrow[r, "LTS2PG(M_{|p}{,}\varphi)"] & \text{PG } \hat{G}_{|p}
\end{tikzcd}$$

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 a 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* $FTS2VPG(M, \varphi)$.

Proof. Assume $(M_{|p}, s) \models \varphi$, using the relation between LTSs and parity games (Theorem 3.2) we find that vertex (s, φ) in parity game $LTS2PG(M_{|p}, \varphi)$ is won by player 0. Using Lemma 5.4 we find that vertex (s, 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, φ) is winning in game $FTS2VPG(M, \varphi)$ for configuration p, hence $(s, \varphi) \in W_0^p$.

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

Example 5.4. *Again consider Example 5.3. We argued that* M *satisfies* φ *for products* $\{\emptyset\}$ *and* $\{f, g\}$. *We see, in the VPG in Figure 5.3, that* $(s_1, \mu X.([a]X \vee \langle b\rangle\top))$ *is indeed winning for player 0 when played for* $\{\emptyset\} = c_1$ *using the strategy*

$$\sigma_0^{c_1} = \{(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top) \mapsto (s_1, [a](\mu X.\phi)), (s_2, [a](\mu X.\phi) \vee \langle b\rangle\top) \mapsto (s_2, \langle b\rangle\top), \dots\}$$

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

For product $\{f\} = c_2$ *player* 1 *wins using strategy*

$$\sigma_1^{c_2} = \{(s_1, [a](\mu X.\phi)) \mapsto (s_1, \mu X.\phi), \dots\}$$

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 $\{f, g\} = c_3$ *player 0 wins using strategy*

$$\sigma_0^{c_3} = \{(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top) \mapsto (s_1, \langle b\rangle\top), (s_1, \langle b\rangle\top) \mapsto (s_1, \top), \dots\}$$

Using this strategy player 0 can prevent the path from 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 visualizing the verification of an FTS in Figure 5.4.

$$\begin{tikzcd}
\text{FTS } M \arrow[dr] & & \text{Modal } \mu\text{-calculus formula } \varphi \arrow[dl] \\
& \text{VPG} \arrow[d] & \\
& \begin{array}{c} M_{|p} \overset{?}{\models} \varphi \\ \text{for every valid product } p \text{ in } M \end{array} &
\end{tikzcd}$$

Figure 5.4: FTS verification using VPG

6. Solving variability parity games

In this section we inspect methods so solve VPGs, for convenience we only consider total VPGs. 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.

As shown in Chapter 5 the projections of VPGs originating from an FTS and a modal μ-calculus formula are identical to the parity games constructed from an projections of the FTS and a model μ-calculus formula. Therefore, independently solving a VPG is the same as model checking all the different products in an FTS independently

We aim to solve VPGs originating from model verification problems, such VPGs generally have certain properties that a random VPG might not have. In general (V)PGs originating from model verification problems have a relatively low number of distinct priorities compared to the number of vertices, this is because new priorities are only introduced when fixed points are nested in the μ-calculus formula. Furthermore the transition guards of FTSs are boolean formulas over features. In general these formulas will be quite simple, specifically excluding or including a small number of features.

6.1 Recursive algorithm for variability parity games

We can use the original Zielonka's recursive algorithm to solve VPGs by creating one big parity game of a VPG through a process we introduce called *unification*. This parity game can be solved using the original recursive algorithm. However, we introduce a way of representing this parity game that potentially increases performance and exploits commonalities between different configurations in the VPG.

6.1.1 Unified parity games

We can create a parity game from a VPG by taking all the projections of the VPG, which are parity games, and combining them into one parity games by taking the union of them. We call the resulting parity games the *unification* of the VPG. A parity game that is the result of a unification is called a *unified parity game*. Also any subgame of it will be called a unified parity games. A unified parity game 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 by $\hat{G}_{\downarrow}$, as*

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

where the disjoint union of two parity games is defined as

$$(V, V_0, V_1, E, \Omega) \uplus (V', V'_0, V'_1, E', \Omega') = (V \uplus V', V_0 \uplus V'_0, V_1 \uplus V'_1, E \uplus E', \Omega \uplus \Omega')$$

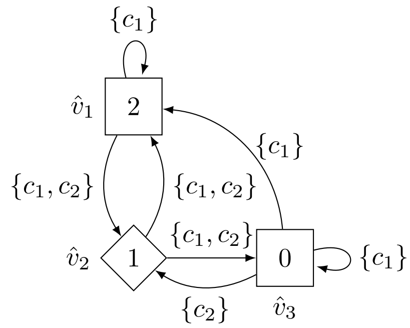
and the disjoint union of functions $\Omega : V \to \mathbb{N}$ and $\Omega' : V' \to \mathbb{N}$ is defined as

$$(\Omega \uplus \Omega')(v) = \begin{cases} \Omega(v) & \textit{if } v \in V \\ \Omega'(v) & \textit{if } v \in V' \end{cases}$$

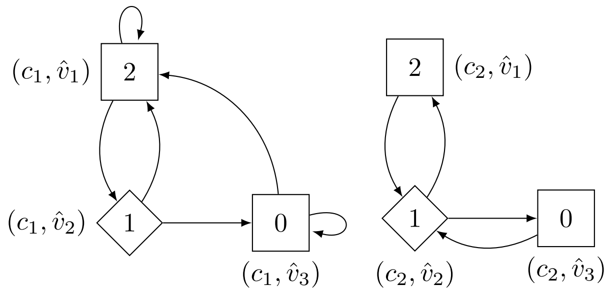
In this section we use the hat decoration $(\hat{G}, \hat{V}, \hat{E}, \hat{\Omega}, \hat{W})$ when referring to a VPG and use no hat decoration when referring to a (unified) parity game.

Every vertex in unified parity game $\hat{G}_\downarrow$ originates from a configuration and an original vertex. Therefore we can consider every vertex in a unification as a vertex-configuration pair, i.e. $V = \mathfrak{C} \times \hat{V}$. 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 do not cross configurations, so for every $((c, \hat{v}), (c', \hat{v}')) \in E$ we have $c = c'$. We call set $\hat{V}$ the *origin vertices* of a unified parity game.

Example 6.1. *Figure 6.1 shows a VPG and its the unification.*



(a) VPG with 2 configurations



(b) Unified parity game, created by unifying the two projections

Figure 6.1: A VPG with its corresponding unified parity game

Clearly solving unified parity game $\hat{G}_\downarrow$ solves all the projections of VPG $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{E}, \mathfrak{C}, \theta)$. Theorem 5.3 shows that if we solve all the projections of a VPG we have solved the VPG. So solving $\hat{G}_\downarrow$ also solves $\hat{G}$. Consider winning sets (W_0^c, W_1^c) for $\hat{G}$ played for configuration c and winning sets (Q_0, Q_1) for $\hat{G}_\downarrow$. Using Theorem 5.3 we find the following relation:

$$W_\alpha^c = \{\hat{v} \mid (c, \hat{v}) \in Q_\alpha\}$$

Projections and totality

A unified parity game can be projected onto a configuration to get one of the parity games from which it is the union. This is very similar to the projection of a VPG onto a configuration. Specifically we have for VPG $\hat{G}$ and configuration c that $\hat{G}_{|c} = (\hat{G}_\downarrow)_{|c}$. Eventhough these definitions are so similar we do need to introduce the projection of unified parity games to be able to reason about projections of subgames of unified parity games.

Definition 6.2. *The projection of unified parity game $G = (V, V_0, V_1, E, \Omega)$ to configuration c, denoted as $G_{|c}$, is the parity game $(V', V'_0, V'_1, E', \Omega)$ such that:*

- $V' = \{\hat{v} \mid (c, \hat{v}) \in V\}$,
- $V'_0 = \{\hat{v} \mid (c, \hat{v}) \in V_0\}$,
- $V'_1 = \{\hat{v} \mid (c, \hat{v}) \in V_1\}$ *and*
- $E' = \{(\hat{v}, \hat{w}) \mid ((c, \hat{v}), (c, \hat{w})) \in E\}$

One of the properties of a parity game is its totality; a game is total if every vertex has at least one outgoing vertex. The VPGs we consider are also total, meaning that every vertex has, for every configuration $c \in \mathfrak{C}$, at least one outgoing edge admitting c. Because VPGs are total their unifications are also total. Since edges in a unified parity game do not cross configurations the projection of a total unified parity game is also total.

6.1.2 Solving unified parity games

Since unified parity games are total they can be solved using Zielonka's recursive algorithm. The recursive algorithm revolves around the attractor operation. Consider the example presented in Figure 6.1. Vertices with the highest priority are

$$\{(c_1, \hat{v}_1), (c_2, \hat{v}_1)\}$$

attracting these for player 0 gives the set

$$\begin{aligned}\{(c_1, \hat{v}_1),&(c_2, \hat{v}_1),\\ (c_1, \hat{v}_2),&(c_2, \hat{v}_2),\\ &(c_2, \hat{v}_3)\}\end{aligned}$$

The algorithm tries to attract vertices $(c_1, \hat{v}_2)$ and $(c_2, \hat{v}_2)$ because they have edges to $\{(c_1, \hat{v}_1), (c_2, \hat{v}_1)\}$. So the algorithm, in this case, asks the questions: "Can vertices $(c_1, \hat{v}_2)$ and $(c_2, \hat{v}_2)$ be attracted?" We could also ask the question: "For which configurations can we attract origin vertex $\hat{v}_2$?" 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 many configurations as possible for each origin vertex. This is the idea of the collective recursive algorithm for VPGs we present next. We introduce a way of efficiently representing unified parity games and an algorithm that behaves the same as the original recursive algorithm but uses the modified representation. Using this representation we can create an attractor set algorithm that tries to attract as many configurations per origin vertex as possible instead of trying to attract each vertex individually.

In VPGs originating from FTSs a large number of edges admit all configurations (as is evident from Definition 5.3). Furthermore the sets that do not admit all configurations originate from the boolean formulas guarding transitions in the FTS. As argued before, these sets will most likely admit many configurations, because in many cases the boolean function will simply include or exclude a small number of features. Because of these two facts we hypothesise that VPGs originating from FTSs have edge guard sets that are relatively large (i.e. admit many of the configurations) and therefore we can attract many configurations at the same time per origin vertex.

6.1.3 Representing unified parity games

Unified parity games have a specific structure because they are the union of parity games that have the same vertices with the same owner and priority. Because they have the same priority we do not 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 hold:

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

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

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

Next we consider how we represent vertices and edges in a unified parity game. A set $X \subseteq (\mathfrak{C} \times \hat{V})$ can be represented as a total function $X^\lambda : \hat{V} \to 2^{\mathfrak{C}}$. The set X and function X^λ are equivalent, denoted by the operator $=_\lambda$: $2^{\mathfrak{C} \times \hat{V}} \times (\hat{V} \to 2^{\mathfrak{C}}) \to \mathbb{B}$, such that

$$X =_\lambda X^\lambda \text{ if and only if } (c, \hat{v}) \in X \iff c \in X^\lambda(\hat{v}) \text{ for all } c \in \mathfrak{C} \text{ and } \hat{v} \in \hat{V}$$

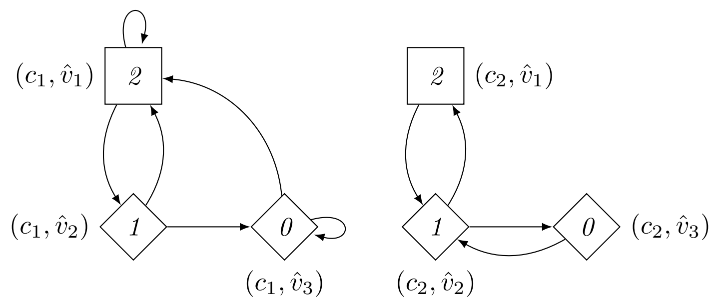
We can also represent edges as a total function $E^\lambda : \hat{E} \to 2^{\mathfrak{C}}$. The set E and function E^λ are equivalent, denoted by the operator $=_\lambda$ $2^{(\mathfrak{C} \times \hat{V}) \times (\mathfrak{C} \times \hat{V})} \times (\hat{E} \to 2^{\mathfrak{C}}) \to \mathbb{B}$, such that:

$$E =_\lambda E^\lambda \text{ if and only if } ((c, \hat{v}), (c, \hat{v}')) \in E \iff c \in E^\lambda(\hat{v}, \hat{v}') \text{ for all } c \in \mathfrak{C} \text{ and } \hat{v}, \hat{v}' \in \hat{V}$$

We use the $=_\lambda$ operator to indicate that a set and a function represent the same vertices or edges. For convenience of notation we denote equality for edges and vertices both using the $=_\lambda$ operator. 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.

Example 6.2. *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:

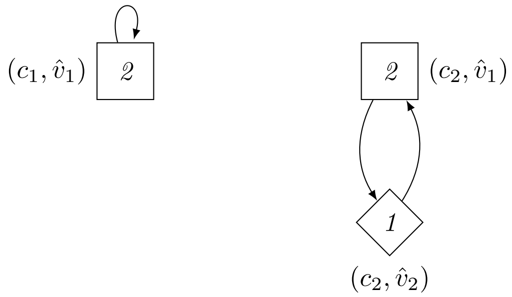
$$V = \{(c_1, \hat{v}_1), (c_2, \hat{v}_1), (c_1, \hat{v}_2), (c_2, \hat{v}_2), (c_1, \hat{v}_3), (c_2, \hat{v}_3)\}$$

$$\begin{aligned} E = \{&((c_1, \hat{v}_1), (c_1, \hat{v}_1)), ((c_1, \hat{v}_1), (c_1, \hat{v}_2)), ((c_1, \hat{v}_2), (c_1, \hat{v}_1)), \\ &((c_1, \hat{v}_2), (c_1, \hat{v}_3)), ((c_1, \hat{v}_3), (c_1, \hat{v}_1)), ((c_1, \hat{v}_3), (c_1, \hat{v}_3)), \\ &((c_2, \hat{v}_1), (c_2, \hat{v}_2)), ((c_2, \hat{v}_2), (c_2, \hat{v}_1)), ((c_2, \hat{v}_2), (c_2, \hat{v}_3)), ((c_2, \hat{v}_3), (c_2, \hat{v}_2))\} \end{aligned}$$

and function-wise:

$$
\begin{aligned}
V^\lambda = \{&\hat{v}_1 \mapsto \{c_1, c_2\}, \hat{v}_2 \mapsto \{c_1, c_2\}, \hat{v}_3 \mapsto \{c_1, c_2\}\} \\
E^\lambda = \{&(\hat{v}_1, \hat{v}_2) \mapsto \{c_1, c_2\}, (\hat{v}_2, \hat{v}_1) \mapsto \{c_1, c_2\}, (\hat{v}_2, \hat{v}_3) \mapsto \{c_1, c_2\}, \\
&(\hat{v}_1, \hat{v}_1) \mapsto \{c_1\}, \\
&(\hat{v}_3, \hat{v}_1) \mapsto \{c_1\}, \\
&(\hat{v}_3, \hat{v}_2) \mapsto \{c_2\}, \\
&(\hat{v}_3, \hat{v}_3) \mapsto \{c_1\}\}
\end{aligned}
$$

Consider the following subgame:



This subgame can be represented set-wise:

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

and function-wise:

$$
\begin{aligned}
V^\lambda = \{&\hat{v}_1 \mapsto \{c_1, c_2\}, \hat{v}_2 \mapsto \{c_2\}, \hat{v}_3 \mapsto \emptyset\} \\
E^\lambda = \{&(\hat{v}_1, \hat{v}_2) \mapsto \{c_2\}, (\hat{v}_2, \hat{v}_1) \mapsto \{c_2\}, (\hat{v}_2, \hat{v}_3) \mapsto \emptyset, \\
&(\hat{v}_1, \hat{v}_1) \mapsto \{c_1\}, \\
&(\hat{v}_3, \hat{v}_1) \mapsto \emptyset, \\
&(\hat{v}_3, \hat{v}_2) \mapsto \emptyset\}
\end{aligned}
$$

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

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

and function-wise:

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

We define the union of two functions $X^\lambda : \hat{V} \to 2^{\mathfrak{C}}$ and $Y^\lambda : \hat{V} \to 2^{\mathfrak{C}}$ point-wise:

$$(X^\lambda \cup Y^\lambda)(\hat{v}) = X^\lambda(\hat{v}) \cup Y^\lambda(\hat{v})$$

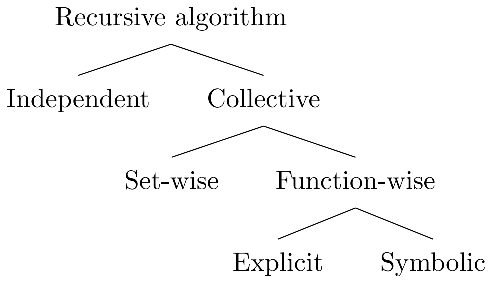
We also define the subset or equal relation point-wise:

$$X^\lambda \subseteq Y^\lambda \text{ if and only if } X^\lambda(\hat{v}) \subseteq Y^\lambda(\hat{v}) \text{ for all } \hat{v} \in \hat{V}$$

Given $X^\lambda =_\lambda X$ and $Y^\lambda =_\lambda Y$, then clearly $X^\lambda \subseteq Y^\lambda \iff X \subseteq Y$ and $X^\lambda \cup Y^\lambda =_\lambda X \cup Y$.

6.1.4 Algorithms

Using the recursive algorithm as a basis we can solve a VPG in numerous ways. First of all we can solve the projections, i.e. solve the VPG independently. Alternatively we can solve it collectively using a set-wise representation or a function-wise representation. For the function-wise representation 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 repeatedly; once for every projection. The collective set-wise approach also uses the original algorithm, applied to a unified parity game. 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 guarding the edges either admit all configurations or originate from boolean functions over the features. These boolean functions will most likely be relatively simple and are therefore specifically appropriate to represent as BDDs.

Set operations $\cap, \cup, \setminus$ over two explicit sets can be performed in $O(m)$ where m is the maximum size of the sets. This is better than the time complexity of a set operation using BDDs, which is $O(m^2)$ (as explained in preliminary section 3.4.1). 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 if the sets are not structured in a way that the BDDs are small then the running time is worse than with an explicit representation.

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 both the game graph and the sets of configurations are 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 [33]. The obstacle is that representing graphs with a large number of nodes can make the corresponding BDDs very complex if no underlying structure is known for the graph. In such a case performance decreases rapidly. For model verification problems a game graph can conceivably be represented as a BDD by using the structure of the original model to build the BDD. However this is not trivial as argued in [33]. As to not repeat work done in [33] we only consider algorithms where we represent the graph explicitly.

6.1.5 Recursive algorithm using a function-wise representation

The recursive algorithm can be modified to work with the function-wise representation of vertices and edges. The algorithm behaves the same as the original; 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.

Algorithm 3 RECURSIVEUPG(*unified parity game* $G = ($
$V^\lambda : \hat{V} \to 2^{\mathfrak{C}}$,
$\hat{V}_0 \subseteq \hat{V}$,
$\hat{V}_1 \subseteq \hat{V}$,
$E^\lambda : \hat{E} \to 2^{\mathfrak{C}}$,
$\hat{\Omega} : \hat{V} \to \mathbb{N}))$

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

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

Definition 6.3. *Given unified parity game $G = (V^\lambda, \hat{V}_0, \hat{V}_1, E^\lambda, \hat{\Omega})$, represented function-wise, and a function $U^\lambda \subseteq V^\lambda$, with $U^\lambda \neq \lambda^\emptyset$, we inductively define α-FAttr(G^λ, U^λ) such that*

$$U_0^\lambda(\hat{v}) = U^\lambda(\hat{v})$$

For $i \geq 0$:

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

Finally:

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

This attractor definition relies heavily on performing set operations on sets of configurations. We will show later that this definition is equal to the original attractor set definition (Definition 3.16).

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

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

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

Note that we can omit the modification to the partition (V_0 and V_1) because, as we have seen, we can use the partitioning from the VPG in the representation of unified parity games. As we will show later, this definition is equal to the original subgame definition (Definition 3.17).

Example 6.3. *Consider unified parity game $G = (V^\lambda, \hat{V}_0, \hat{V}_1, E^\lambda, \hat{\Omega})$, originating from a VPG with configurations $\mathfrak{C} = \{c_1, c_2, c_3\}$, represented function-wise in Figure 6.2. We annotated every edge (v, w) with the set $E^\lambda(v, w)$. All the origin vertices are depicted and for every origin vertex $\hat{v}$ we annotate the square or diamond with a label $\hat{v} \mid C$ where $C = V^\lambda(\hat{v})$. We calculate the function-wise attractor set for player 0 from origin vertex $\hat{v}_2$ with all configuration, we have*

$$U_0^\lambda = U^\lambda = \{\hat{v}_1 \mapsto \emptyset, \hat{v}_2 \mapsto \mathfrak{C}, \hat{v}_3 \mapsto \emptyset, \hat{v}_4 \mapsto \emptyset, \hat{v}_5 \mapsto \emptyset, \hat{v}_6 \mapsto \emptyset, \hat{v}_7 \mapsto \emptyset\}$$

After the first iteration we find

$$U_1^\lambda = \{\hat{v}_1 \mapsto \mathfrak{C}, \hat{v}_2 \mapsto \mathfrak{C}, \hat{v}_3 \mapsto \mathfrak{C}, \hat{v}_4 \mapsto \emptyset, \hat{v}_5 \mapsto \{c_2\}, \hat{v}_6 \mapsto \emptyset, \hat{v}_7 \mapsto \emptyset\}$$

Note that $\hat{v}_5$ can be attracted for configuration $\{c_2\}$ because $\mathfrak{C} \backslash E^\lambda(\hat{v}_5, \hat{v}_7) = \{c_2\}$, $U_0^\lambda(\hat{v}_3) = \mathfrak{C}$ and for any other origin vertex $\hat{v}$ we have $\mathfrak{C} \backslash E^\lambda(\hat{v}_5, \hat{v}) = \mathfrak{C}$.

In the next iteration we find

$$U_2^\lambda = \{\hat{v}_1 \mapsto \mathfrak{C}, \hat{v}_2 \mapsto \mathfrak{C}, \hat{v}_3 \mapsto \mathfrak{C}, \hat{v}_4 \mapsto \{c_1, c_2\}, \hat{v}_5 \mapsto \{c_2\}, \hat{v}_6 \mapsto \{c_3\}, \hat{v}_7 \mapsto \emptyset\}$$

Next iterations result in the same function, so 0*-FAttr*$(G, U^\lambda) = U_2^\lambda$. *We create subgame $G \backslash U_2^\lambda$ depicted in Figure 6.3.*

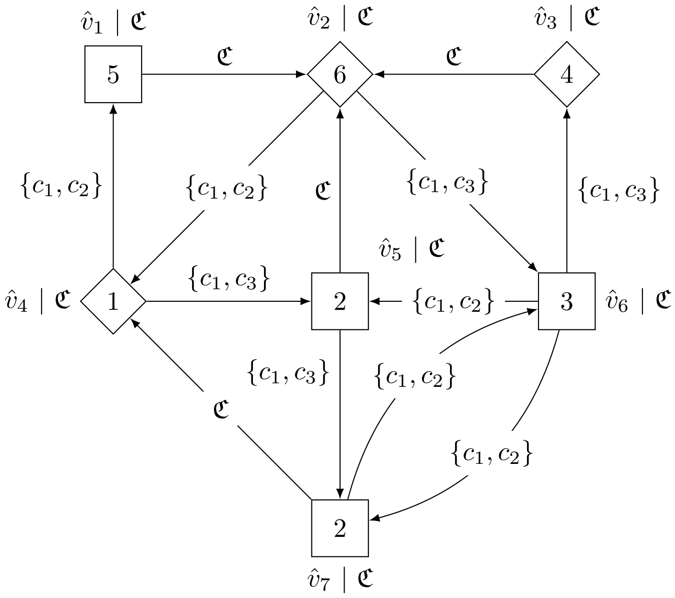


Figure 6.2: Unified parity game originating from a VPG with configurations $\mathfrak{C} = \{c_1, c_2, c_3\}$

In the next two lemma's we show that the function-wise attractor and subgame operators give results that are equal to the original attractor and subgame operators.

Lemma 6.1. *Given:*

- *unified parity game* $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$,
- *set* $U \subseteq V$, *and*
- *function* U^λ *such that* $U =_\lambda U^\lambda$

it holds that the function-wise attractor α*-FAttr*(G, U^λ) *is equivalent to the set-wise attractor* α*-Attr*(G, U) *for any* $\alpha \in \{0, 1\}$.

Proof. Let V and E be the set-wise representation of the vertices and edges for game G. Let V^λ and E^λ be the function-wise representation of the vertices and edges for game G.

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.

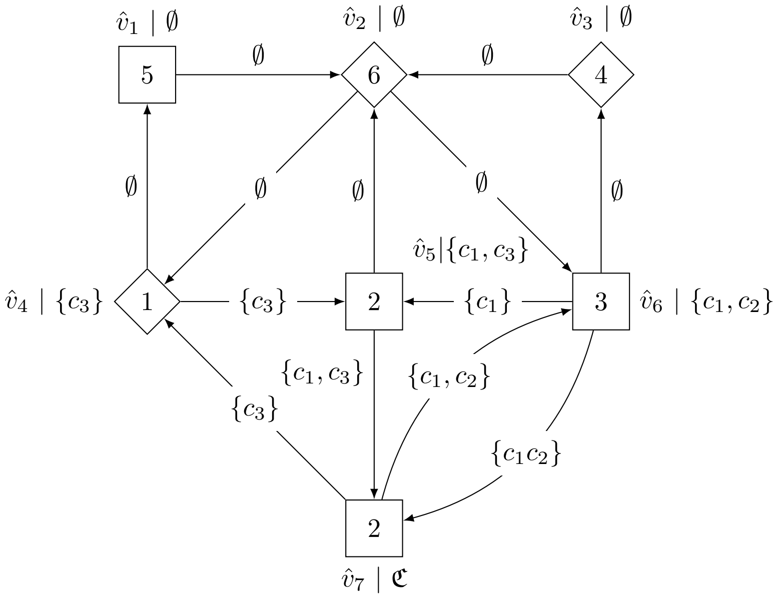


Figure 6.3: Unified parity game $G \backslash U_2^\lambda$

- 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 do not 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 $c \notin U^{\lambda}_{i+1}(\hat{v})$.

□

Lemma 6.2. *Given:*

- *unified parity game* $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$,
- *set* $U \subseteq V$ *and*
- *function* U^{λ} *such that* $U =_{\lambda} U^{\lambda}$

it holds that the subgame $G \backslash U = (V', \hat{V}_0, \hat{V}_1, E', \hat{\Omega})$ represented set-wise is equal to the subgame $G \backslash U^\lambda$ represented function-wise.

Proof. Let V^λ and E^λ denote the function-wise representations of V and E respectively. Let $G \backslash U^\lambda = (V^{\lambda'}, \hat{V}_0, \hat{V}_1, E^{\lambda'}, \hat{\Omega})$. We know $V =_\lambda V^\lambda$, $E =_\lambda E^\lambda$ and $U =_\lambda U^\lambda$. To prove: $V' =_\lambda V^{\lambda'}$ and $E' =_\lambda E^{\lambda'}$.

1. 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 3.17) 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 3.17) and $c \in V^{\lambda'}(\hat{v})$ (by Definition 6.4).

   We conclude that $V' =_\lambda V^{\lambda'}$.

2. 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 3.17). 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).

   We conclude that $E' =_\lambda E^{\lambda'}$.

□

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 6.3. *Given unified parity game $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ and $G^\lambda = (V^\lambda, \hat{V}_0, \hat{V}_1, E^\lambda, \hat{\Omega})$ which is the functional representation of G. It holds that the winning sets resulting from* RecursiveUPG(G^λ) *are equal to the winning sets resulting from* RecursivePG(G).

Proof. Proof by induction on 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.

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 $\hat{\Omega}(\hat{v}) = h$ and 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 6.1) and that the created subgames are equal (as shown in Lemma 6.2). Since the subgames are equal we can apply the theorem on it by induction and conclude that the winning sets are also equal. ☐

Theorem 5.3 shows that solving a unified parity game solves the VPG, furthermore the algorithm RECURSIVEUPG correctly solves a unified parity game. Therefore, we can conclude that for VPG $\hat{G}$ vertex $\hat{v}$ is won by player α for configuration c if and only if $c \in W^\lambda_\alpha(\hat{v})$ with $(W^\lambda_0, W^\lambda_1) =$ 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 attractor 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.

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

$A^\lambda \leftarrow U^\lambda$
Queue $Q \leftarrow \{\hat{v} \in \hat{V} \mid U^\lambda(\hat{v}) \neq \emptyset\}$
while Q is not empty **do**
    $\hat{v}' \leftarrow Q.pop()$
    for every $\hat{v}$ such that $E^\lambda(\hat{v}, \hat{v}') \neq \emptyset$ **do**
        if $\hat{v} \in \hat{V}_\alpha$ **then**
            $a \leftarrow V^\lambda(\hat{v}) \cap E^\lambda(\hat{v}, \hat{v}') \cap A^\lambda(\hat{v}')$
        else
            $a \leftarrow V^\lambda(\hat{v})$
            for every $\hat{v}''$ such that $E^\lambda(\hat{v}, \hat{v}'') \neq \emptyset$ **do**
                $a \leftarrow a \cap ((\mathfrak{C} \backslash E^\lambda(\hat{v}, \hat{v}'')) \cup A^\lambda(\hat{v}''))$
            end for
        end if
        if $a \backslash A^\lambda(\hat{v}) \neq \emptyset$ **then**
            $A^\lambda(\hat{v}) \leftarrow A^\lambda(\hat{v}) \cup a$
            $Q.push(\hat{v})$
        end if
    end for
end while
return A^λ

We prove that the result calculated by α-FATTRACTOR is equal to the definition of α-*FAttr* (Definition 6.3).

Theorem 6.4. *Given unified parity game $G = (V^\lambda, \hat{V}_0, \hat{V}_1, E^\lambda, \hat{\Omega})$, represented function-wise, and $U^\lambda \subseteq V^\lambda$, the algorithm α-*FATTRACTOR(G, U^λ) *correctly calculates α-FAttr(G, U^λ).*

Proof.

Termination. First note that the algorithm terminates. This follows from the fact that only vertices are added to Q (line 16) when something is added to A^λ (lines 14 - 15). Consider

$$dec(A^\lambda) = |\hat{V}| * |\mathfrak{C}| - \sum_{\hat{v}} |A^\lambda(\hat{v})|$$

At every iteration either $dec(A^\lambda)$ decreases or stays the same. In the latter case the size of Q decreases. After finitely many iterations $dec(A^\lambda)$ can not decrease any more so Q decreases until Q is empty and the algorithm terminates.

Soundness. To prove the soundness of the algorithm we must show that at the end of the algorithm we have for every $c \in A^\lambda(\hat{w})$ that $c \in \alpha\text{-}\mathrm{FATTRACTOR}(G, U^\lambda)(\hat{w})$. This property actually holds throughout the entire algorithm. Before the while loop (line 3) we have $A^\lambda = U^\lambda$ and the property holds trivially. Consider the beginning of a while loop iteration (the algorithm is on line 4) and assume that the property holds. The algorithm considers a number of vertices in the first for loop (line 5), let $\hat{v}$ be such a vertex. The algorithm calculates $a \subseteq \mathfrak{C}$, which is added to $A^\lambda(\hat{v})$ on line 15. Note that this is the only place in the while loop where A^λ is modified. The value calculated for a on lines 6-13 exactly reflects the definition of α-*FAttr* (Definition 6.3). Because we assumed that the property holds at the beginning of the while loop iteration we can conclude that $a \subseteq \alpha\text{-}FAttr(G, U^\lambda)(\hat{v})$. We conclude that the property is maintained during the while loop and that it holds at the end of the algorithm.

Completeness. Consider the values for U_i^λ for $\alpha\text{-}FAttr(G, U^\lambda)$ as defined in Definition 6.3.

For attractor set $\alpha\text{-}FAttr(G, U^\lambda)$ we fix strategy σ_α^c such that for every $i > 0$ and $\hat{v} \in \hat{V}_\alpha$ with $c \in U_i^\lambda(\hat{v})$ and $c \notin U_{i-1}^\lambda(\hat{v})$ we have $\sigma_\alpha^c(\hat{v}) = \hat{w}$ with $c \in U_{i-1}^\lambda(\hat{w})$. It follows from Definition 6.3 that this strategy exists and is valid for c. Furthermore, if the token is on $\hat{v}$ with $c \in (\alpha\text{-}Fattr(G, U^\lambda)(\hat{v}) \backslash U^\lambda(\hat{v}))$ then the token ends up in a vertex $\hat{w}$ with $c \in U^\lambda(\hat{w})$ for all paths that conform to σ_α^c and are valid for configuration c.

We introduce the following predicate to help with the proof of completeness. Predicate $\psi(c, \hat{v}, \hat{q})$ holds if and only if we have the following conditions:

- $\hat{q} \in Q$,
- $c \in A^\lambda(\hat{q})$ and
- there exists a path π from $\hat{v}$ to $\hat{q}$ valid for c and conforming to σ_α^c such that every vertex $\hat{w}$ between $\hat{v}$ and $\hat{q}$ in π has $c \notin A^\lambda(\hat{w})$

We prove the following loop invariant over the while loop: For all $c \in \alpha\text{-}FAttr(G, U^\lambda)(\hat{v})$ we have either $c \in A^\lambda(\hat{v})$ or $\exists_{\hat{q}} : \psi(c, \hat{v}, \hat{q})$.

When the while loop terminates Q is empty so $\psi(c, \hat{v}, \hat{q})$ never holds and therefore we have $c \in A^\lambda(\hat{v})$ which shows completeness.

Initialization: Consider the values for U_i^λ for $\alpha\text{-}FAttr(G, U^\lambda)$ as defined in Definition 6.3. We show by induction on i that the loop invariant holds for all $c \in \alpha\text{-}FAttr(G, U^\lambda)(\hat{v})$ before the while loop starts.

Base $i = 0$: Before the while loop we get $A^\lambda = U^\lambda$. So the loop invariant holds for all $c \in U_0^\lambda(\hat{v})$, furthermore $\hat{v}$ is placed in Q.

Step $i > 0$: Consider $c \in U_i^\lambda(\hat{v})$. If $c \in U_{i-1}^\lambda(\hat{v})$ then we apply induction on $i-1$ to find that the loop invariant is satisfied.

If $c \notin U_{i-1}^\lambda$ then we distinguish two cases:

* If $\hat{v} \in \hat{V}_\alpha$ then we choose $\hat{w} = \sigma_\alpha^c(\hat{v})$. By the way we constructed σ_α^c we find $c \in U_{i-1}^\lambda(\hat{w})$. We apply induction on $i-1$ to find that either $c \in A^\lambda(\hat{w})$ or $\psi(c, \hat{w}, \hat{q})$ with path π. In the former case we also find $\hat{w} \in Q$ and the path $\hat{v}\hat{w}$ satisfies $\psi(c, \hat{v}, \hat{w})$. In the latter case we construct path $\hat{v}\pi$, which satisfies $\psi(c, \hat{v}, \hat{q})$. In both cases the loop invariant is satisfied.
* If $\hat{v} \in \hat{V}_{\overline{\alpha}}$ then we pick $\hat{w}$ such that $c \in E^\lambda(\hat{v}, \hat{w})$. Using Definition 6.3 we find that $c \in U_{i-1}^\lambda(\hat{w})$. We apply induction on $i-1$ to find that either $c \in A^\lambda(\hat{w})$ or $\psi(c, \hat{w}, \hat{q})$ with path π. In the former case we also find $\hat{w} \in Q$ and the path $\hat{v}\hat{w}$ satisfies $\psi(c, \hat{v}, \hat{w})$. In the latter case we construct path $\hat{v}\pi$, which satisfies $\psi(c, \hat{v}, \hat{q})$. In both cases the loop invariant is satisfied.

Maintenance: Assume the invariant holds at the beginning of the while loop iteration (line 4), we prove that the invariant also holds at the end of the while loop iteration (line 18).

Consider $c \in \alpha\text{-}FAttr(G, U^\lambda)(\hat{v})$. If $c \in A^\lambda(\hat{v})$ by the end of the iteration then the loop invariant is maintained. Assume $c \notin A^\lambda(\hat{v})$ by the end of the iteration.

We find $\psi(c, \hat{v}, \hat{q})$ with path π. If $\hat{q}$ is not popped during this iteration then we can only get $\neg\exists_{\hat{q}'} : \psi(c, \hat{v}, \hat{q}')$ if we find a vertex $\hat{w}$ between $\hat{v}$ and $\hat{q}$ in π such that $c \in A^\lambda(\hat{w})$. Let $\hat{w}$ be the vertex closest to $\hat{v}$ in π such that $c \in A^\lambda(\hat{w})$. By the beginning of the iteration we had $c \notin A^\lambda(\hat{w})$. So c is added to $A^\lambda(\hat{w})$ during this iteration on line 15. In this case we find $\hat{w} \in Q$ because line 16 and $\psi(c, \hat{v}, \hat{w})$. So if $\hat{q}$ is not popped during the iteration the loop invariant holds. Assume $\hat{q}$ is popped during the iteration.

If by the beginning of the iteration we had $\psi(c, \hat{v}, \hat{q})$ and $\psi(c, \hat{v}, \hat{q}')$ with path π' such that $\hat{q} \neq \hat{q}'$ then by the end of the iteration we either have $\psi(c, \hat{v}, \hat{q}')$ or $\psi(c, \hat{v}, \hat{w})$ where $\hat{w}$ is a vertex between $\hat{v}$ and $\hat{q}'$ in π'. In either case the loop invariants holds. Assume that by the beginning of the iteration there is a single $\hat{q}$ for which $\psi(c, \hat{v}, \hat{q})$.

Consider path π from $\hat{v}$ to $\hat{q}$ valid for c and conforming to σ_α^c. If by the end of the iteration there is a vertex $\hat{w}$ between $\hat{v}$ and $\hat{q}$ such that $c \in A^\lambda(\hat{w})$ then we have $\psi(c, \hat{v}, \hat{w})$ and the loop invariant holds. Assume for any path from $\hat{v}$ to $\hat{q}$ valid for c and conforming to σ_α^c that there is no $\hat{w}$ between $\hat{v}$ and $\hat{q}$ such that $c \in A^\lambda(\hat{w})$ by the end of the iteration.

Let π be the path satisfying $\psi(c, \hat{v}, \hat{q})$ by the beginning of the iteration. Let $\pi = \ldots \hat{x}\hat{q}$ (note that it is possible that $\hat{x} = \hat{v}$). If $\hat{x} \in \hat{V}_\alpha$ then we find $c \in A^\lambda(\hat{x})$ by the end of the iteration which is a contradiction. We find $\hat{x} \in \hat{V}_{\overline{\alpha}}$.

If we get $\neg\exists_{\hat{q}'} : \psi(c, \hat{x}, \hat{q}')$ by the end of the iteration then player $\overline{\alpha}$ must be able to move the token from $\hat{x}$ to a vertex $\hat{w}$ such that $\neg\exists_{\hat{q}'} : \psi(c, \hat{w}, \hat{q}')$.

We have shown that the only way the loop invariant does not hold for the pair $(c, \hat{v})$ is when there exists an $\hat{x}$ owned by player $\overline{\alpha}$ such that the token can go from $\hat{x}$ to $\hat{q}$ for configuration c but also from $\hat{x}$ to some $\hat{v}'$ for which the loop invariant also not holds. Similarly to $\hat{v}$ we find that the invariant does not hold for $\hat{v}'$ when there exists a $\hat{x}'$ owned by player $\overline{\alpha}$ such that the token can go from $\hat{x}'$ to $\hat{q}$ for configuration c but also from $\hat{x}'$ to some $\hat{v}''$ for which the loop invariant also not holds. For $\hat{v}''$ we find the same property.

This induces a set of vertices $\hat{X}$ such that every $\hat{x} \in \hat{X}$ is owned by player $\overline{\alpha}$ and there is a path from $\hat{x}$ to $\hat{x}' \in \hat{X}$ valid for c and conforming to σ_α^c. We also find that for no $\hat{x} \in \hat{X}$ do we have $c \in U^\lambda(\hat{x})$ and finally that for all $\hat{x} \in \hat{X}$ we have $c \in \alpha\text{-}FAttr(G, U^\lambda)(\hat{x})$. From this we conclude that player $\overline{\alpha}$ has a strategy to keep the play in $\hat{X}$, this contradicts the properties of an attractor set. Therefore we find that the loop invariant holds for pair $(c, \hat{v})$ by the end of the iteration.

□

6.1.6 Running time

We consider the running time for solving VPG $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$ independently and collectively using the different types of representations. We use n to denote the number of vertices, e the number of edges, d the number of distinct priorities and c the number of configurations.

The original algorithm runs in $O(e * n^d)$ [17], if we run c parity games independently we get $O(c * e * n^d)$. We can also apply the original algorithm to a unified parity game (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 upper bound can be improved by using the property that a unified parity game 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 dominates 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 check if there are no more vertices in the game. For the set-wise variant this is done in $O(1)$. For the function wise algorithms this is 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 when representing an empty set.

Attractor sets

For the set-wise collective approach we can use the attractor calculation from the original algorithm which has a time complexity of $O(e)$ [42]. So for a unified parity game having $c * e$ edges we have $O(c * e)$.

The function-wise variants use 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.

During an iteration of the while loop, the first for loop considers all vertices with an edge to the vertex under consideration by the while loop. We note that during one iteration of the while loop the first for loop never considers a vertex twice. Because of this we can also conclude that during one iteration of the while loop the second for loop considers no edge twice. Since the while loop runs at most $c * n$ times and in every iteration the second for loop considers at most e edges, we conclude that the second for loop runs at most $c * n * e$ times.

The second for loop performs set operations on the set of configurations which can be done in $O(c)$ using an explicit representation. This gives a total time complexity for the attractor set of $O(n * c^2 * e)$.

Symbolic 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	$O(1)$	$O(c * e)$
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 recursion for all three algorithms at the same time. Let O_B denote the time complexity of the base case for the algorithm and let O_A denote the time complexity of the attractor set. For all variants of the algorithm we have $O_B \leq O_A$ and $O_A + O_B = O_A$.

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 parity game 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 $\bar{w}$ denote 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 $\bar{w}$ can be at most n. Furthermore since $\bar{w}$ depends on A, which depends on the maximum priority, the value $\bar{w}$ gets reset when the top priority is removed in the first recursion. We can now write down the recursion of the algorithm:

$$T(d, \bar{w}) \leq T(d-1, n) + T(d, \bar{w} - 1) + O_A$$

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

$$T(d, 0) \leq T(d-1, n) + O_A$$

Finally we have the base cases. If $d = 0$ then there are no vertices and we have the base time complexity.

$$T(0, \overline{w}) \leq O_B$$

If $d = 1$ then all the vertices have the same priority, therefore the first subgame created is empty and entirely won by player α. So we never go in the second recursion.

$$T(1, \overline{w}) \leq T(0, n) + O_A \leq O_B + O_A = O_A$$

Expanding the second recursion gives

$$
\begin{aligned}
T(d) &\leq (n+1)T(d-1) + (n+1)O_A \\
T(1) &\leq O_A
\end{aligned}
$$

We prove that $T(d) \leq (n+d)^d O_A$ by induction on d.

Base $d = 1$: $T(1) \leq O_A \leq (n+1)^1 O_A$

Step $d > 1$:

$$
\begin{aligned}
T(d) &\leq (n+1)T(d-1) + (n+1)O_A \\
&\leq (n+1)(n+d-1)^{d-1}O_A + (n+1)O_A
\end{aligned}
$$

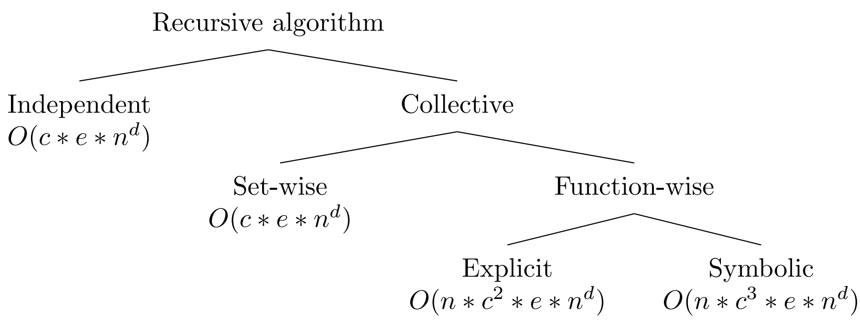
Since $n + 1 \leq n + d - 1$ we get:

$$
\begin{aligned}
T(d) &\leq (n+d-1)(n+d-1)^{d-1}O_A + (n+1)O_A \\
&\leq ((n+d-1)(n+d-1)^{d-1} + n + 1)O_A \\
&\leq (n(n+d-1)^{d-1} + d(n+d-1)^{d-1} - (n+d-1)^{d-1} + n + 1)O_A \\
&\leq (n(n+d)^{d-1} + d(n+d)^{d-1} - (n+d-1)^{d-1} + n + 1)O_A
\end{aligned}
$$

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

$$
\begin{aligned}
T(d) &\leq (n(n+d)^{d-1} + d(n+d)^{d-1})O_A \\
&\leq (n+d)(n+d)^{d-1}O_A \\
&\leq (n+d)^d O_A
\end{aligned}
$$

This gives a time complexity of $O(O_A * (n+d)^d) = O(O_A * n^d)$ because $n \geq d$. Filling in values for O_A gives the following time complexities:



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 [44], we can implement BDDs to keep 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 VPG 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 find 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 a single 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 does not spent time finding vertices that are already pre-solved. Finally, when there is only a single 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.

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*

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

if $|\mathfrak{C}| = 1$ **then**
  Let $\{c\} = \mathfrak{C}$
  $(W_0', W_1') \leftarrow$ solve $G_{|c}$ using P_0 and P_1
  return $(\mathfrak{C} \times W_0', \mathfrak{C} \times W_1')$
end if
$P_0' \leftarrow$ find vertices won by player 0 for all configurations in $\mathfrak{C}$
$P_1' \leftarrow$ find vertices won by player 1 for all configurations in $\mathfrak{C}$
if $P_0' \cup P_1' = V$ **then**
  return $(\mathfrak{C} \times P_0', \mathfrak{C} \times P_1')$
end if
$\mathfrak{C}^a, \mathfrak{C}^b \leftarrow$ partition $\mathfrak{C}$ in non-empty parts
$(W_0^a, W_1^a) \leftarrow \textsc{IncPreSolve}(G \cap \mathfrak{C}^a, P_0', P_1')$
$(W_0^b, W_1^b) \leftarrow \textsc{IncPreSolve}(G \cap \mathfrak{C}^b, P_0', P_1')$
$W_0 \leftarrow W_0^a \cup W_0^b$
$W_1 \leftarrow W_1^a \cup W_1^b$
return (W_0, W_1)

- $E' = \{e \in E \mid \theta'(e) \neq \emptyset\}$.

VPGs we consider 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, i.e. $(G \cap \mathfrak{X}) \cap \mathfrak{X}' = G \cap (\mathfrak{X} \cap \mathfrak{X}') = G \cap \mathfrak{X} \cap \mathfrak{X}'$.

6.2.1 Finding P_0 and P_1

We can find P_0 and P_1 using *pessimistic* parity games; a pessimistic parity game is a parity game created from a VPG for a player $\alpha \in \{0, 1\}$ such that the parity game 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 parity game $G_{\triangleright\alpha}$ 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 parity game that is not total might result in a finite path, in which case the player that cannot make a move loses the path.

When solving a pessimistic parity game $G_{\triangleright\alpha}$ 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 6.5. *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 parity game* $G_{\triangleright\alpha}$ *with winning sets* (P_0, P_1)

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

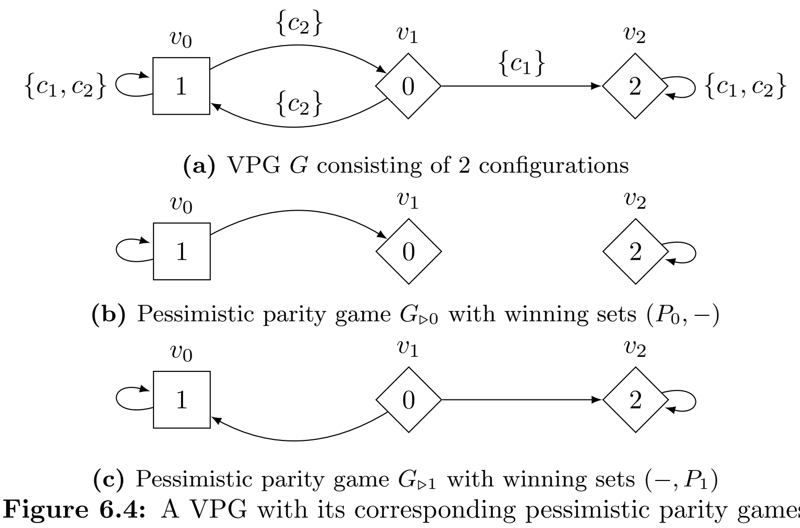
Proof. Player α has a strategy in game $G_{\triangleright\alpha}$ such that vertices in P_α are won. We show that this strategy can also be applied to game $G_{|c}$ to win the same or more vertices.

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 cannot 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 $\overline{\alpha}$ cannot win a vertex in $G_{\triangleright\alpha}$ then he/she cannot 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. Note that this strategy might be incomplete for game $G_{|c}$, it could be the case that a vertex owned by player α in game $G_{\triangleright\alpha}$ has no successor while the same vertex has successors in $G_{|c}$. In such a case the vertex is never in P_α so it is not relevant to the theorem who would win this vertex in $G_{|c}$. $\square$

Example 6.4. *Figure 6.4 shows an example VPG with corresponding pessimistic parity games. After solving the pessimistic parity games we find* $P_0 = \{v_2\}$ *and* $P_1 = \{v_0\}$.



(a) VPG G consisting of 2 configurations

(b) Pessimistic parity game $G_{\triangleright 0}$ with winning sets $(P_0, -)$

(c) Pessimistic parity game $G_{\triangleright 1}$ with winning sets $(-, P_1)$

Figure 6.4: A VPG with its corresponding pessimistic parity games

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 6.6. *Given:*

- *VPG* $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$,
- P_0 *being the winning set of pessimistic parity game* $G_{\triangleright 0}$ *for player* 0,
- P_1 *being the winning set of pessimistic parity 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 pessimistic parity game* $(G \cap \mathfrak{X})_{\triangleright\alpha}$

we have

$$P_0 \subseteq Q_0$$
$$P_1 \subseteq Q_1$$

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 $G_{\triangleright\alpha}$ has and the same or fewer 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 $(G \cap \mathfrak{X})_{\triangleright\alpha}$, i.e. $P_\alpha \subseteq Q_\alpha$.

Let $v \in P_{\overline{\alpha}}$, using Theorem 6.5 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 6.5 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 pessimistic parity games. Specifically we want a parity game algorithm that uses the vertices that are already pre-solved to efficiently solve the pessimistic parity games. Note that when there is a single configuration left we also need a parity game algorithm that uses the vertices that are already pre-solved. 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 while assuming the correctness of the Solve algorithm. Later we explore an appropriate Solve algorithm.

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

1: **if** $|\mathfrak{C}| = 1$ **then**
2:     Let $\{c\} = \mathfrak{C}$
3:     $(W_0', W_1') \leftarrow \textsc{Solve}(G_{|c}, P_0, P_1)$
4:     **return** $(\mathfrak{C} \times W_0', \mathfrak{C} \times W_1')$
5: **end if**
6: $(P_0', -) \leftarrow \textsc{Solve}(G_{\rhd 0}, P_0, P_1)$
7: $(-, P_1') \leftarrow \textsc{Solve}(G_{\rhd 1}, P_0, P_1)$
8: **if** $P_0' \cup P_1' = V$ **then**
9:     **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: $(W_0^a, W_1^a) \leftarrow \textsc{IncPreSolve}(G \cap \mathfrak{C}^a, P_0', P_1')$
13: $(W_0^b, W_1^b) \leftarrow \textsc{IncPreSolve}(G \cap \mathfrak{C}^b, P_0', P_1')$
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)

A Solve algorithm is correct when it correctly solves a parity game using sets P_0 and P_1, as long as P_0 and P_1 are in fact vertices that are won by player 0 and 1 respectively. We assume that the Solve algorithm is correct and prove that the values for P_0 and P_1 are always correct in IncPreSolve.

Lemma 6.7. *Given VPG $\hat{G}$ and assuming the correctness of* Solve. *For every* $\textsc{Solve}(G, P_0, P_1)$ *that is invoked during* $\textsc{IncPreSolve}(\hat{G}, \emptyset, \emptyset)$ *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 start the analyses after the first recursion.

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 $\hat{G}_{\rhd 0}$ and the set P_1 is the winning set for player 1 for game $\hat{G}_{\rhd 1}$. 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 $(\hat{G} \cap \mathfrak{X})_{\rhd 0}$ and P_1 being the winning set for player 1 in game $(\hat{G} \cap \mathfrak{X})_{\rhd 1}$. In general, after the kth recursion, with $k > 0$, the game is of the form $(\hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1}) \cap \mathfrak{X}^k$. Furthermore P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1})_{\rhd 0}$ and P_1 is the winning set for player 1 for game $(\hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1})_{\rhd 1}$.

Next we inspect the three places where 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 $(\hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1})_{\rhd 0}$ the vertices in P_0 are won by player 0 in game $G_{|c}$ for all $c \in \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1}$ (using Theorem 6.5). 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 $G_{|c}$ with $\{c\} = \mathfrak{C}$.

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

2. On line 6 the game $G_{\triangleright 0}$ is solved with P_0 and P_1. Because $G = \hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1} \cap \mathfrak{X}^k$ and P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ and P_1 is the winning set for player 1 for game $(\hat{G} \cap \mathfrak{X}^1 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 1}$ we can apply Lemma 6.6 to conclude that the lemma holds in this case.
3. On line 7 we apply the same reasoning and lemma to conclude that the lemma holds in this case.

□

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

Theorem 6.8. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ with winning sets (W_0^c, W_1^c) and $(W_0, W_1) =$ IncPreSolve$(G, \emptyset, \emptyset)$. For every configuration $c \in \mathfrak{C}$ it holds that:*

$$(c, v) \in W_0 \iff v \in W_0^c$$

$$(c, v) \in W_1 \iff v \in W_1^c$$

Proof. We assumed that Solve(G', P_0, P_1) correctly solves G' as long as vertices in P_0 and P_1 are won by player 0 and 1 respectively. Lemma 6.7 shows that this is always the case when invoking IncPreSolve$(G, \emptyset, \emptyset)$. We therefore find that Solve(G', P_0, P_1) always correctly solves G' during the algorithm.

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 6.5 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 6.5 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 $\mathfrak{C}^a \cup \mathfrak{C}^b = \mathfrak{C}$ and $\mathfrak{C}^a \cap \mathfrak{C}^b = \emptyset$ all vertex configuration combinations are in the winning sets and the correctness follows from induction. □

6.2.3 A parity game algorithm using P_0 and P_1

We can modify the fixed-point iteration algorithm to solve parity games using pre-solved vertices. Recall that the fixed-point iteration algorithm calculates an alternating fixed-point formula to find the winning set for player 0. When iterating fixed-point formula $\mu X.f(X)$ we choose some initial value for X and keep iterating $f(X)$ until we find $X = f(X)$. The original fixed-point iteration algorithm chooses $\emptyset$ as the initial value. In this section we show that given P_0 and P_1 we can use the fixed-point iteration algorithm, but instead of choosing initial value $\emptyset$ we choose initial value P_0. This will most likely decrease the number of iterations needed before we find $X = f(X)$. Moreover we show that we can ignore vertices in P_0 in parts of the calculation because we already know these vertices are winning. Similarly, we find that we can choose initial value $V\backslash P_1$ instead of V (where V is the set of vertices) when iterating a greatest fixed-point formula and ignore vertices in P_1.

We choose to use the fixed-point parity game algorithm because the modified version using pre-solved vertices is very similar to the original version. When experimenting with the incremental pre-solve algorithm we can compare its performance with the performance of independently solving the projections using the fixed-point iteration algorithm to get a good idea of how well the incremental pre-solve idea performs.

First recall the fixed-point formula to calculate W_0:

$$S(G) = \nu Z_{d-1}.\mu Z_{d-2}.\dots.\nu Z_0.F_0(G, Z_{d-1}, \dots, Z_0)$$

with

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

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 parity game 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

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

We will prove that formula

$$S^P(G) = \nu Z_{d-1}.\mu Z_{d-2} \dots \nu Z_0.(F_0(G, Z_{d-1}, \dots, Z_0) \cap (V\backslash P_1) \cup P_0)$$

also solves W_0 for G. Note that the formula $F_0(G, Z_{d-1}, \dots, Z_0) \cap (V\backslash P_1) \cup P_0$ is still monotonic, as shown in Lemma 6.9.

Lemma 6.9. *Given lattice $\langle 2^D, \subseteq\rangle$, monotonic function $f : 2^D \to 2^D$ and $A \subseteq D$. The functions $f^\cup(x) = f(x) \cup A$ and $f^\cap(x) = f(x) \cap A$ are also monotonic.*

Proof. Let $x, y \subseteq D$ and $x \subseteq y$ then $f(x) \subseteq f(y)$.

Let $e \in f(x) \cup A$. If $e \in f(x)$ then $e \in f(y)$ and $e \in f(y) \cup A$. If $e \in A$ then $e \in f(y) \cup A$. We find $f^\cup(x) \subseteq f^\cup(y)$.

Let $e \in f(x) \cap A$. We have $e \in f(x)$ and $e \in A$. Therefore $e \in f(y)$ and $e \in f(y) \cap A$. We find $f^\cap(x) \subseteq f^\cap(y)$. $\square$

Fixed-point iteration index

We introduce the notion of fixed-point *iteration indices* to help with the proof of S^P.

Consider the following alternating fixed-point formula:

$$\nu X_{m-1}.\mu X_{m-2} \dots \nu X_0.f(X_{m-1}, \dots, X_0)$$

Using fixed-point iteration to solve this formula results in a number of intermediate values for the iteration variables $X_{m-1}, \dots X_0$. We define an iteration index that, intuitively, indicates where in the iteration process we are. For an alternating fixed-point formula with m fixed-point variables we define an iteration index $\zeta \subseteq \mathbb{N}^m$.

When applying iteration to formula $\nu X_j.f(X)$ we start with some value for X_j^0 and calculate $X_j^{i+1} = f(X_j^i)$. So we get a list of values for X_j, however when we have alternating fixed-point formulas we might iterate X_j multiple times but get different lists of values because the values for $X_{m-1}, \dots, X_{j-1}$ have changed. We use the iteration index to distinguish between these different lists.

Iteration index $\zeta = (k_{m-1}, \dots, k_0)$ indicates where in the iteration process we are. We start at $\zeta = (0, 0, \dots, 0)$ and first iterate X_0. When we calculate X_0^1 we are at iteration index $\zeta = (0, 0, \dots, 1)$, when we calculate X_0^2 we are at iteration index $\zeta = (0, 0, \dots, 2)$ and so on. In general when we calculate a value for X_j^i then $k_j = i$ in ζ. This induces the lexicographical order

$$\begin{array}{c}
(0, \dots, 0, 0, 0)\\
(0, \dots, 0, 0, 1)\\
(0, \dots, 0, 0, 2)\\
\vdots\\
(0, \dots, 0, 1, 0)\\
(0, \dots, 0, 1, 1)\\
(0, \dots, 0, 1, 2)\\
\vdots
\end{array}$$

We define $\{k_{m-1}, \dots, k_0\} - 1 = \{k_{m-1}, \dots, k_0 - 1\}$ and $\{k_{m-1}, \dots, k_0\} + 1 = \{k_{m-1}, \dots, k_0 + 1\}$ for convenience of notation.

We write X_j^ζ to indicate the value of variable X_j at moment ζ in the iteration process. Variable X_j does not change values when a variable X_l with $j > l$ changes values; we have for indexes $\zeta = (k_{m-1}, \dots, k_j, k_{j-1}, \dots, k_0)$ and $\zeta' = (k_{m-1}, \dots, k_j, k'_{j-1}, \dots, k'_0)$ that $X_j^\zeta = X_j^{\zeta'}$.

We use the fixed-point iteration definition to define the values for X_j^ζ. Let $\zeta = (k_{m-1}, \dots, k_0)$, we have:

$$X_0^{\zeta+1} = f(X_{m-1}^\zeta, \dots, X_0^\zeta)$$

and for any even $0 < j < m$

$$X_j^{(\dots, k_j+1, \dots)} = \mu X_{j-1} \dots = \bigcup_{i \geq 0} X^{(\dots, k_j, i, \dots)}$$

and for any odd $0 < j < m$

$$X_j^{(\dots, k_j+1, \dots)} = \nu X_{j-1} \dots = \bigcap_{i \geq 0} X^{(\dots, k_j, i, \dots)}$$

Γ-games We define Γ, which transforms a parity game, to help with the proof. The Γ operator removes the pre-solved vertices from a game and modifies it such that the winners of the remaining vertices are preserved.

Definition 6.7. *Given parity game $G = (V, V_0, V_1, E, \Omega)$ with winning set W_0 such that $P_0 \subseteq W_0 \subseteq V \backslash P_1$. We define $\Gamma(G, P_0, P_1) = (V', V_0', V_1', E', \Omega')$ such that*

$$
\begin{aligned}
&V' = (V \backslash P_0 \backslash P_1) \cup \{s_0, s_1\}\\
&V_0' = (V_0 \cap V') \cup \{s_1\}\\
&V_1' = (V_1 \cap V') \cup \{s_0\}\\
&E' = (E \cap (V' \times V')) \cup \{(v, s_\alpha) \mid (v, w) \in E \wedge w \in P_\alpha\}\\
&\Omega'(v) = \begin{cases} 0 & \textit{if } v \in \{s_0, s_1\}\\ \Omega(v) & \textit{otherwise}\end{cases}
\end{aligned}
$$

Parity game $\Gamma(G, P_0, P_1)$ contains vertices s_0 and s_1 such that they have no outgoing edges and s_α is owned by player $s_{\overline{\alpha}}$. Clearly if the token ends in s_α then player α wins. Vertices that had edges to a vertex in P_α now have an edge to s_α.

Note that because s_0 and s_1 do not have successors, their priorities do not matter for winning sets of G'. Also note that this parity game is not total, as shown in [43] the formula $S(G)$ also solves non-total games.

Next we show that vertices in $V \backslash P_0 \backslash P_1$ have the same winner in games G and G'.

Lemma 6.10. *Given parity game $G = (V, V_0, V_1, E, \Omega)$ with winning set W_0 such that $P_0 \subseteq W_0 \subseteq V \backslash P_1$ and parity game $G' = \Gamma(G, P_0, P_1)$ with winning set Q_0 we have $W_0 \backslash P_0 \backslash P_1 = Q_0 \backslash \{s_0, s_1\}$.*

Proof. Let vertex $v \in V \backslash P_0 \backslash P_1$. Assume v is won by player α in G using strategy $\sigma_\alpha : V_\alpha \to V$. We construct a strategy $\sigma_\alpha' : V_\alpha' \to V'$ for game G' as follows:

$$
\sigma_\alpha'(w) = \begin{cases} s_\beta & \text{if } \sigma_\alpha(w) \in P_\beta \text{ for some } \beta \in \{0, 1\}\\ \sigma_\alpha(w) & \text{otherwise}\end{cases}
$$

This strategy maps the vertices to the same successors except when a vertex is mapped to a vertex in P_β, in which case σ_α' maps the vertex to s_β.

Let π' be a valid path in G', starting from v and conforming to σ_α'. Since vertices s_0 and s_1 do not have any successors we distinguish three cases for π':

- Assume π' ends in $s_{\overline{\alpha}}$. Let $\pi' = (x_0 \dots x_m s_{\overline{\alpha}})$ with $v = x_0$. Because s_0 and s_1 do not have successors no x_i is s_0 or s_1; we find $x_i \in V \backslash P_0 \backslash P_1$. For every $x_i x_{i+1}$ we have $(x_i, x_{i+1}) \in E'$, any such edge is also in E because the edges between vertices in $V \backslash P_0 \backslash P_1$ were left intact when creating G'. Finally we find that $(x_m, y) \in E$ with $y \in P_{\overline{\alpha}}$. There must exist a valid path $\pi = (x_0 \dots x_m y \dots)$ in game G conforming to σ_α because σ_α' and σ_α map to the same vertices for all $x_0 \dots x_{m-1}$ and x_m maps to a vertex in $P_{\overline{\alpha}}$. Player $\overline{\alpha}$ has a winning strategy from y so we conclude that π is won by $\overline{\alpha}$ in game G. Because π exists and conforms to σ_α we find that σ_α is not winning for α from v in G. This is a contradiction so we conclude that π' never ends in $s_{\overline{\alpha}}$.
- Assume π' ends in s_α. In this case player α wins the path.

- Assume π' never visits s_α or $s_{\overline{\alpha}}$. Assume the path is won by player $\overline{\alpha}$, as we argued above we find that this path is also valid in game G, conforms to σ_α and is winning for $\overline{\alpha}$. Therefore σ_α is not winning for player α from v in game G, this is a contradiction so we conclude that player α wins the path π'.

We find that π' is always won by player α in game G'. We conclude that any vertex $v \in V \backslash P_0 \backslash P_1$ won by player α in game G is also won by player α in G'.

Let $v \in V' \backslash \{s_0, s_1\}$. Let v be won by player α in game G'. Assume that v is not won by α in game G then v is won by $\overline{\alpha}$ in game G. Clearly $v \in V \backslash P_0 \backslash P_1$ so we conclude that v is won by player $\overline{\alpha}$ in game G'. This is a contradiction so v is won by player α in game G. □

Proof Using the Γ operator and the iteration indices we can now prove the correctness of S^P.

Theorem 6.11. *Given parity game $G = (V, V_0, V_1, E, \Omega)$ with winning set W_0 such that $P_0 \subseteq W_0 \subseteq V \backslash P_1$. The formula*

$$S^P(G) = \nu Z_{d-1}.\mu Z_{d-2} \dots \nu Z_0.(F_0(G, Z_{d-1}, \dots, Z_0) \cap (V \backslash P_1) \cup P_0)$$

correctly solves W_0 for G.

Proof. Let $G' = (V', V'_0, V'_1, E', \Omega') = \Gamma(G, P_0, P_1)$. We consider $S(G')$, which calculates the winning set for player 0 for game G'. Formula $F_0(G', Z_{d-1}, \dots, Z_0)$ will always include s_0 and never include s_1, regardless of the values for $Z_{d-1} \dots Z_0$. Clearly any $\nu Z_i \dots$ or $\mu Z_i \dots$ contains s_0 and does not contain s_1. As shown in [13] we can start the iteration of least fixed-point formula $\mu X.f(X)$ at any value $X^0 \subseteq \mu X.f(X)$. Similarly, we can start the iteration of greatest fixed-point formula $\nu X.f(X)$ at any value $X^0 \supseteq \nu X.f(X)$. So we can calculate $S(G')$ using fixed-point iteration, starting least fixed-point variables at $\{s_0\}$ and greatest fixed-point variables at $V' \backslash \{s_1\}$.

We can also calculate $S^P(G)$ using fixed-point iteration starting at P_0 and $V \backslash P_1$ because clearly any $\nu Z_i \dots$ or $\mu Z_i \dots$ contains all vertices from P_0 and none from P_1.

We prove the theorem by going through the iteration process of $S^P(G)$ and $S(G')$ simultaneously. We write Z_i to denote variables in $S(G')$ and Y_i to denote variables in $S^P(G)$. We will show that for any iteration index ζ any iteration variable Z_i^ζ is equal to Y_i^ζ for vertices $V \backslash P_0 \backslash P_1$, that is $Y_i^\zeta \backslash P_0 \backslash P_1 = Z_i^\zeta \backslash \{s_0, s_1\}$. We only prove that this is the case when we start iteration of $S^P(G)$ at P_0 and $V \backslash P_1$ and start iteration of $S(G')$ at $\{s_0\}$ and $V' \backslash \{s_1\}$. As argued above, starting at these values correctly calculates $S^P(G)$ and $S(G')$.

Trivially, for any ζ and $i \in [0, d-1]$ we have $P_0 \subseteq Y_i^\zeta \subseteq V \backslash P_1$ and $\{s_0\} \subseteq Z_i^\zeta \subseteq V \backslash \{s_1\}$.

We define operator $\simeq: V \times V' \to \mathbb{B}$ such that for $Y \subseteq V$ and $Z \subseteq V'$ we have $Y \simeq Z$ if and only if:

$$Y \backslash P_0 \backslash P_1 = Z \backslash \{s_0, s_1\}$$

We prove, by induction on ζ, that for any $\zeta = (k_{d-1}, \dots, k_0)$ we have $Y_i^\zeta \simeq Z_i^\zeta$ for every $i \in [0, d-1]$.

Base $\zeta = (0, 0, \dots, 0)$: we have for least fixed-point variables Z_i^ζ and Y_i^ζ the values $\{s_0\}$ and P_0, clearly $Y_i^\zeta \simeq Z_i^\zeta$.

For greatest fixed-point variables Z_j^ζ and Y_j^ζ we have $Z_j^\zeta\backslash\{s_0, s_1\} = V\backslash P_1\backslash P_0$. So we find $Y_j^\zeta \simeq Z_j^\zeta$.

Step: Consider $\zeta = (k_{d-1}, \dots, k_0)$. Let $j \in [0, d-1]$. If $k_j = 0$ then $Z_j^\zeta = Z_j^{(0,0,\dots,0)}$ and $Y_j^\zeta = Y_j^{(0,0,\dots,0)}$, furthermore $Z_j^{(0,0,\dots,0)} \simeq Y_j^{(0,0,\dots,0)}$ so we find $Y_j^\zeta \simeq Z_j^\zeta$. If $k_j > 0$ then we distinguish three cases for j to show that $Y_j^\zeta \simeq Z_j^\zeta$:

- Case $j = 0$: We have the following equations:

$$Y_0^\zeta = F_0(G, Y_{d-1}^{\zeta-1}, \dots, Y_0^{\zeta-1}) \cap (V\backslash P_1) \cup P_0$$

  and

$$Z_0^\zeta = F_0(G', Z_{d-1}^{\zeta-1}, \dots, Z_0^{\zeta-1})$$

  By induction we find $Y_i^{\zeta-1} \simeq Z_i^{\zeta-1}$ for all $i \in [0, d-1]$.

  Consider vertex $v \in V\backslash P_0\backslash P_1$. We distinguish two cases:

  – Assume $v \in V_0$.

    If $v \in Y_0^\zeta$ then v must have an edge in game G to w such that $w \in Y_{\Omega(w)}^{\zeta-1}$. We find $w \notin P_1$ because vertices from P_1 are never in the iteration variable. If $w \in P_0$ then it follows from the way we created G' that in G' there exists an edge from v to s_0 and since s_0 is always in the iteration variable we find $v \in Z_0^\zeta$. If $w \notin P_0$ then because $Y_{\Omega(w)}^{\zeta-1} \simeq Z_{\Omega(w)}^{\zeta-1}$ we find $w \in Z_{\Omega(w)}^{\zeta-1}$ and therefore $v \in Z_0^\zeta$.

    If $v \in Z_0^\zeta$ then v must have an edge in game G' to w such that $w \in Z_{\Omega(w)}^{\zeta-1}$. We find $w \neq s_1$ because s_1 is never in the iteration variable. If $w = s_0$ then it follows from the way we created G' that in G there exists an edge from v to a vertex in P_0 and since any vertex in P_0 is always in the iteration variable we find $v \in Y_0^\zeta$. If $w \neq s_0$ then because $Y_{\Omega(w)}^{\zeta-1} \simeq Z_{\Omega(w)}^{\zeta-1}$ we find $w \in Y_{\Omega(w)}^{\zeta-1}$ and therefore $v \in Y_0^\zeta$.

  – Assume $v \in V_1$.

    If $v \in Y_0^\zeta$ then for any successor w of v in game G it holds that $w \in Y_{\Omega(w)}^{\zeta-1}$. Consider successor x of v in game G'. We distinguish three cases:

    * $x = s_0$: In this case $x \in Z_{\Omega(x)}^{\zeta-1}$ because s_0 is always in the iteration variables.
    * $x = s_1$: Because of the way G' is constructed we find vertex v must have a successor w in P_1 in game G. However we found $w \in Y_{\Omega(w)}^{\zeta-1}$. This is a contradiction because vertices in P_1 are never in the iteration variables. So this case can not happen.
    * $x \notin \{s_0, s_1\}$: We have $x \in V'\backslash\{s_0, s_1\}$ and therefore x is also a successor of v in game G. We find $x \in Y_{\Omega(x)}^{\zeta-1}$ and because $Y_{\Omega(x)}^{\zeta-1} \simeq Z_{\Omega(x)}^{\zeta-1}$ we have $x \in Z_{\Omega(x)}^{\zeta-1}$.

    We always find $x \in Z_{\Omega(x)}^{\zeta-1}$, therefore $v \in Z_0^\zeta$.

    If $v \in Z_0^\zeta$ then for any successor w of v in game G' it holds that $w \in Z_{\Omega(w)}^{\zeta-1}$. Consider successor x of v in game G. We distinguish three cases:

    * $x \in P_0$: In this case $x \in Y_{\Omega(x)}^{\zeta-1}$ because vertices in P_0 are always in the iteration variables.

- $*$ $x \in P_1$: Because of the way G' is constructed we find vertex v must have successor s_1 in game G', however we found that for any successor w of v in game G' we have $w \in Z^{\zeta-1}_{\Omega(w)}$. This is a contradiction because s_1 is never in the iteration variable. So this case can not happen.
- $*$ $x \in V \backslash P_0 \backslash P_1$: We find that x is also a successor of v in game G'. We find $x \in Z^{\zeta-1}_{\Omega(w)}$ and because $Y^{\zeta-1}_{\Omega(x)} \simeq Z^{\zeta}_{\Omega(x)}$ we have $x \in Y^{\zeta}_{\Omega(x)}$.

We always find $x \in Y^{\zeta-1}_{\Omega(x)}$, therefore $v \in Y^{\zeta}_0$.

- Case $j > 0$ being even: We have

$$Z^{\zeta}_j = \mu Z_{j-1} \cdots = \bigcup_{i \geq 0} Z^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$$

and

$$Y^{\zeta}_j = \mu Y_{j-1} \cdots = \bigcup_{i \geq 0} Y^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$$

Let $v \in V \backslash P_0 \backslash P_1$.

If $v \in Z^{\zeta}_j$ then there exists some i such that $v \in Z^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$. Since $\{k_{d-1},\ldots,k_j - 1, i, \ldots\} < \zeta$ we apply induction to find $Y^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1} \simeq Z^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$. Because $v \in V \backslash P_0 \backslash P_1$ we find $v \in Y^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$ and therefore $v \in Y^{\zeta}_j$.

If $v \in Y^{\zeta}_j$ then we apply symmetrical reasoning to find $v \in Z^{\zeta}_j$.

- Case $j > 0$ being odd: We have

$$Z^{\zeta}_j = \nu Z_{j-1} \cdots = \bigcap_{i \geq 0} Z^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$$

and

$$Y^{\zeta}_j = \nu Y_{j-1} \cdots = \bigcap_{i \geq 0} Y^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$$

Let $v \in V \backslash P_0 \backslash P_1$.

If $v \in Z^{\zeta}_j$ then for all $i \geq 0$ we have $v \in Z^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$. Assume $v \notin Y^{\zeta}_j$, there must exist an $l \geq 0$ such that $v \notin Y^{\{k_{d-1},\ldots,k_j-1,l,\ldots\}}_j$. Since $\{k_{d-1},\ldots,k_j - 1, l, \ldots\} < \zeta$ we apply induction to find $Y^{\{k_{d-1},\ldots,k_j-1,l,\ldots\}}_{j-1} \simeq Z^{\{k_{d-1},\ldots,k_j-1,l,\ldots\}}_{j-1}$. Because $v \in V \backslash P_0 \backslash P_1$ we find $v \notin Z^{\{k_{d-1},\ldots,k_j-1,i,\ldots\}}_{j-1}$ which is a contradiction so we have $v \in Y^{\zeta}_j$.

If $v \in Y^{\zeta}_j$ then we apply symmetrical reasoning to find $v \in Z^{\zeta}_j$.

This proves that for any ζ we have $Y^{\zeta}_i \simeq Z^{\zeta}_i$ for every $i \in [0, d-1]$.

We have shown that when starting the iteration of $S(G')$ and $S^P(G)$ at specific values we get identical results for vertices in $V \backslash P_0 \backslash P_1$. We chose these values such that they solve the formulas correctly, so we conclude that $S(G') \backslash \{s_0, s_1\} = S^P(G) \backslash P_0 \backslash P_1$. Lemma 6.10 shows that $S(G')$

correctly solves vertices in $V \backslash P_0 \backslash P_1$ for game G. So $S^P(G)$ also correctly solves vertices $V \backslash P_0 \backslash P_1$ for game G.

Moreover, any vertex in P_0 is in $S^P(G)$, which is correct because P_0 vertices are winning for player 0. Any vertex in P_1 is not in $S^P(G)$, which is correct because P_1 vertices are winning for player 1. We conclude that all vertices are correctly solved by $S^P(G)$. ☐

Algorithm

We use the original fixed-point algorithm presented in [4] and modify it such that its starts in iteration at P_0 and $V \backslash P_1$. Moreover, we ignore vertices in P_0 or P_1 in the diamond and box calculation. Finally we always add vertices in P_0 to the results of the diamond and box operator. The correctness follow from Theorem 6.11 and [4, 43].

Note that in [4] total games are used. However, it is argued that the algorithm correctly solves the formula presented in [43]. The only property of parity games that is used in this argumentation is that parity games have a unique owner and priority. Clearly this is still the case for total parity games so the algorithm correctly solves the formula presented in [43]. In [43] it is shown that the formula also correctly solves non-total parity games.

Algorithm 7 Fixed-point iteration with P_0 and P_1

function FPITER($G = (V, V_0, V_1, E, \Omega)$, $P_0 \subseteq V, P_1 \subseteq V$)
    for $i \leftarrow d-1, \ldots, 0$ **do**
        INIT(i)
    end for
    repeat
        $Z'_0 \leftarrow Z_0$
        $Z_0 \leftarrow P_0 \cup$ DIAMOND() $\cup$ BOX()
        $i \leftarrow 0$
        while $Z_i = Z'_i \wedge i < d-1$ **do**
            $i \leftarrow i + 1$
            $Z'_i \leftarrow Z_i$
            $Z_i \leftarrow Z_{i-1}$
            INIT($i-1$)
        end while
    until $i = d-1 \wedge Z_{d-1} = Z'_{d-1}$
    return $(Z_{d-1}, V \backslash Z_{d-1})$
end function

function INIT(i)
    $Z_i \leftarrow P_0$ if i is odd, $V \backslash P_1$ otherwise
end function

function DIAMOND
    return $\{v \in V_0 \backslash P_0 \backslash P_1 \mid \exists_{w \in V}\ (v, w) \in E \wedge w \in Z_{\Omega(w)}\}$
end function

function BOX
    return $\{v \in V_1 \backslash P_0 \backslash P_1 \mid \forall_{w \in V}\ (v, w) \in E \implies w \in Z_{\Omega(w)}\}$
end function

This algorithm can be used as a SOLVE algorithm in INCPRESOLVE since it solves parity games using 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. We use n to denote the number of vertices, e the number of edges, d the number of distinct priorities and c the number of configurations.

The fixed-point iteration algorithm without P_0 and P_1 runs in $O(e * n^d)$ [4]. We can use this algorithm to solve G independently, i.e. 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. In the worst case there are c leaves so 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 $O(c + 2c - 2) = O(c)$ parity games that are being solved by IncPreSolve. In the worst case the values 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.

Running time in practice

The incremental pre-solve algorithm will, most likely, need to solve more (pessimistic) parity games than an independent approach would need to solve. However, the algorithm keeps trying to increase the number of pre-solved vertices which might speed up the solving of these games. This would cause the algorithm to solve the (pessimistic) increasingly quickly. Therefore, we hypothesize that, even though more games are solved, the increment pre-solve algorithm performs better than an independent approach.

7. Locally solving (variability) parity games

As discussed in the preliminaries, parity games can be solved either globally or locally. Similar to parity games we can solve VPGs either globally or locally. When locally solving a VPG for vertex $\hat{v}_0$ we determine for every configuration the winner of vertex $\hat{v}_0$. 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 do not 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 in 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 parity game algorithms we have seen: Zielonka's recursive algorithm and fixed-point iteration algorithm. Furthermore, we introduce local variants for the novel VPG algorithms: 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. These local variants can be used to solve VPGs locally and independently.

7.1.1 Local recursive algorithm for parity games

The recursive algorithm has two recursion steps. The first recursion step gives two winning sets which are used to find set B such that all the vertices in B won are by a particular player. The second recursion step solves the remaining part of the game. When locally solving a parity game we can sometimes avoid entering the second recursion when we already found the vertex we are interested in to be in set B. In this section we introduce an algorithm that utilizes this idea to create a local variant of the recursive algorithm.

First we inspect the notion of *traps* [45]. Traps are used in the original proof of the recursive algorithm and we will use them again to reason about a local variant of the recursive algorithm. Consider total parity game (V, V_0, V_1, E, Ω) in the next definition and two lemma's.

Definition 7.1. *[45] Set $X \subseteq V$ is an α-trap in G if and only if player $\overline{\alpha}$ can play in such a way that once the token is in X, it will not leave X.*

Lemma 7.1. *[45] Set $V \backslash \alpha\text{-}Attr(G, X)$ is an α-trap in G for any non-empty set $X \subseteq V$.*

Lemma 7.2. *[45] Let $X \subseteq V$ be an α-trap in G. Then $\overline{\alpha}\text{-}Attr(G, X)$ is also an α-trap in G.*

Observe that a winning set W_α of parity game G is an $\overline{\alpha}$-trap in G. 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 show that if a vertex in the first recursion is won by player $\overline{\alpha}$, as calculated in the recursive algorithm, then this vertex is also won by player $\overline{\alpha}$ in the game itself.

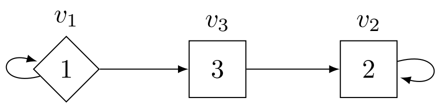
Lemma 7.3. *Given total parity game $G = (V, V_0, V_1, E, \Omega)$, player $\alpha \in \{0, 1\}$ and non-empty set $X \subseteq V$ it holds that the winning set $W_{\overline{\alpha}}$ for player $\overline{\alpha}$ in $G' = G \backslash \alpha\text{-}Attr(G, X)$ is an α-trap in G and all vertices in $W_{\overline{\alpha}}$ are winning for $\overline{\alpha}$ in G.*

Proof. Using Lemma 7.1 we find that $V' = V \backslash \alpha\text{-}Attr(G, X)$ is a an α-trap in G. Set $W_{\overline{\alpha}}$ is an α-trap in G because if α could escape to $V \backslash V'$ then V' would not be an α-trap in G and if α could escape to $V' \backslash W_{\overline{\alpha}}$ then $W_{\overline{\alpha}}$ would not be an α-trap in G'. Moreover keeping the token in $W_{\overline{\alpha}}$ causes player $\overline{\alpha}$ to win the path because the strategy that was winning in G' can also be applied in G. □

Let v_0 be the vertex we are trying to solve locally. We could argue that if the algorithm finds v_0 to be winning for player $\overline{\alpha}$ in the first recursion of the algorithm then we can terminate and report v_0 to be winning for $\overline{\alpha}$. Using the lemma above we find that indeed v_0 is winning for player $\overline{\alpha}$ in game G when v_0 is winning for $\overline{\alpha}$ in the first recursion. However game G itself might be the subgame of some game H. Vertex v_0 is winning for $\overline{\alpha}$ in G and in the subgame created in the first recursion; however if we want to terminate early then v_0 must also be winning for $\overline{\alpha}$ in game H. If v_0 is not winning for $\overline{\alpha}$ in game H and game G is the first subgame created from game H then in order to correctly solve game H we need the complete winning sets of G. In the conjecture below we express this property. If the conjecture holds we can terminate when we find v_0 in the first recursion to be winning for player $\overline{\alpha}$. However, as is shown below, the conjecture does not hold.

Conjecture 7.4 (Disproven)**.** *For any* RecursivePG$(G' \backslash A)$*, with winning sets* (W'_0, W'_1)*, 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; 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($G \backslash A \backslash \tilde{B}$)
$\tilde{W}''_0 = \{v_2\}, \tilde{W}''_1 = \emptyset$
$W'_0 = W'_{\overline{\alpha}} = \{v_2\}$

| $W'_1 = W'_\alpha = \{v_1\}$
| $B = V$
| | RecursivePG($G \backslash B$)
| $W_0 = W_{\overline{\alpha}} = V$

This counterexample disproves the conjecture. □

When v_0 is not winning for player $\overline{\alpha}$ in the first recursion then we need the complete winning sets to calculate B and go in the next recursion. We extend the recursive algorithm with a variable $\Delta \subseteq \{0, 1\}$. The algorithm either returns partial winning sets, containing v_0, when v_0 is won by player $\beta \in \Delta$ or the algorithm returns complete winning sets. This solves the problem, that Conjecture 7.4 does not hold, by only allowing the algorithm to terminate before the second recursion when $\overline{\alpha}$ is in Δ. Pseudo code for the algorithm using Δ is provided in Algorithm 8.

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

if $V = \emptyset$ **then**
    return $(\emptyset, \emptyset)$
end if
$h \leftarrow \max\{\Omega(v) \mid v \in V\}$
$\alpha \leftarrow 0$ if h is even and 1 otherwise
$U \leftarrow \{v \in V \mid \Omega(v) = h\}$
$A \leftarrow \alpha\text{-}Attr(G, U)$
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
    $B \leftarrow \overline{\alpha}\text{-}Attr(G, W'_{\overline{\alpha}})$
    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
return (W_0, W_1)

To prove the correctness we show any vertex in the winning set W_γ resulting from RecursivePGLocal is indeed winning for player γ and that either the winning sets completely partition the graph or that vertex v_0 is in winning set W_β such that $\beta \in \Delta$.

The proof is in many ways similar to the proof given in [45]. We repeat part of the original reasoning in the following lemma.

Lemma 7.5. *Given:*

- *total parity game* $G = (V, V_0, V_1, E, \Omega)$*,*
- *non-empty set* $X \subseteq V$ *such that* X *is an* α*-trap in* G *and all vertices in* X *are winning for player* $\overline{\alpha}$ *in game* G*,*
- *subgame* $G' = G \backslash \overline{\alpha}\text{-}Attr(G, X)$

it holds that the winner of any vertex in G' *is also the winner of that vertex in* G*.*

Proof. Let (W'_0, W'_1) be the winning sets of game G'. Using Lemma 7.2 we find that $\overline{\alpha}\text{-}Attr(G, X)$ is an α-trap in G. Using lemma 7.1 we find that $V' = V \backslash \overline{\alpha}\text{-}Attr(G, X)$ is an $\overline{\alpha}$-trap in G.

Consider winning set W'_α for game G'. Set W'_α is an $\overline{\alpha}$-trap in G'. In game G we find that player $\overline{\alpha}$ can not escape W'_α by going to $V \backslash V'$ because V' is an $\overline{\alpha}$-trap in G. Furthermore player $\overline{\alpha}$ can not escape to $W'_{\overline{\alpha}}$ because W'_α is an $\overline{\alpha}$-trap in G'. We find that W_α is an $\overline{\alpha}$-trap in G. Finally we know that the strategy for player α used in game G' is still applicable in game G to win the vertices in W'_α for game G.

Consider winning set $W'_{\overline{\alpha}}$ for game G'. Set $W'_{\overline{\alpha}}$ is an α-trap in G'. In game G we find that player α can not escape $W'_{\overline{\alpha}}$ by going to W'_α, however he/she might escape by going to $V \backslash V' = \overline{\alpha}\text{-}Attr(G, X)$. When play goes to $\overline{\alpha}\text{-}Attr(G, X)$ then player $\overline{\alpha}$ can get the play into X which is an α-trap in G and is winning for player $\overline{\alpha}$ in G. So when the token is in $W'_{\overline{\alpha}}$ either the play stays there and $\overline{\alpha}$ uses the strategy from game G' to win or the play goes to X where player $\overline{\alpha}$ can keep the play and win. □

Theorem 7.6. *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, Q_1) *for game* G*. We have for sets* $(W_0, W_1) =$ RecursivePGLocal(G, v_0, Δ) *that at least one of the following statements hold:*

(I) For some $\beta \in \Delta$ *we have* $v_0 \in W_\beta$*,* $W_0 \subseteq Q_0$ *and* $W_1 \subseteq Q_1$*.*

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

Proof. First note that both statements require the vertices in the winning sets to be in the correct winning sets. Statement (I) only allows winning sets to be incomplete, it does not allow vertices to be in a winning set when that vertex is not actually won by that player. Furthermore note that statement (II) simply states that the game is solved completely.

Proof by induction on G.

Base: When G is empty then the algorithm returns $(\emptyset, \emptyset)$ in which case statement (II) holds trivially.

Step: The algorithm considers the highest priority in the game and assigns the parity of this priority to α. The set U contains all vertices with this priority and A contains all vertices from where player α can force the play into U.

The first recursion removes vertices in A from the game, since A is non-empty we can apply induction to find that at least one of the two statements hold for $G\backslash A$ and winning sets (W'_0, W'_1).

If $W'_{\overline{\alpha}} = \emptyset$ (line 13) then statement (II) must be true for $G\backslash A$ because α is never passed into the Δ parameter for recursion $G\backslash A$ (lines 9 and 11). We find that indeed all vertices in $G\backslash A$ are won by player α, moreover player α has a strategy σ_α for $G\backslash A$ that is winning for all vertices in $V\backslash A$. Clearly this strategy can also be applied game to G. Consider valid path π in game G conforming to σ_α. When this path eventually stays in $V\backslash A$ then player α wins because σ_α is winning here. Otherwise the path visits A infinitely often, in which case player α can force the play infinitely often into U and therefore the highest priority occurring infinitely often has parity α. So player α wins all vertices in V and the algorithm returns winning sets accordingly; statement (II) holds.

If $W'_{\overline{\alpha}} \neq \emptyset$ we use Lemma 7.3 and induction on $G\backslash A$ to find that all vertices in $W'_{\overline{\alpha}}$ are also won by player $\overline{\alpha}$ in game G.

The algorithm continues with calculating set B (line 17). If $\overline{\alpha} \in \Delta$ and $v_0 \in B$ (line 18) then the algorithm returns all vertices in B to be winning for player $\overline{\alpha}$. As argued, all vertices in $W'_{\overline{\alpha}}$ are winning for player $\overline{\alpha}$ in game G. Clearly any vertex where player $\overline{\alpha}$ can force the play to $W'_{\overline{\alpha}}$ is also winning for player $\overline{\alpha}$. So all vertices in B are winning for player $\overline{\alpha}$ in G. Because $v_0 \in B$ and $\overline{\alpha} \in \Delta$, statement (I) holds for game G.

If $\overline{\alpha} \notin \Delta$ or $v_0 \notin B$ then statement(II) holds for game $G\backslash A$ (because $W'_{\overline{\alpha}} \subseteq B$).

The algorithm goes into the second recursion (line 22). Using induction we find that any vertex $v \in W''_\beta$ is indeed won by player β in game $G\backslash B$. The algorithm returns v to be winning for player β in game G, using Lemma 7.5 we find this to be correct. Note that we can apply Lemma 7.5 because statement (II) holds for $G\backslash A$ and using Lemma 7.3 we find that $W'_{\overline{\alpha}}$ is an α-trap in G. The algorithm also returns B to be winning for $\overline{\alpha}$, which is correct because it contains vertices such that player $\overline{\alpha}$ can play to $W'_{\overline{\alpha}}$ where player $\overline{\alpha}$ wins. If statement (II) holds for $G\backslash B$ then statement (II) also holds for G. If statement (I) holds for $G\backslash B$ then statement (I) also holds for G because we pass Δ into the recursion unmodified. □

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 the game is solved locally.

The worst-case 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 then the algorithm behaves the same as the original, so its worst-case time complexity is the same.

7.1.2 Local fixed-point iteration algorithm

The fixed-point iteration algorithm can be modified to locally solve a parity 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 identical worst-case time complexities of $O(e * n^d)$.

7.2 Locally solving variability parity games

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

In the previous section we have seen a local variant of Zielonka's recursive algorithm for parity games that uses $\Delta \subseteq \{0, 1\}$ to indicate for which player we are trying to find the specific vertex.

Consider VPG $\hat{G}$ with configuration set $\mathfrak{C}$ and origin vertex $\hat{v}_0$ which we are trying to solve locally. Unified parity game $\hat{G}_\downarrow$ contains vertices $\mathfrak{C} \times \{\hat{v}_0\}$. If we find the winning player for each of these vertices we have solved the VPG locally. We are going to solve a unified parity game locally, but instead of finding the winner of a single vertex we are finding the winners for a set of vertices, specifically $\mathfrak{C} \times \{\hat{v}_0\}$.

When we locally solve a parity game using the recursive algorithm we can sometimes avoid the second recursion because we already found the winner of $\hat{v}_0$. However, when locally solving a unified parity game we might find $\hat{v}_0$ for some configuration but not for all. 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, i.e. 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 returns the configurations for which we have found the local solution.

The algorithm uses definitions to reason about projections of unified parity games and sets to configuration(s). Previously we introduced a simple projection definition that projects a unified parity game to a configuration (Definition 5.2). This is possible because vertices in a unified parity game 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 7.2. *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.

Definition 7.3. *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*

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

We prove the correctness of Algorithm 9 by showing that every projection of the unified parity game is either solved globally or locally. We first prove the following auxiliary lemma to reason about projections.

Lemma 7.7. *Given unified parity game $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$, configuration $c \in \mathfrak{C}$ and non-empty set $X \subseteq (\mathfrak{C} \times \hat{V})$ such that $X_{|c} \neq \emptyset$, it holds that α-Attr$(G, X)_{|c} = \alpha$-Attr$(G_{|c}, X_{|c})$.*

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

if $V = \emptyset$ **then**
    return $(\emptyset, \emptyset)$
end if
$h \leftarrow \max\{\hat{\Omega}(\hat{v}) \mid (c, \hat{v}) \in V\}$
$\alpha \leftarrow 0$ 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
    $B \leftarrow \overline{\alpha}\text{-}Attr(G, W'_{\overline{\alpha}})$
    $C_B \leftarrow$ LocalConfs(B)
    $(W''_0, W''_1) \leftarrow$ RecursiveUPGLocal$(G \backslash B \backslash (V_{||C_B}), \hat{v}_0, \Delta)$
    $W_\alpha \leftarrow W''_\alpha$
    $W_{\overline{\alpha}} \leftarrow W''_{\overline{\alpha}} \cup B$
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

Proof. This lemma follows immediately from the fact that a unified parity game is the union of its projections. Furthermore, edges in unified parity games do not cross configurations, i.e. for any $((c, v), (c', v')) \in E$ we get $c = c'$. $\square$

Next we prove the correctness. We prove that either the projection onto a configuration is solved globally by the algorithm or in the projection $\hat{v}_0$ is found to be winning for player $\beta \in \Delta$. This is very similar to the local recursive algorithm for parity games (Algorithm 8 and Theorem 7.6) where we proved similar properties.

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

at least one of the following statements hold:

(I) For some $\beta \in \Delta$ *we have* $(c, \hat{v}_0) \in W_\beta$, $(W_0)_{|c} \subseteq Q_0$ *and* $(W_1)_{|c} \subseteq Q_1$.

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

Proof. Proof by induction on G.

Base: If G is empty then the algorithm returns $(\emptyset, \emptyset)$ in which case statement (II) holds trivially.

Step: When $G_{|c}$ is empty then $(W_0)_{|c} = \emptyset$ and $(W_1)_{|c} = \emptyset$ because the algorithm only returns vertices in the winning sets that are in V. In this case statement (II) holds trivially. Assume for the remainder of the proof that $G_{|c}$ is not empty, that is $V_{|c} \neq \emptyset$.

The algorithm considers the highest priority in the game and assigns the parity of this priority to α. The set U contains all vertices with this priority and A contains all vertices from where player α can force the play into U.

The first recursion removes vertices in A from the game. Since A is non-empty we can apply induction to find that at least one of the two statements hold for $G \backslash A$ and winning sets (W'_0, W'_1).

If $W'_{\overline{\alpha}} = \emptyset$ (line 13) then no vertex is won by player $\overline{\alpha}$ in $G \backslash A$ and therefore no vertex in $(G \backslash A)_{|c}$ is won by player $\overline{\alpha}$. Therefore statement (II) holds for $G \backslash A$ and indeed all vertices in $(G \backslash A)_{|c}$ are won by player α, moreover player α has a strategy σ_α for $(G \backslash A)_{|c}$ that is winning for all vertices. Clearly this strategy can also be applied in game $G_{|c}$. Consider valid path π in game $G_{|c}$ conforming to σ_α. When this path eventually stays in $(V \backslash A)_{|c}$ then player α wins because σ_α is winning here. Otherwise the path visits $A_{|c}$ infinitely often, in which case player α can force the play into $U_{|c}$ infinitely often and therefore the highest priority occurring infinitely often has parity α. So player α wins all vertices in $V_{|c}$ and the algorithm returns winning sets accordingly; statement (II) holds.

Otherwise the algorithm continues with calculating set B (line 17) and C_B (line 18).

For the remainder of the proof numerous case distinction need to be made. These distinctions will be presented in a Fitch-like style to improve readability.

First we distinguish two cases for $A_{|c}$.

> Assume $A_{|c} = \emptyset$
>
> ---
>
> Clearly $G_{|c} = (G \backslash A)_{|c}$, so all vertices in $(W'_{\overline{\alpha}})_{|c}$ are won by player $\overline{\alpha}$ in game $G_{|c}$.

> Assume $A_{|c} \neq \emptyset$
>
> ---
>
> We use Lemma's 7.7, Lemma 7.3 and induction on $G \backslash A$ to find that the vertices in $(W'_{\overline{\alpha}})_{|c}$ are won by player $\overline{\alpha}$ in $G_{|c}$.

In either case we find that all vertices in $(W'_{\overline{\alpha}})_{|c}$ are won by player $\overline{\alpha}$ in $G_{|c}$.

> Assume $c \in C_B$
>
> ---
>
> The second subgame that is created by the algorithm (line 19) does not contain any vertices with configuration c because $V_{||C_B}$ is removed from the game. Therefore W''_0 and W''_1 do not contain any vertices with configuration c. We find that the only vertices with configuration c that are returned by the algorithm are in the set B.
> For c to be in C_B we must have $(W'_{\overline{\alpha}})_{|c} \neq \emptyset$. We can apply Lemma 7.7 to find that set $B_{|c}$ contains all vertices such that player $\overline{\alpha}$ can force the play to $(W'_{\overline{\alpha}})_{|c}$ in game $G_{|c}$. Earlier we found that all vertices in $(W'_{\overline{\alpha}})_{|c}$ are won by player $\overline{\alpha}$ in game $G_{|c}$, so clearly all vertices in $B_{|c}$ are also won by player $\overline{\alpha}$.
> The algorithm returns vertices $B_{|c}$ to be winning for player $\overline{\alpha}$. Because $c \in C_B$ we find that $\overline{\alpha} \in \Delta$ and $\hat{v}_0 \in B_{|c}$. We conclude that statement (I) holds.

> Assume $c \notin C_B$
>
> ---
>
> If statement (I) holds for $G \backslash A$ then we would have found $\overline{\alpha} \in \Delta$ and $\hat{v}_0 \in (W'_{\overline{\alpha}})_{|c}$. Because $(W'_{\overline{\alpha}})_{|c} \subseteq B_{|c}$ we would also have found $c \in C_B$. Since this is not the case we find that statement (II) holds for $G \backslash A$.
>
> > Assume $(W'_{\overline{\alpha}})_{|c} = \emptyset$
> >
> > ---
> >
> > In this case $B_{|c} = \emptyset$ and the second subgame G' created (line 19) projected onto c is identical to $G_{|c}$. Using induction we find that statement (I) or (II) hold for G'. The algorithm returns W''_0 and W''_1 for game G' so the same statement that holds for the subgame holds for G. Note that B does not contain vertices with configuration c.

Assume $(W'_{\overline{\alpha}})_{|c} \neq \emptyset$

We apply Lemma 7.7 to find that $B_{|c} = \overline{\alpha}\text{-}Attr(G_{|c}, (W'_{\overline{\alpha}})_{|c})$. For the second subgame G' created (line 19) we have $(G')_{|c} = G_{|c} \backslash B_{|c}$ because $V_{||C_B}$ contains no vertices with configuration c.
The algorithm returns any vertex $\hat{v}$ in $(W''_{\beta})_{|c}$ to be winning for player β in game $G_{|c}$. Using induction and Lemma 7.5 we find that indeed $\hat{v}$ is won by player β in game $G_{|c}$. Furthermore the algorithm returns $B_{|c}$ to be winning for player $\overline{\alpha}$. Earlier we found that all vertices in $(W'_{\overline{\alpha}})_{|c}$ are won by player $\overline{\alpha}$ in game $G_{|c}$, so clearly all vertices in $B_{|c}$ are also won by player $\overline{\alpha}$.
The vertices with configuration c that are returned by the algorithm are in the correct winning set. If statement (I) holds for the subgame G' then statement (I) also holds for G because we use Δ unmodified in the recursion. If statement (II) holds for the subgame G' then statement (II) also holds for G.

□

The pseudo code presented for the algorithm uses a set-wise representation of unified parity games. 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 a definition for projecting to sets of configurations as well as the LocalConfs subroutine. We introduce function-wise variants for this definition and subroutine.

Definition 7.4. *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*

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

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

Algorithm 10 Function-wise LocalConfs subroutine

function FLocalConfs($X : \hat{V} \to 2^{\mathfrak{C}}$)
  if $\overline{\alpha} \in \Delta$ **then**
    return $X(\hat{v}_0)$
  else
    return $\emptyset$
  end if
end function

We can solve a VPG locally using this local recursive algorithm for unified parity games. We can either represent the parity games set-wise or we can represent them function-wise, in which case we can represent the sets of configurations explicitly or symbolically. In all three cases the time complexities are identical to their global counterparts because in the worst case the vertex we are searching for is never won by player $\overline{\alpha} \in \Delta$ at any recursion level. Furthermore the added projection operation and the LocalConfs subroutine are subsumed in worst-case time complexity by the attractor set calculation. Therefore the worst-case 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; 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 a single configuration left the incremental pre-solve algorithm solves the corresponding parity game. When taking a local approach it is sufficient to solve this parity game locally using the local fixed-point iteration algorithm. Note that the pessimistic games still must be solved globally to find as much assistance as possible for further recursions.

If v_0 is not found in either P_0 or P_1 and is only solved when there is one configuration left, then the local algorithm behaves the same as the global algorithm; we have identical worst-case time complexities: $O(c * e * n^d)$.

8. Experimental evaluation

The algorithms proposed to solve VPGs collectively all 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 does not say much about the performance in case there are many commonalities. In order to evaluate actual running time the algorithms are implemented and a number of test VPGs are created to test the performance on. In this section we discuss the implementation and look at the results.

During the previous sections we put forth a number of hypotheses about the performance of the algorithms introduced. In this section we evaluate these hypotheses, specifically we hypothesised:

- that the recursive algorithm for VPGs can attract a large number of configurations per origin vertex at the same time,
- that the recursive symbolic algorithm for VPGs performs well when solving VPGs originating from FTSs,
- that the incremental pre-solve algorithm outperforms independent approaches and,
- that the increase in performance between a global-collective and local-collective approach is greater than the increase in performance between a global-independent and local-independent approach.

8.1 Implementation

The algorithms are implemented in C++ version 14 and use BuDDy[1] [27] 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 and their running times are not considered in the experimental evaluation.

The parsing phase of the algorithm creates BDDs from the input file and 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. Creating BDDs is not a trivial task, however one could argue that an FTS should already express its transition guards as BDDs. In any case, we leave the creation of BDDs out of scope.

8.1.1 Game representation

The graph is represented using adjacency lists for incoming and outgoing edges, furthermore every edge maps to a set of configurations representing the θ value for the edge. Sets of configurations are either represented symbolically or explicitly. In the former case we use BDDs, in the latter case we use bit-vectors. For independent algorithms the edges are not mapped to sets of configurations. Finally sets of vertices are represented using bit-vectors.

[1] https://sourceforge.net/projects/buddy/
[2] https://github.com/SjefvanLoo/VariabilityParityGames/tree/master/implementation/VPGSolver

Note that only the representation of the games used during the algorithm is relevant. Since we do not evaluate the parsing phase it is not relevant how the games are stored in a file.

8.1.2 Independent algorithms

Four independent algorithms are implemented, i.e. standard parity game algorithms. A global and local variant is implemented of the following algorithms:

- Zielonka's recursive algorithm and
- fixed-point iteration algorithm.

We implement the fixed-point iteration algorithm to use pre-solved vertices P_0 and P_1. When using the algorithm for an independent approach we use $\emptyset$ for P_0 and P_1, in which case the algorithm behaves the same as the original fixed-point iteration algorithm.

A few optimizations are applied to the fixed-point iteration algorithm. The following three are described in [4]:

- 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 updates 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 updated.
- 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 formulas, i.e. $Z_{m-2}, Z_{m-4}, \ldots$, 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.

Furthermore, 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. 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 iterating all the different vertices is due to the fact that a bit vector uses integers to store its boolean values. A 64-bit integer can store 64 boolean values. Iterating and writing every boolean value individually causes the integer to be written 64 times. However with a memory copy we can simply copy the entire integer value and the integer is only written once.

Finally, priority compression is applied when using the fixed-point iteration algorithm. Priority compression makes sure the lowest priority is 0 or 1 and for every priority p that is lower or equal to the highest priority occurring in the game we have p being either the lowest priority in the game or there is a vertex in the game with priority $p-1$ [4, 17].

8.1.3 Collective algorithms

Six collective algorithms are implemented, i.e. algorithms for solving VPGs. A global and local variant is implemented of the following algorithms:

- Zielonka's recursive algorithm for VPGs with explicit configuration set representation,
- Zielonka's recursive algorithm for VPGs with symbolic configuration set representation and
- incremental pre-solve algorithm.

The incremental pre-solve algorithms uses the fixed-point iteration algorithm as described above to solve the (pessimistic) parity games. When using the incremental pre-solve algorithm we apply priority compression once, directly on the VPG. Since the (pessimistic) parity games that are created during the algorithm have the same vertices as the VPG we do not have to apply priority compression again when using the fixed-point iteration algorithm to solve them.

The incremental pre-solve algorithm creates subgames by splitting the set of configurations. The games we evaluate are based on features so we simply split the set of configurations by arbitrarily choosing a feature and including this feature in one set of configurations and excluding this feature in the other set of configurations. The problem of finding good heuristics to split the sets of configurations is also described in [37]. We leave this problem out of scope for this research.

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 parity games. These parity games are solved using the PGSolver tool [20]. All algorithms implemented are used to solve the 200 VPGs independently and collectively, the solutions are verified against the solutions created by the PGSolver.

8.2 Test cases

We evaluate the performance of the algorithms on numerous test cases. We have two SPL model checking problems as well as random VPGs. The model checking VPGs are created as described in chapter 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.

In this section we present the different test cases and their characteristics. In the next section the running times are presented.

8.2.1 Model checking games

We use two SPL examples. First, the minepump example as described in [25] and implemented in the mCRL2 toolset [6] as described in [37]. 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, i.e. products.

The mCRL2 implementation creates an LTS with parametrized actions where the parameters describe the boolean formulas guarding the transitions, effectively making it an FTS consisting of 582 states and 1376 transitions. This FTS is interpreted in combination with nine different μ-calculus formulas to create nine VPGs.

We choose to represent the sets of configurations using 10 boolean variables even though 128 configurations could be represented using only 7 boolean variables. By using the same number of variables as there are features the boolean formulas from the FTS are left intact when using them in the VPG. Table 8.1 shows the different formulas, as well as the result of the verification and the size of the resulting games. All the properties can be expressed in the modal μ-calculus we introduced in Definition 3.10. However, for readability, we present them using action formulas, regular formulas and universal quantifiers [22].

Next we have the elevator example, described in [30]. This example models the behaviour of an elevator where five different features modify the behaviour of the model. All feature assignments are valid. Therefore, we have $2^5 = 32$ feature assignments, i.e. products. Again an mCRL2 implementation[3] (created by T.A.C. Willemse) is used to create seven VPGs. The FTS consists of 33738 states and 206290 transitions. Table 8.2 shows the different formulas, as well as the result of the verification and the size of the resulting games.

8.2.2 Random games

We create a set of random VPGs such that some games are very similar to the VPGs originating from the SPL verification problems and some games are very different. We use these games to further evaluate the performance of the algorithms.

The guard sets in the minepump and elevator games have a very specific distribution where nearly all of the sets admit either 100% or 50% of the configurations. This is because an edge requiring the presence or absence of one specific feature results in a set admitting 50%. On average the edges in the examples admit 92% of the configurations. Most likely VPGs originating from FTSs will have such a distribution.

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 constructed.

We use λ to denote the average relative size of guard sets in a VPG. So for every guard set in a VPG we divide its size by the total number of configurations to get the relative size of the guard set. Taking the average of all these relative sizes calculates λ.

For every random game we create, we pick a specific λ. This allows us to create games that have a λ similar to those observed in the minepump and elevator example, i.e. $\lambda = 0.92$, and games that have a λ very different from the SPL games.

Once we decided a value for λ we need to decide the sizes of the individual guard set. We do so by using a probabilistic distribution ranging from 0 to 1 with a mean equal to λ. We consider two distributions, namely a modified Bernoulli distribution which creates guard sets of only relative

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

	formula	**t/f**	n	d
φ_1	*Absence of deadlock* $[\mathbf{true}^*]\langle\mathbf{true}\rangle\top$	128/0	3494	2
φ_2	*The controller cannot infinitely often receive water level readings* $\mu X.[(\neg levelMsg)^*.levelMsg]X$	0/128	3004	3
φ_3	*The controller cannot fairly receive each of the three message types* $\mu X.([\mathbf{true}^*.commandMsg]X \vee [\mathbf{true}^*.alarmMsg]X \vee$ $[\mathbf{true}^*.levelMsg]X)$	0/128	9156	3
φ_4	*The pump cannot be switched on infinitely often* $(\mu X.\nu Y.([pumpStart.(\neg pumpStop)^*.pumpStop]X \wedge$ $[\neg pumpStart]Y)) \wedge ([\mathbf{true}^*.pumpStart]\mu Z.[\neg pumpStop]Z)$	96/32	6236	4
φ_5	*The system cannot be in a situation in which the pump runs indefinitely in the presence of methane* $[\mathbf{true}^*](([pumpStart.(\neg pumpStop)^*.methaneRise]\mu X.[R]X) \wedge$ $([methaneRise.(\neg methaneLower)^*.pumpStart]\mu X.[R]X))$ for $R = \neg(pumpStop + methaneLower)$	96/32	7096	3
φ_6	*Assuming fairness (φ_3), the system cannot be in a situation in which the pump runs indefinitely in the presence of methane (φ_5)* $[\mathbf{true}^*](([pumpStart.(\neg pumpStop)^*.methaneRise]\Psi) \wedge$ $([methaneRise.(\neg methaneLower)^*.pumpStart]\Psi))$ for $\Psi = \mu X.([R^*.commandMsg]X \vee [R^*.alarmMsg]X \vee [R^*.levelMsg]X)$ and $R = \neg(pumpStop + methaneLower)$	112/16	9224	4
φ_7	*The controller can always eventually receive/read a message, i.e. it can return to its initial state from any state* $[\mathbf{true}^*]\langle\mathbf{true}^*.receiveMsg\rangle\top$	128/0	5285	3
φ_8	*Invariantly the pump is not started when the low water level signal fires* $[\mathbf{true}^*.lowLevel.(\neg(normalLevel + highLevel))^*.pumpStart]\bot$	128/0	3902	2
φ_9	*Invariantly, when the level of methane rises, it inevitably decreases* $[\mathbf{true}^*.methaneRise]\mu X.[\neg methaneLower]X \wedge \langle\mathbf{true}\rangle\top$	0/128	5418	3

Table 8.1: Minepump properties with their partitioning and the size of the resulting VPG. In the **t/f** columns the first number shows for how many products the property holds. Columns n and d shows the number of vertices and distinct priorities in the resulting VPG. The formula column is taken verbatim from [37]

.

	formula	**t/f**	n	d
φ_1	*If a landing button is pressed at Level i, the lift will inevitably open its doors on Level i* $[\mathbf{true}^*]\forall i \in [1,5].[\mathit{landingButton}(i)](\mu X.([\neg \mathit{open}(i)]X \land \langle \mathbf{true} \rangle \top)$ $)$	2/30	1379959	3
φ_2	*If a lift button is pressed at Level i, the lift will inevitably open its doors on Level i* $[\mathbf{true}^*]\forall i \in [1,5].[\mathit{liftButton}(i)](\mu X.([\neg \mathit{open}(i)]X \land \langle \mathbf{true} \rangle \top))$	4/28	1381390	3
φ_3	*If the lift is travelling up while there are calls in that direction it will not change the direction it is travelling* $[\mathbf{true}^*]($ $[\mathit{direction}(\mathit{up}).(\neg(\mathit{direction}(\mathit{down}) + \exists k \in [1,5].\mathit{open}(k)))^*]$ $\forall i \in [1,5].[\mathit{open}(i)]\forall j \in [i+1,5].$ $[\mathit{liftButton}(j)]\mu Y.($ $[\neg \mathit{open}(j)]Y \land [\mathit{direction}(\mathit{down})]\mathbf{false} \land \langle \mathbf{true} \rangle \top))$	4/28	1778065	3
φ_4	*If the lift is travelling down while there are calls in that direction it will not change the direction it is travelling* $[\mathbf{true}^*]($ $[\mathit{direction}(\mathit{down}).(\neg(\mathit{direction}(\mathit{up}) + \exists k \in [1,5].\mathit{open}(k)))^*]$ $\forall i :\in [1,5].[\mathit{open}(i)]\forall j \in [1,i-1].$ $[\mathit{liftButton}(j)]\mu Y.($ $[\neg \mathit{open}(j)]Y \land [\mathit{direction}(\mathit{up})]\mathbf{false} \land \langle \mathbf{true} \rangle \top))$	4/28	1853633	3
φ_5	*If the lift is idling on Level i, it can remain at Level i* $(\forall i \in [1,5].\langle \mathbf{true} * .\mathit{idling}(i) \rangle \top) \land$ $[\mathbf{true}^*]\forall i \in [1,5].[\mathit{idling}(i)]\nu Y.\langle \mathit{idling}(i) \rangle Y$	16/16	1282147	2
φ_6	*The lift may stop at Levels 2,3 and 4 for landing calls when travelling upwards* $\forall i \in [2,4].(\langle (\neg \mathit{liftButton}(i))^*.\mathit{direction}(\mathit{up}).$ $(\neg(\mathit{liftButton}(i) + \mathit{direction}(\mathit{down})))^*.\mathit{open}(i) \rangle \top)$	32/0	443352	2
φ_7	*The lift may stop at Levels 2,3 and 4 for landing calls when travelling downwards* $\forall i \in [2,4].(\langle (\neg \mathit{liftButton}(i))^*.\mathit{direction}(\mathit{down}).$ $(\neg(\mathit{liftButton}(i) + \mathit{direction}(\mathit{up})))^*.\mathit{open}(i) \rangle \top)$	32/0	443012	2

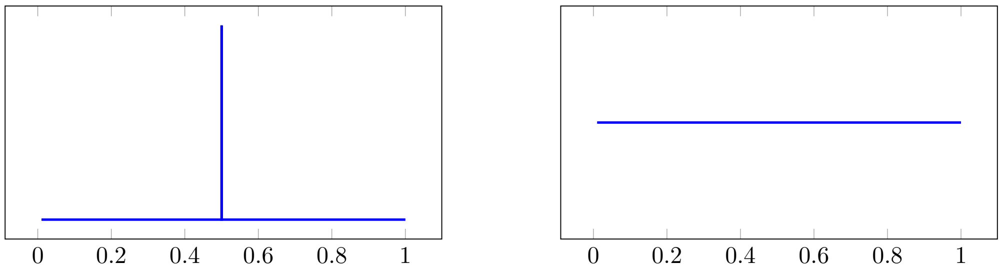
Table 8.2: Elevator properties with their partitioning and the size of the resulting VPG. In the **t/f** columns the first number shows for how many products the property holds. Columns n and d shows the number of vertices and distinct priorities in the resulting VPG.

size 0.5 and 1 and a beta distribution which creates guard sets with a more varying range of relative sizes. We can use the former to create random games that are similar to the SPL games and use the latter to create random games that are different to the SPL games.

- 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 cannot use this distribution when $\lambda < 0.5$ because p becomes less than 0.

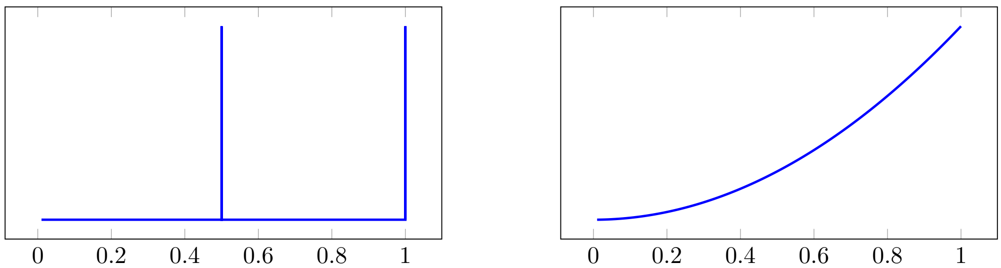
- A beta distribution; a beta distribution ranges from 0 to 1 and is curved such that it has a specific mean. The beta distribution has two parameters: α and β and a mean of $\frac{\alpha}{\alpha+\beta}$. We pick $\beta = 1$ and $\alpha = \frac{\lambda\beta}{1-\lambda}$ to get a mean of λ.

Figures 8.1, 8.2 and 8.3 show the shapes of the distribution for different values for λ.



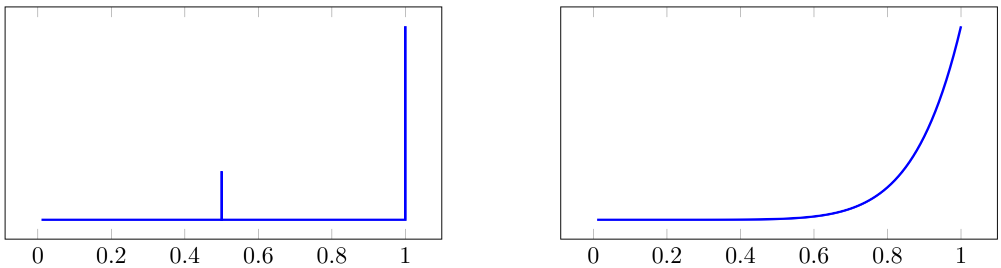
(a) Modified Bernoulli distribution with $p = 0$ **(b)** Beta distribution with $\beta = 1$ and $\alpha = 1$

Figure 8.1: 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 8.2: 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 8.3: Edge guard size distribution for $\lambda = 0.9$

Consider the creation of a random game that has 2^m configurations, some λ is decided upon and one of the above distribution is chosen. For an individual guard set we use the random distribution to decide how large this guard set should be. Consider the creation of a specific guard set and let r denote the relative size of this guard set, decided using the random distribution we chose.

We now need to consider how to create a guard set of relative size r. We can simply create a random set of configurations without any notion of features; we call this a *configuration based* approach. Using this approach we can easily create a guard set of relative size r by simply picking $\lfloor 2^m * r \rfloor$ configurations randomly.

Alternatively we can use a *feature based* approach where we create sets by looking at features. Consider features $f_0, \dots, f_m$, we can create a boolean function that is the conjunction of k features where every feature in the conjunction has probability $\frac{1}{2}$ 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 $\frac{2^{m-k}}{2^m} = 2^{-k}$. Since we are creating a set of 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 2^{-i} for some $i \in \mathbb{N}$.

When creating a random game for some λ we have considered how we can choose the size of individual set sizes and how to construct sets of that size. This gives us four different ways to construct games:

1. Bernoulli distributed and feature based. These games are most similar to the SPL games.
2. Bernoulli distributed and configuration based. These games do have the characteristics of an SPL game in terms of set size but have unstructured sets guarding the edges. Furthermore with a configuration based approach fewer guard sets will be identical than with a feature based approach.
3. Beta distributed and configuration based. These games are most different from the SPL games.
4. Beta distributed and feature based. Using a feature based approach we can only create sets of size 2^{-i} for any $\lambda \geq \frac{1}{2}$. So using a beta distribution we must round to such a size. Almost all the sets will get a relative size of either $\frac{1}{2}$ or 1. So this creates almost the exact same games as using the Bernoulli distribution, therefore we will not consider this category of games.

<table>
<tr><th>Category</th><th># vertices</th><th>Maximum
successors</th><th># distinct
priorities</th><th># confs</th><th>λ</th></tr>
<tr><td>Type 1, scale in λ</td><td rowspan="3">$100 - 600$</td><td rowspan="3">$3 - 20$</td><td rowspan="3">$1 - 10$</td><td rowspan="3">$2^4 - 2^{12}$</td><td rowspan="3">$\frac{game\ nr}{100}$</td></tr>
<tr><td>Type 2, scale in λ</td></tr>
<tr><td>Type 3, scale in λ</td></tr>
<tr><td>Type 1, scale in # confs</td><td>$100 - 600$</td><td>$3 - 20$</td><td>$1 - 10$</td><td>$2^{game\ nr}$</td><td>0.92</td></tr>
</table>

Table 8.3: Categories of random games

We create four sets of random games. For random games of type 1,2 and 3 we create 25 games: game 75 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 52 games to evaluate how the algorithm scales when the number of features becomes larger. For every $i \in [2, 15]$ we create random games i, $i.25$, $i.50$ and $i.75$ of type 1 with $\lambda = 0.92$, $\lfloor i \rfloor$ features and a random number of nodes, edges and maximum priority.

Besides the number of configurations and the value for λ we need to choose the number of vertices for a game, the minimum number of successors of a vertex, the maximum number of successors of a vertex and the number of distinct priorities in the game. The number of minimum and maximum successor is decided per game. So if we pick l and h as the number of minimum and maximum then for every vertex in the game we uniformly pick its number of successors between l and h.

Table 8.3 shows the different categories of games and the corresponding parameters. The minimum number of successors per vertex is always 1 so this value is omitted from the table. The games that scale in λ share the same random configuration per game number. So game i of type 1 that scales in λ has the same number of vertices, maximum successors, distinct priorities and configurations as game i of type 2 and 3 that scale in λ.

8.3 Results

In this section the experimental results are presented.[4] We evaluate the performance on six sets of games:

- the minepump games,
- the elevator games,
- random games of type 1 with an increasing λ,
- random games of type 2 with an increasing λ,
- random games of type 3 with an increasing λ, and
- random games of type 1 with an increasing number of configuration.

We present the times it took to solve a VPG. For an independent approach this means the sum of the times it takes to solve every projection of the VPG. For a collective approach this

[4] The complete collection of test VPGs can be found at:
https://github.com/SjefvanLoo/VariabilityParityGames/tree/master/implementation

simply means the solve time for the VPG. In either case we only measure the solve time; parsing, projecting and solution printing is excluded from the evaluation.

The exact times can be found in appendix A; in this section the results are visualized and presented in a way such that we can easily compare independent and collective approaches. We have four independent approaches:

- Recursive algorithm (global),
- Recursive algorithm (local),
- fixed-point iteration (global) and
- fixed-point iteration (local).

For every set of problems we present four charts; for every independent approaches we present a chart where its performance is compared to one or two collective algorithms. We compare the performance of the recursive algorithm for VPGs with the performance of the original recursive algorithm and the performance of the incremental pre-solve algorithm with the performance of the fixed-point iteration algorithm. In some of the charts the solve times are divided by the independent solve times to visualize how much better or worse the collective variants perform.

The following legend holds for all charts presented in this section:

Independent approaches:

Recursive algorithm for parity games (global)
Fixed-point iteration algorithm for parity games (global)
Recursive algorithm for parity games (local)
Fixed-point iteration algorithm for parity games (local)

Collective approaches:

Recursive algorithm for VPGs with a symbolic representation of configurations (global)
Recursive algorithm for VPGs with an explicit representation of configurations (global)
Incremental pre-solve algorithm (global)
Recursive algorithm for VPGs with a symbolic representation of configurations (local)
Recursive algorithm for VPGs with an explicit representation of configurations (local)
Incremental pre-solve algorithm (local)

All the experiments are ran on a Linux x64 operating system with an Intel i5-4570 @ 3.20 GHz processor and 8GB of DDR3 RAM.

8.3.1 SPL examples

Figures 8.4 and 8.5 show the solving times (in ms) of the algorithms when applied to the SPL examples.

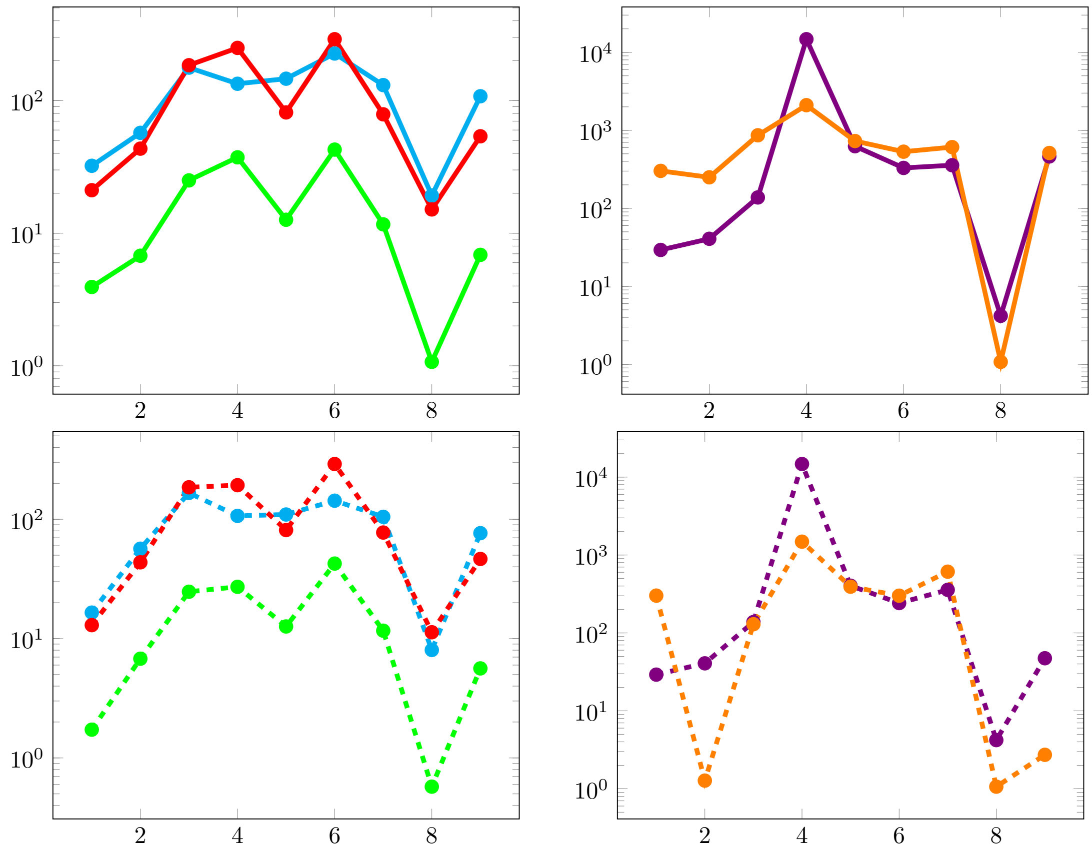


Figure 8.4: Running times on the minepump games. The x-axis shows the game numbers, these correspond with the formulas described in Table 8.1. The y-axis shows, on a logarithmic scale, the number of milliseconds required to solve the VPG.

For the minepump example we see that the recursive algorithm for VPGs using a symbolic representation performs particularly well; about a 3 to 18 times increase in performance compared to the independent approach. For the elevator example we also find an increase in performance for the symbolic recursive algorithm compared to the independent algorithm; about a 2 to 6 times increase. The difference is most likely because the minepump games have twice as many features as the elevator games have. Notably, for the elevator games we also find a good performance for the incremental pre-solve algorithm, which we do not find for the minepump games. Finally, we observe that there is no clear difference between the relative performances of the global algorithms and the local algorithms.

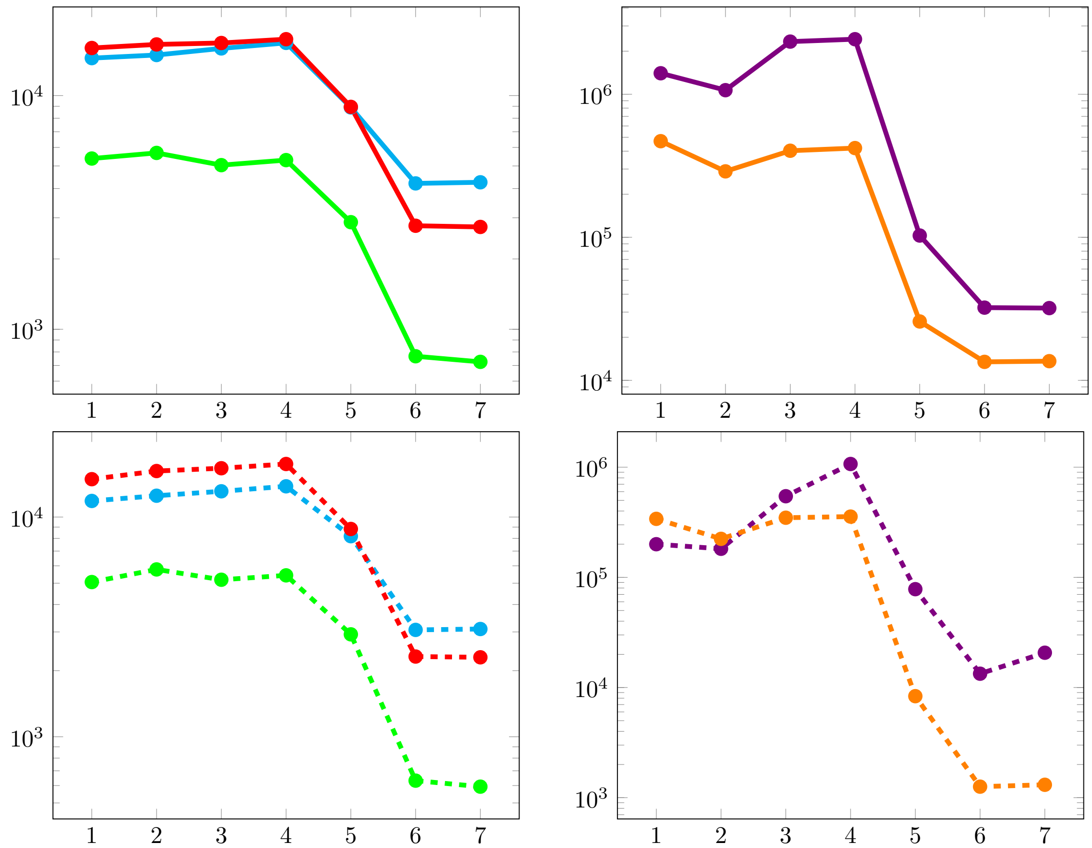


Figure 8.5: Running times on the elevator games. The x-axis shows the game numbers, these correspond with the formulas described in Table 8.2. The y-axis shows, on a logarithmic scale, the number of milliseconds required to solve the VPG.

8.3.2 Random games

Figure 8.6 shows the performance of the algorithms on type 1 games. The recursive algorithms for VPGs perform quite well, even though there are a few instances where the performance is worse than the independent approach. The symbolic variant performs quite a bit better than the explicit variant. The relative performance of the local variants of the recursive algorithms is about the same as the relative performance of the global variants.

For the incremental pre-solve algorithm we do see a big difference between a local and global approach. The global variant performs well only for games 90 and up. The local variant, however, performs well for nearly all the games. Furthermore, even for the games where the global variant performs well relative to the independent global approach does the local variant performs even better relative to the independent local approach.

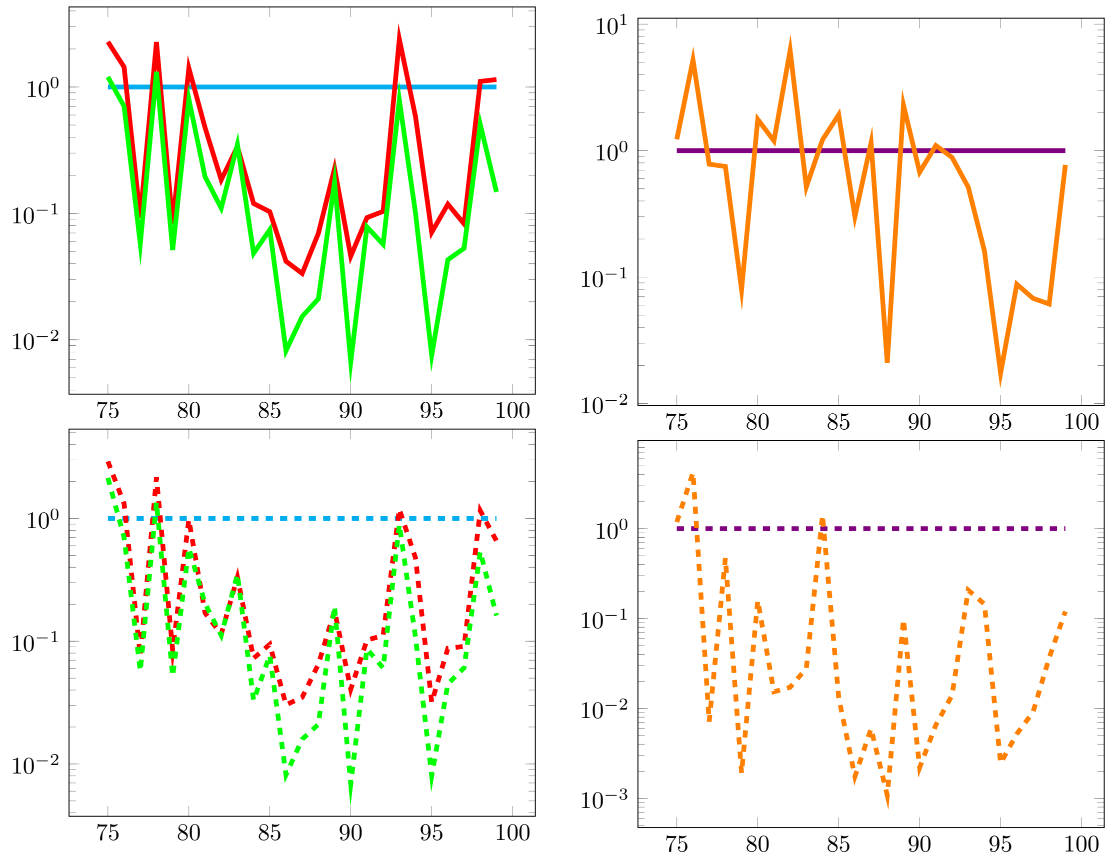


Figure 8.6: Running times on random games of type 1 with $\lambda = \frac{game\ nr}{100}$. The x-axis shows the game numbers. The y-axis shows, on a logarithmic scale, how much faster the algorithms solved a VPG compared to the independent algorithm. Clearly the independent algorithm always has value of 1 for every VPG. If an algorithm has a value above 1 for a VPG then it performed worse than the independent algorithm; if the value is below 1 then it performed better than the independent algorithm.

Figure 8.7 shows the performance of the algorithms on type 2 games. For the recursive algorithms we see that the explicit variant takes over from the symbolic variant. This is to be expected since these games have edge guards that are not created from features but created by picking random configurations. This decreases the performance of symbolic set operations but has no effect the performance of explicit set operations. Both variants still perform somewhat better than the independent approach. Again we do not find a significant difference between the global and local approach.

For the incremental pre-solve algorithm we find a similar result as with type 1 games. The global variant performs well only when λ is high. The local variant performs significantly better and performs well for almost all games.

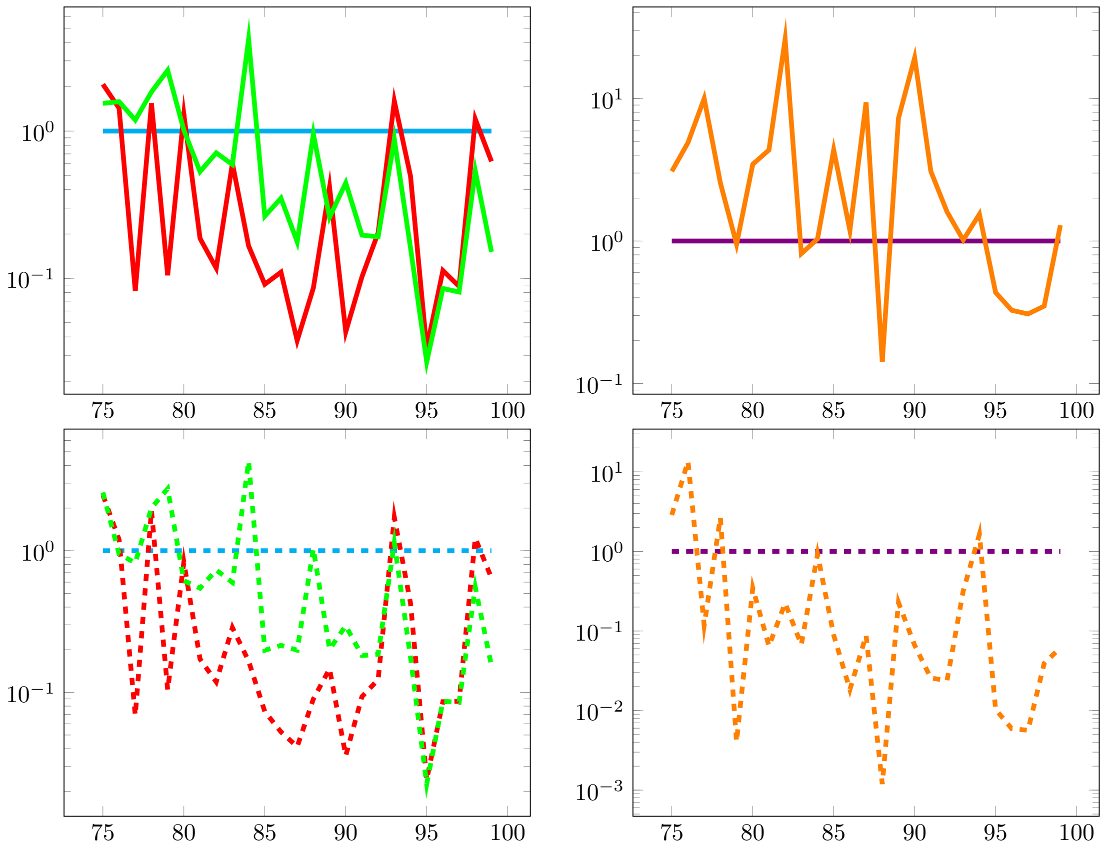


Figure 8.7: Running times on random games of type 2 with $\lambda = \frac{game\ nr}{100}$ The x-axis shows the game numbers. The y-axis shows, on a logarithmic scale, how much faster the algorithms solved a VPG compared to the independent algorithm.

Figure 8.8 shows the performance of the algorithms on type 3 games. For these games we see the symbolic variant of the recursive algorithm performing worse than the independent approach for almost all games. The explicit variant still performs significantly better than the symbolic variant and performs somewhat better than the independent approach. This is similar to type 2 games, which is to be expected because both types of games use configuration sets not based on features. Again we do not find a significant difference between the global and local approach.

The global incremental pre-solve algorithm performs worse than the independent approach for almost all games. Notably for these games we do not find a significant increase in relative performance when using a local variant.

Notably, the explicit recursive algorithm seems to be the only algorithm unaffected by the fact that the guard sets sizes of type 3 games vary wildly (they are distributed using a beta distribution). Maybe surprisingly, the incremental pre-solve algorithm is affected heavily by this. This is most likely because there are a lot fewer edges that admit all configurations and therefore player α will probably win fewer vertices in a pessimistic game for player α.

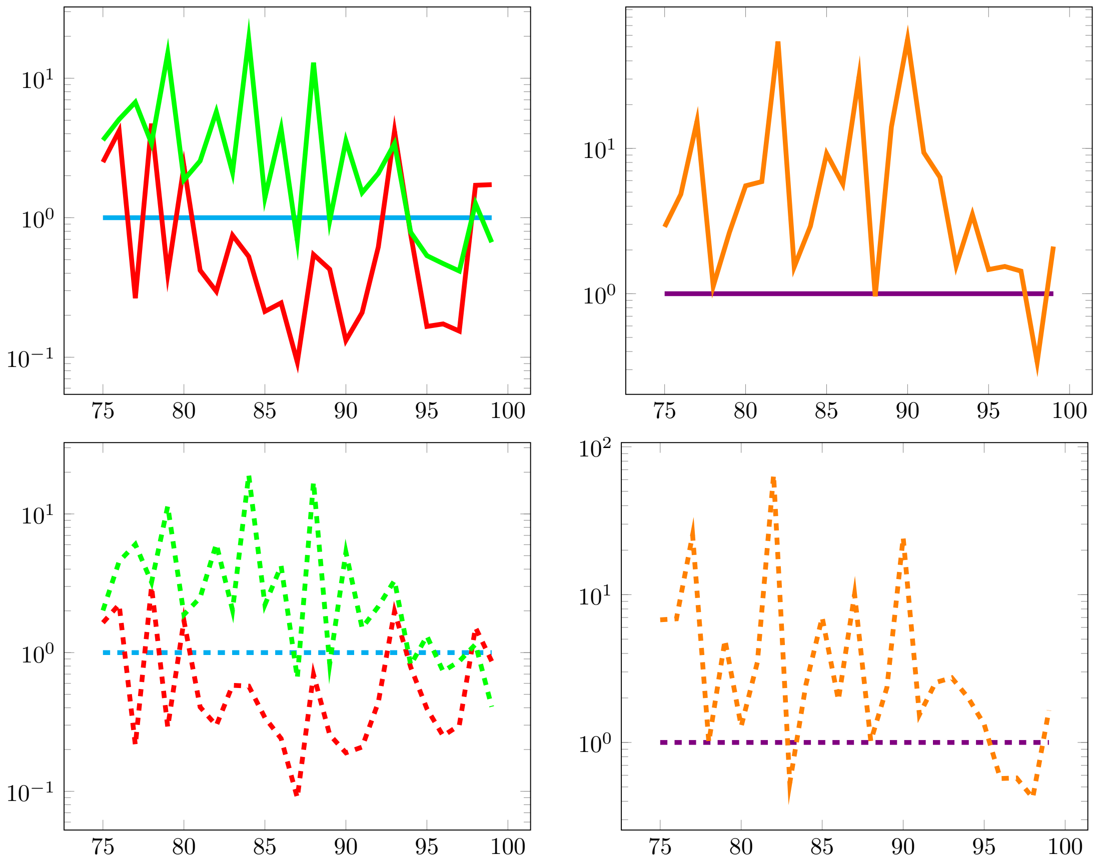


Figure 8.8: Running times on random games of type 3 with $\lambda = \frac{game\ nr}{100}$ The x-axis shows the game numbers. The y-axis shows, on a logarithmic scale, how much faster the algorithms solved a VPG compared to the independent algorithm.

8.3.3 Scaling

Figure 8.9 shows the performance of the algorithms on type 1 games where the number of configurations increase exponentially in the x-axis of the charts. For the recursive algorithm we see that the collective approach starts outperforming the independent approach around 2^4 configurations. As the number of configurations grow we see that the symbolic variant keeps increasing in relative performance while the explicit variants relative performance starts to flatten. This is to be expected because the performance of the explicit variant always scales linearly in the number of configurations. In the worst case the symbolic variant scales quadratically in the number of configurations, however when the sets of configurations can be represented efficiently it scales much better and in this case sublinear (since the performance of the local variant keeps increasing relative to the explicit variant).

The global incremental pre-solve algorithms does not increase notably in relative performance when the number of configurations increases. However, the relative performance of the local variant does increase in performance when the number of configurations increases. The recursion of the incremental pre-solve algorithm can be conceptualized as a tree where at every node the algorithm tries to increase the pre-solved vertices. The local variant can terminate when at some node the vertex that is being locally solved is found. In such a case the whole subtree of that node is longer computed. When the number of configurations grow then potentially the size of this subtree also grows. The fact that the local incremental pre-solve algorithm scales well in the number of configurations is most likely because the algorithm can terminate early for a relatively large set of configurations.

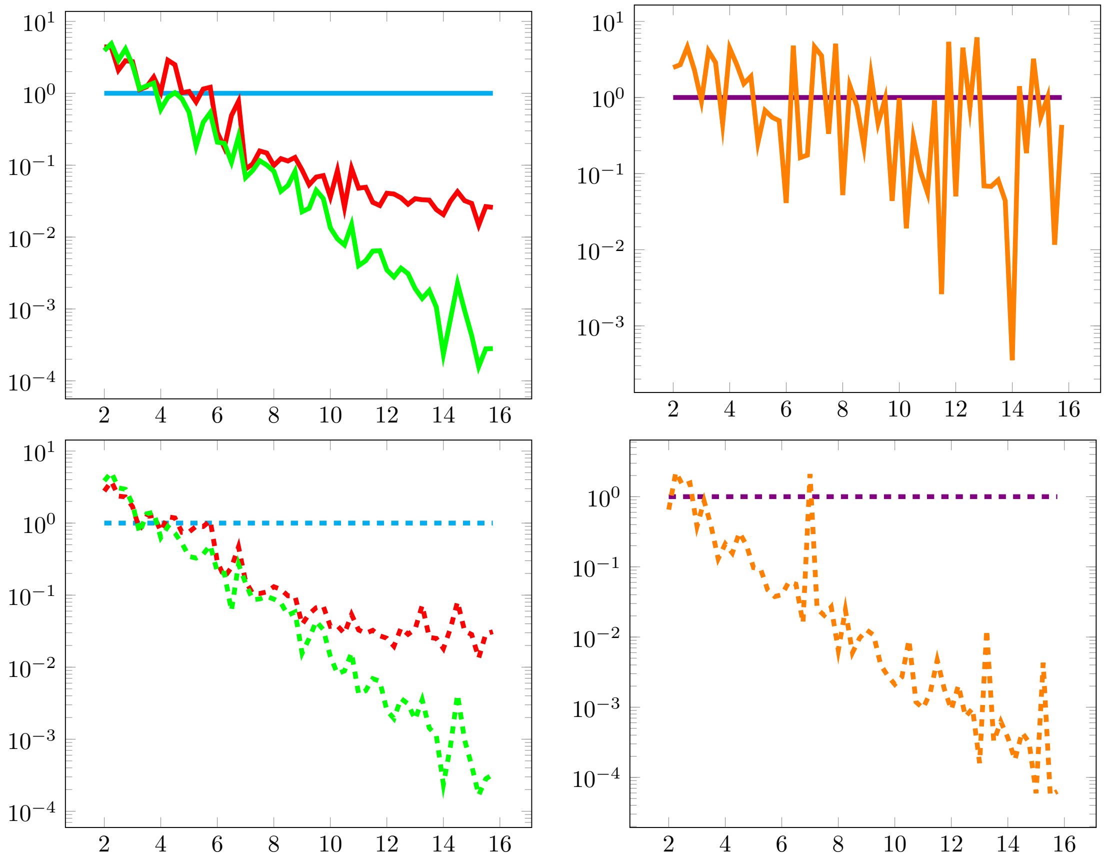


Figure 8.9: Running times on random games of type 1 with $\lambda = 0.92$ and the number of features equal to $\lfloor game\ nr \rfloor$. The x-axis shows the game numbers. The y-axis shows, on a logarithmic scale, how much faster the algorithms solved a VPG compared to the independent algorithm.

8.3.4 Internal metrics

Earlier we hypothesised that the recursive algorithm for VPGs could perform well if we can attract many configurations simultaneously. For every VPG we measure the average number of configurations that were attracted simultaneously. We measure this relative to the total number of configurations in the VPG. This gives a number for every VPG. For every set of VPGs we average this number to get an average set size for every problem set. These values indicate how many configurations were attracted simultaneously. In Table 8.4 these values are presented for the different problems being globally solved, note that whether the sets are represented explicitly or symbolically is irrelevant. We see that this number somewhat predicts the performance of the recursive algorithms.

Minepump	46%
Elevator	51%
Type 1, scaling in λ	61%
Type 2, scaling in λ	55%
Type 3, scaling in λ	22%
Type 1, scaling in # confs	72%

Table 8.4: Relative size of attracted sets

The incremental pre-solve algorithm tries to outperform its independent counterpart by growing the set of pre-solved vertices. We measure how many vertices are pre-solved for the different sets of problems. For every VPG we measure the average number of pre-solved vertices for every (pessimistic) parity game solved. We measure this relative to the number of vertices in the VPG. This gives a number for every VPG. For every set of VPGs we average this number to get an average vertex size for every problem set. These values indicate how many vertices were pre-solved on average. In Table 8.5 these values are presented for the different problems. The algorithm recurses into two branches after every two pessimistic parity games solved. The further we go down the tree the higher the number of pre-solved vertices. So the average numbers presented in the table are quite high because there are exponentially more parity games that are further down the recursion tree.

For the global variant the number presented in Table 8.5 somewhat predicts the performance of the incremental pre-solve algorithm compared to the fixed-point iteration algorithm. For the local variant this is not the case. The local variant performs well when parts of the recursion tree are not calculated because we have terminated early. However, if the recursion tree is less deep then the numbers in the table decrease. Therefore, for the local algorithm the numbers do not predict the performance.

	Global	Local
Minepump	58%	29%
Elevator	84%	38%
Type 1, scaling in λ	87%	9%
Type 2, scaling in λ	82%	9%
Type 3, scaling in λ	57%	19%
Type 1, scaling in # confs	91%	12%

Table 8.5: Relative number of pre-solved vertices

8.3.5 Discussion

From the experimental results we observe that the symbolic variant of the recursive algorithms for VPGs performs well for the model verification problems. For type 1 random games it also performs well and particularly scales very well in the number of configurations. For type 2 and 3 games the sets of configurations can no longer be efficiently represented symbolically and performance drops.

After comparing independent and collective approaches we compare the performances of the algorithms overall.

First we compare the independent algorithms. Table 8.6 shows for every set of VPGs how long it took each algorithm to solve all the VPGs in that set. We observe that the recursive algorithm

	Recursive global	Recursive local	Fixed-point global	Fixed-point local
Minepump	1032 ms	788 ms	16713 ms	15989 ms
Elevator	79468 ms	65588 ms	7393468 ms	2108539 ms
Type 1, scaling in λ	1830 ms	1730 ms	53677 ms	51506 ms
Type 2, scaling in λ	1936 ms	1721 ms	74710 ms	72536 ms
Type 3, scaling in λ	1892 ms	1680 ms	62568 ms	51231 ms
Type 1, scaling in # confs	32903 ms	29393 ms	439199 ms	274201 ms

Table 8.6: Comparison of independent algorithms. The times shown are the times it took an algorithm to solve all the VPGs in a problem set independently.

performs significantly better than the fixed-point algorithm across all problems. We also see that the local variants perform somewhat better across the board. In table 8.7 we compare the performance of the collective algorithms. We observe that the recursive symbolic variant performs the best for model-checking problems and for type 1 games. Furthermore, most likely the algorithm will scale well for models with a large number of features. The local variant of the incremental pre-solve algorithm also performs well relative to its independent counterpart. However, because the fixed-point iteration is heavily outperformed by the recursive algorithm its overall performance is worse that the recursive variants. On average, the global variant of the incremental pre-solve algorithm also outperforms its independent counterpart. However, we have seen in the comparison charts that it does so less consistently and significantly than the local variant and the symbolic recursive algorithm do.

	Recursive explicit global	Recursive explicit local	Recursive symbolic global	Recursive symbolic local	Incremental pre-solve global	Incremental pre-solve local
Minepump	1019 ms	942 ms	148 ms	133 ms	5900 ms	3223 ms
Elevator	81225 ms	78635 ms	25764 ms	25602 ms	1634659 ms	1278387 ms
Type 1, scaling in λ	209 ms	158 ms	91 ms	86 ms	8040 ms	3801 ms
Type 2, scaling in λ	234 ms	199 ms	2741 ms	2585 ms	67458 ms	13459 ms
Type 3, scaling in λ	677 ms	665 ms	15891 ms	15897 ms	196328 ms	102182 ms
Type 1, scaling in # confs	1088 ms	1048 ms	114 ms	104 ms	53460 ms	683 ms

Table 8.7: Comparison of collective algorithms. The times shown are the times it took an algorithm to solve all the VPGs in a problem set collectively.

Furthermore, we observe that the explicit variant of the recursive algorithm performs decent across most games. We conclude from this that the efficiency of the symbolic algorithm does not only come from representing sets of configurations efficiently; using a collective approach even without this representation seems to be efficient. It seems that, for the games we experimented with, using the explicit recursive algorithm never significantly hurts performance but in some cases can significantly increase performance compared to the independent approach.

Earlier we hypothesized that a local-collective approach would increase performance more compared to a global-collective approach than a local-independent approach would compared to a global-independent approach. We observe that this is the case for the incremental pre-solve algorithm but not at all the case for the recursive algorithms. Furthermore, this is only the case for the random games; in the SPL games there is not a significant difference in relative performance. We conclude that local solving has the potential to increase performance, however this is not a given and depends on the algorithm and the type of VPG.

9. Conclusion

An SPL can be verified using traditional model-checking techniques. These techniques check every product described by the SPL independently. However, the number of products potentially scales exponentially in the number of features. So we could end up with a large number of products which makes independent checking undesirable. We have presented a method of model-checking an SPL, that models its behaviour using an FTS, such that commonalities between the different products are exploited to increase performance.

We generalized parity games to express variability; these games are called variability parity games (VPGs). VPGs express variability through configurations; a VPG describes a parity game for every configuration. We have shown that we can construct a VPG from an FTS and modal μ-calculus formula such that solving the FTS gives the information needed to decide which products satisfy the formula. VPGs can be solved independently where we solve every parity game described by the VPG separately. We introduced several collective algorithms that solve a VPG as a whole and try to exploit commonalities between the different configurations.

First we introduced a variant of Zielonka's recursive algorithm that solves VPGs. The algorithm views a VPG as a collection of parity games; a parity game for every configuration. We can represent such a collection with a single game graph and for every vertex and edge we have a set of configurations indicating if this vertex or edge is part of the parity game of that configuration. We modified the recursive algorithm to use such a representation. Specifically, we modified the attractor algorithm to try and attract multiple configurations per vertex at the same time. This modified attractor algorithm relies heavily on set operations over the sets of configurations associated with the vertices and edges. These sets can be either represented symbolically or explicitly, giving two variants of the recursive algorithm for VPGs.

Next we introduced the incremental pre-solve algorithm for VPGs. This algorithm tries to find vertices that are won by one of the players for all configurations, if such a vertex is found it is said to be pre-solved. The algorithm tries to find these vertices and then splits the configurations in two sets and goes into recursion for both of them. In the recursion the configuration set has decreased in size so potentially more vertices can be pre-solved. The algorithm finds these vertices through solving pessimistic parity games. Pessimistic parity games are created from a VPG and for a player α, they have the property that any vertex won by player α is also won by player α in the VPG played for any configuration. The incremental pre-solve algorithm creates two pessimistic parity games (for player 0 and 1) and solves them using the fixed-point iteration algorithm. The fixed-point algorithm is modified to use vertices that already were pre-solved to increase its performance. The algorithm incrementally builds up the set of pre-solved vertices until either all vertices are pre-solved or a single configuration remains. In the worst case the algorithm solves linearly more (pessimistic) parity game than using an independent approach would. However, by increasing the number of pre-solved vertices the algorithm tries to outperform the independent approach by solving the (pessimistic) parity games increasingly quicker.

We introduced local variants of the recursive algorithm for parity games and the fixed-point algorithm for parity games. A local parity game algorithm tries to only determine the winner of a single vertex instead of all the vertices, potentially increasing its performance. We also introduced local variants of the collective algorithms mentioned above.

The incremental pre-solve algorithm has the same time complexity as independently solving a VPG using the fixed-point iteration algorithm. The recursive algorithms for VPGs have a worse worst-case time complexity than independently solving a VPG using the recursive algorithm for parity games. However, the aim of the algorithms is to solve VPGs originating from FTSs efficiently. Often times these VPGs will have a lot of commonalities between the configurations. The algorithms are implemented and their actual performance is compared to independent approaches. The minepump and elevator SPLs are used to evaluate the performances. Furthermore a collection of random games is created with different characteristics ranging from very similar to VPGs originating from SPLs to completely different from VPGs originating from SPLs.

We observed that for the SPL VPGs and random VPGs created to be similar to SPL VPGs the symbolic variant of the incremental pre-solve algorithm performs best. The incremental pre-solve algorithm generally also outperforms its independent counterpart, i.e. independently solving a VPG using the fixed-point iteration algorithm. However, for the SPL VPGs, it does so less significantly and consistently than the symbolic recursive algorithm does. It also scales poorer in the number of features than the symbolic recursive algorithm does. Furthermore the independent approach using the recursive algorithm greatly outperforms the independent approach using the fixed-point algorithm. Because the incremental pre-solve algorithm uses the fixed-point algorithm its absolute performance is significantly worse than the recursive algorithm for VPGs.

We observed the explicit recursive variant performs either very similar or better than the independent approach across all games considered. From this we conclude that, even when VPGs are less similar to the SPL VPGs, there is room to exploit commonalities and in some cases increase performance without running the risk of significantly decreasing performance. Whether there are types of VPGs for which the explicit algorithm would perform significantly worse than the independent approach is left unanswered.

Notably, the difference between the local and global variants of the recursive algorithms for VPGs is very little. However, the difference between the local and global variant of the incremental pre-solve algorithm is large for some of the games. However, the difference is not large for the SPL VPGs. We conclude that local algorithms for VPGs can increase performance compared to global algorithms, more so that locally solving parity games increases performance compared to globally solving parity games. However, this is highly dependent on the algorithm and the type of VPG.

Future work Even though the incremental pre-solve algorithms performance was not the best, it did on average outperform its independent counterpart. Therefore it would be interesting to study the incremental pre-solve algorithm using a different way of solving pessimistic parity games; for example, using a variant of the recursive algorithm that can work with pre-solved vertices. This would potentially yield an algorithm that is more *robust* than the symbolic recursive algorithm in the sense that it performs well across different VPGs and not only for VPGs that originate from SPLs.

In this research we did not look into ways of splitting configurations in such a way that the incremental pre-solve algorithm performs well; we simply split configurations based on an arbitrary feature. In [37] it is observed that finding good heuristics for splitting sets of products is relevant to efficiently verifying SPLs. It would be interesting to study such heuristics and to evaluate if heuristics found for VPGs would applicable to the method described in [37], and vice versa.

Furthermore, many optimizations are known for solving parity games [39, 19, 21, 42]. It would be interesting to study if these improvements are applicable to VPGs and if they increase the performance of VPG solving more than they increase the performance of parity game solving.

Finally, the creation of VPGs is left unstudied in this thesis, it would be interesting to study how one could efficiently create VPGs from FTSs including the creation of BDDs.

Bibliography

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

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

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

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

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

[6] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse. The mcrl2 toolset for analysing concurrent systems. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.

[7] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, Mar 2000.

[8] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, Aug 2013.

[9] 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.

[10] A. Classen, P. Heymans, P. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 321–330, May 2011.

[11] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 335–344, May 2010.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 2001.

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

[14] A. Fantechi and S. Gnesi. Formal modeling for product families engineering. In *2008 12th International Software Product Line Conference*, pages 193–202, Sep. 2008.

[15] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, ROSATEA '06, pages 39–48, New York, NY, USA, 2006. ACM.

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

[17] O. Friedmann. Recursive algorithm for parity games requires exponential time. *RAIRO. Theoretical Informatics and Applications*, 45, 11 2011.

[18] 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.

[19] 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.

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

[21] M. Gazda and T. Willemse. Zielonka's recursive algorithm : dull, weak and solitaire games and tighter bounds. In G. Puppis and T. Villa, editors, *Fourth International Symposium on Games, Automata, Logics and Formal Verification (Borca di Cadore, Dolomites, Italy, August 29-31, 2013)*, Electronic Proceedings in Theoretical Computer Science, pages 7–20. EPTCS, 2013.

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

[23] M. Jurdziski. Deciding the winner in parity games is in up $\cap$ co-up. *Information Processing Letters*, 68(3):119 – 124, 1998.

[24] K. Lauenroth and K. Pohl and S. Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, volume , pages 269–280, Nov 2009.

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

[26] K. G. Larsen, U. Nyman, and A. Wsowski. Modal i/o automata for interface and product line theories. In R. De Nicola, editor, *Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[27] J. Lind-Nielsen. *BuDDy : A binary decision diagram package.* 1999.

[28] P. Manolios. *Mu-Calculus Model-Checking*, pages 93–111. Springer US, Boston, MA, 2000.

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

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

[31] A. Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[32] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.

[33] L. Sanchez, W. Wesselink, and T. A. C. Willemse. A comparison of bdd-based parity game solvers. In *Proceedings Ninth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2018, Saarbrücken, Germany, 26-28th September 2018.*, pages 103–117, 2018.

[34] 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.

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

[36] M. Ter Beek, E. De Vink, and T. Willemse. Towards a feature mu-calculus targeting spl verification. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 206:61–75, 3 2016.

[37] 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.

[38] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, 85(2):287 – 315, 2016.

[39] 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.

[40] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54, Aug 2001.

[41] M. Vardi and P. Wolper. Automata-theoretic approach to automatic program verification. 01 1986.

[42] M. Verver. Practical improvements to parity game solving. December 2013.

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

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

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

A. Running time results

A.1 Minepump

	Independent recursive
1	32.244785 ms
2	57.142866 ms
3	177.300524 ms
4	133.978037 ms
5	146.279183 ms
6	226.548281 ms
7	130.892825 ms
8	19.238491 ms
9	107.901228 ms

	Independent recursive local
1	16.516449 ms
2	56.678234 ms
3	166.261729 ms
4	106.755556 ms
5	109.640023 ms
6	143.138385 ms
7	104.86387 ms
8	8.027416 ms
9	76.519869 ms

	Independent fixed-point iteration
1	29.264917 ms
2	40.674684 ms
3	137.869316 ms
4	14726.550907 ms
5	625.375016 ms
6	329.514887 ms
7	356.932852 ms
8	4.185899 ms
9	462.522931 ms

	Independent fixed-point iteration local
1	29.151388 ms
2	40.714354 ms
3	137.886107 ms
4	14727.598255 ms
5	403.33502 ms
6	241.624277 ms
7	357.023658 ms
8	4.211994 ms
9	47.330273 ms

	Collective recursive explicit
1	21.105355 ms
2	43.435309 ms
3	184.89286 ms
4	249.706248 ms
5	81.458676 ms
6	290.469176 ms
7	78.83509 ms
8	15.117246 ms
9	53.835425 ms

	Collective recursive explicit local
1	12.970156 ms
2	43.580676 ms
3	185.462523 ms
4	193.083871 ms
5	81.18502 ms
6	290.545893 ms
7	77.43355 ms
8	11.307562 ms
9	46.51709 ms

	Collective recursive symbolic
1	3.933221 ms
2	6.762879 ms
3	25.027742 ms
4	37.363271 ms
5	12.63568 ms
6	42.715093 ms
7	11.664699 ms
8	1.07143 ms
9	6.86998 ms

	Collective recursive symbolic local
1	1.726035 ms
2	6.785146 ms
3	24.711643 ms
4	27.177316 ms
5	12.641605 ms
6	42.53417 ms
7	11.653422 ms
8	0.574656 ms
9	5.629393 ms

	Incremental pre-solve
1	301.939317 ms
2	249.815426 ms
3	863.458598 ms
4	2101.13278 ms
5	729.301981 ms
6	532.095869 ms
7	609.091435 ms
8	1.075446 ms
9	511.882915 ms

	Incremental pre-solve local
1	301.278774 ms
2	1.272113 ms
3	129.359424 ms
4	1483.78104 ms
5	392.239293 ms
6	300.501929 ms
7	611.020332 ms
8	1.063772 ms
9	2.727468 ms

A.2 Elevator

	Independent recursive
1	14466.302056 ms
2	14915.141763 ms
3	15915.937528 ms
4	16807.44934 ms
5	8897.951572 ms
6	4209.467145 ms
7	4256.189379 ms

	Independent recursive local
1	11838.428495 ms
2	12506.151312 ms
3	13098.469679 ms
4	13802.101726 ms
5	8183.027254 ms
6	3065.580388 ms
7	3094.191485 ms

	Independent fixed-point iteration
1	1401628.67246 ms
2	1067460.29847 ms
3	2332354.10185 ms
4	2424947.16972 ms
5	102844.054783 ms
6	32238.619157 ms
7	31994.798117 ms

	Independent fixed-point iteration local
1	199995.861137 ms
2	181748.74947 ms
3	545209.245758 ms
4	1069414.76658 ms
5	78133.171755 ms
6	13345.84391 ms
7	20690.864052 ms

	Collective recursive explicit
1	15983.614369 ms
2	16555.489895 ms
3	16789.886104 ms
4	17433.006986 ms
5	8948.312123 ms
6	2773.313927 ms
7	2741.204448 ms

	Collective recursive explicit local
1	14872.319637 ms
2	16197.062632 ms
3	16679.869784 ms
4	17439.30272 ms
5	8826.895628 ms
6	2318.720608 ms
7	2300.391218 ms

	Collective recursive symbolic
1	5378.403412 ms
2	5684.740647 ms
3	5039.525624 ms
4	5295.952172 ms
5	2872.894073 ms
6	766.722225 ms
7	725.489882 ms

	Collective recursive symbolic local
1	5060.426053 ms
2	5777.377482 ms
3	5184.639277 ms
4	5428.615541 ms
5	2925.932923 ms
6	631.693549 ms
7	593.154428 ms

	Incremental pre-solve
1	469686.601635 ms
2	288973.070041 ms
3	402712.387165 ms
4	420408.603193 ms
5	25793.497523 ms
6	13467.418415 ms
7	13617.735125 ms

	Incremental pre-solve local
1	339790.3129 ms
2	223654.225843 ms
3	347841.546228 ms
4	356190.556856 ms
5	8342.778568 ms
6	1258.440146 ms
7	1308.733329 ms

A.3 Random games of type 1, scaling in λ

	Independent recursive
75	5.783399 ms
76	3.507118 ms
77	68.609542 ms
78	4.307716 ms
79	171.997666 ms
80	3.473343 ms
81	20.470233 ms
82	43.836144 ms
83	10.754656 ms
84	291.048354 ms
85	34.906948 ms
86	105.089482 ms
87	52.09508 ms
88	217.12306 ms
89	6.753343 ms
90	499.414706 ms
91	66.743923 ms
92	19.052549 ms
93	3.431335 ms
94	3.383136 ms
95	60.166179 ms
96	68.754161 ms
97	55.764131 ms
98	8.832847 ms
99	4.415689 ms

	Independent fixed-point iteration
75	19.627751 ms
76	2.14961 ms
77	314.44995 ms
78	105.881319 ms
79	9147.908303 ms
80	6.795065 ms
81	67.508151 ms
82	35.756278 ms
83	76.968366 ms
84	3205.297742 ms
85	59.952406 ms
86	528.268287 ms
87	41.405433 ms
88	36154.141433 ms
89	5.312185 ms
90	1734.147577 ms
91	214.862985 ms
92	38.540643 ms
93	16.448299 ms
94	0.814891 ms
95	188.552178 ms
96	933.268529 ms
97	581.380477 ms
98	184.611497 ms
99	12.650092 ms

	Independent recursive local
75	4.517718 ms
76	3.349256 ms
77	60.562244 ms
78	4.157373 ms
79	168.280489 ms
80	2.656416 ms
81	19.806984 ms
82	43.172386 ms
83	10.726911 ms
84	280.660021 ms
85	30.78568 ms
86	105.265016 ms
87	47.11039 ms
88	213.383031 ms
89	5.910715 ms
90	466.893468 ms
91	60.137594 ms
92	17.647992 ms
93	2.539196 ms
94	3.181258 ms
95	56.683168 ms
96	63.162656 ms
97	47.249534 ms
98	8.310182 ms
99	3.762845 ms

	Independent fixed-point iteration local
75	10.478241 ms
76	1.66616 ms
77	150.873986 ms
78	94.269122 ms
79	9152.663811 ms
80	3.13701 ms
81	67.256277 ms
82	35.863347 ms
83	76.960887 ms
84	2585.838483 ms
85	37.120349 ms
86	528.651148 ms
87	34.945019 ms
88	36241.631034 ms
89	2.324323 ms
90	604.389592 ms
91	214.881639 ms
92	38.737549 ms
93	7.405805 ms
94	0.81029 ms
95	189.256045 ms
96	932.433677 ms
97	304.747583 ms
98	184.646778 ms
99	4.620931 ms

	Collective recursive explicit
75	13.179941 ms
76	5.045615 ms
77	6.336736 ms
78	9.772348 ms
79	14.215647 ms
80	4.898211 ms
81	9.771256 ms
82	8.075173 ms
83	3.630777 ms
84	34.836531 ms
85	3.598688 ms
86	4.393503 ms
87	1.743596 ms
88	15.101981 ms
89	1.528683 ms
90	22.885089 ms
91	6.176384 ms
92	1.976313 ms
93	8.196764 ms
94	1.971804 ms
95	4.259683 ms
96	8.133055 ms
97	4.606805 ms
98	9.784656 ms
99	5.056011 ms

	Collective recursive symbolic
75	6.942651 ms
76	2.46779 ms
77	3.948197 ms
78	5.674459 ms
79	8.810585 ms
80	2.776076 ms
81	3.996077 ms
82	4.821725 ms
83	3.773631 ms
84	14.074491 ms
85	2.593924 ms
86	0.861131 ms
87	0.797176 ms
88	4.577257 ms
89	1.197462 ms
90	3.340015 ms
91	5.222941 ms
92	1.08472 ms
93	2.707787 ms
94	0.339749 ms
95	0.453768 ms
96	2.956694 ms
97	2.951662 ms
98	4.460949 ms
99	0.652745 ms

	Collective recursive explicit local
75	13.200242 ms
76	4.425631 ms
77	4.606634 ms
78	9.009662 ms
79	14.223333 ms
80	2.603217 ms
81	3.365161 ms
82	4.929775 ms
83	3.564961 ms
84	20.384742 ms
85	2.905875 ms
86	3.191562 ms
87	1.644649 ms
88	13.847766 ms
89	1.049236 ms
90	18.964289 ms
91	6.147093 ms
92	1.962646 ms
93	3.024402 ms
94	1.515896 ms
95	1.801201 ms
96	5.530923 ms
97	4.290898 ms
98	9.632382 ms
99	2.472193 ms

	Collective recursive symbolic local
75	9.684677 ms
76	2.348568 ms
77	3.455208 ms
78	5.663252 ms
79	8.707741 ms
80	1.524961 ms
81	4.0165 ms
82	4.820994 ms
83	3.768966 ms
84	9.240854 ms
85	2.363849 ms
86	0.859659 ms
87	0.749002 ms
88	4.490098 ms
89	1.135063 ms
90	3.203913 ms
91	5.210439 ms
92	1.068052 ms
93	2.270763 ms
94	0.341987 ms
95	0.453611 ms
96	2.834343 ms
97	2.868619 ms
98	4.468894 ms
99	0.609133 ms

	Incremental pre-solve
75	24.10313 ms
76	11.218956 ms
77	245.581084 ms
78	79.38162 ms
79	729.210486 ms
80	12.009477 ms
81	80.603635 ms
82	221.42212 ms
83	39.769917 ms
84	3906.57147 ms
85	115.230275 ms
86	158.678762 ms
87	47.751048 ms
88	760.61273 ms
89	12.159277 ms
90	1171.34907 ms
91	235.925567 ms
92	34.046692 ms
93	8.460347 ms
94	0.131598 ms
95	3.280932 ms
96	82.014619 ms
97	39.518362 ms
98	11.305339 ms
99	9.764942 ms

	Incremental pre-solve local
75	12.429163 ms
76	7.00983 ms
77	1.076886 ms
78	44.26665 ms
79	17.488957 ms
80	0.498895 ms
81	1.032711 ms
82	0.618111 ms
83	2.150437 ms
84	3652.819939 ms
85	0.492338 ms
86	0.9242 ms
87	0.208906 ms
88	39.231847 ms
89	0.22777 ms
90	1.332649 ms
91	1.409215 ms
92	0.542712 ms
93	1.535109 ms
94	0.117479 ms
95	0.466705 ms
96	4.859004 ms
97	2.771066 ms
98	6.767962 ms
99	0.554674 ms

A.4 Random games of type 2, scaling in λ

	Independent recursive
75	5.572237 ms
76	3.886284 ms
77	71.522315 ms
78	2.304195 ms
79	175.527 ms
80	3.462186 ms
81	19.904324 ms
82	49.190906 ms
83	9.397631 ms
84	370.173095 ms
85	33.034355 ms
86	101.017842 ms
87	54.011849 ms
88	277.018051 ms
89	7.332896 ms
90	482.92123 ms
91	61.479266 ms
92	17.699253 ms
93	3.155225 ms
94	3.254518 ms
95	56.647944 ms
96	58.454469 ms
97	55.691609 ms
98	9.093579 ms
99	4.250813 ms

	Independent fixed-point iteration
75	13.248044 ms
76	4.904823 ms
77	212.49508 ms
78	29.021233 ms
79	11100.371072 ms
80	6.147743 ms
81	52.954776 ms
82	34.200157 ms
83	94.732149 ms
84	14663.811045 ms
85	94.903842 ms
86	678.222564 ms
87	38.267891 ms
88	43620.52326 ms
89	4.121256 ms
90	1494.600774 ms
91	225.48317 ms
92	44.600972 ms
93	15.181433 ms
94	4.081361 ms
95	210.571513 ms
96	992.03064 ms
97	963.018774 ms
98	105.644549 ms
99	7.015923 ms

	Independent recursive local
75	4.993032 ms
76	2.777006 ms
77	51.969796 ms
78	2.145411 ms
79	166.831989 ms
80	2.625606 ms
81	19.594841 ms
82	48.034131 ms
83	8.562243 ms
84	366.608514 ms
85	25.455226 ms
86	73.522082 ms
87	47.168199 ms
88	261.574307 ms
89	5.020165 ms
90	368.893109 ms
91	66.534406 ms
92	18.584134 ms
93	2.570556 ms
94	3.267228 ms
95	51.163045 ms
96	56.701722 ms
97	53.53649 ms
98	8.846735 ms
99	4.164454 ms

	Independent fixed-point iteration local
75	12.470087 ms
76	1.470259 ms
77	95.511138 ms
78	27.627334 ms
79	11093.380929 ms
80	2.364584 ms
81	53.370789 ms
82	33.992381 ms
83	46.078413 ms
84	14230.865929 ms
85	40.219652 ms
86	235.093898 ms
87	25.92723 ms
88	43606.352352 ms
89	2.162115 ms
90	585.769655 ms
91	224.934857 ms
92	44.599862 ms
93	4.075561 ms
94	3.437583 ms
95	98.24308 ms
96	992.33474 ms
97	963.997442 ms
98	105.2284 ms
99	6.975608 ms

	Collective recursive explicit
75	11.582399 ms
76	5.557129 ms
77	5.882631 ms
78	3.56182 ms
79	18.407353 ms
80	4.50393 ms
81	3.715213 ms
82	5.772085 ms
83	5.68015 ms
84	61.428437 ms
85	3.011144 ms
86	11.129041 ms
87	2.04068 ms
88	23.852764 ms
89	3.053057 ms
90	20.924669 ms
91	6.294277 ms
92	3.608108 ms
93	5.046793 ms
94	1.609743 ms
95	1.899699 ms
96	6.603682 ms
97	4.909781 ms
98	10.970918 ms
99	2.646825 ms

	Collective recursive symbolic
75	8.590814 ms
76	6.154989 ms
77	85.029403 ms
78	4.249487 ms
79	454.203139 ms
80	3.51584 ms
81	10.555987 ms
82	35.00474 ms
83	5.56905 ms
84	1557.613182 ms
85	8.732898 ms
86	35.374897 ms
87	9.563621 ms
88	264.682693 ms
89	1.904499 ms
90	214.847173 ms
91	12.082922 ms
92	3.388263 ms
93	2.774365 ms
94	0.5237 ms
95	1.53478 ms
96	4.988797 ms
97	4.497413 ms
98	4.880229 ms
99	0.642964 ms

	Collective recursive explicit local
75	12.371467 ms
76	3.326987 ms
77	3.581174 ms
78	4.136829 ms
79	17.425269 ms
80	2.178803 ms
81	3.366602 ms
82	5.695551 ms
83	2.455281 ms
84	61.029529 ms
85	1.862928 ms
86	3.866284 ms
87	1.950294 ms
88	23.496673 ms
89	0.736546 ms
90	13.178012 ms
91	6.20594 ms
92	2.287151 ms
93	4.422565 ms
94	1.394032 ms
95	1.238352 ms
96	4.873597 ms
97	4.63077 ms
98	10.86504 ms
99	2.670858 ms

	Collective recursive symbolic local
75	12.884992 ms
76	2.628672 ms
77	42.703954 ms
78	4.302818 ms
79	457.175236 ms
80	1.612656 ms
81	10.656932 ms
82	35.11936 ms
83	5.11544 ms
84	1567.519498 ms
85	5.046267 ms
86	15.793766 ms
87	9.364436 ms
88	270.137893 ms
89	1.01092 ms
90	108.909367 ms
91	12.10892 ms
92	3.452279 ms
93	2.920813 ms
94	0.578589 ms
95	1.153317 ms
96	4.919032 ms
97	4.535802 ms
98	4.890286 ms
99	0.64128 ms

	Incremental pre-solve
75	40.864312 ms
76	24.04339 ms
77	2125.279617 ms
78	74.18574 ms
79	10548.586273 ms
80	21.25207 ms
81	230.552061 ms
82	891.771734 ms
83	77.268964 ms
84	15161.231952 ms
85	413.975688 ms
86	804.467899 ms
87	360.230714 ms
88	6193.263862 ms
89	29.984868 ms
90	28914.631371 ms
91	694.84558 ms
92	71.502263 ms
93	15.43645 ms
94	6.294129 ms
95	91.693459 ms
96	323.904371 ms
97	296.481792 ms
98	36.927466 ms
99	9.060785 ms

	Incremental pre-solve local
75	35.812991 ms
76	19.939048 ms
77	11.267082 ms
78	73.497531 ms
79	45.171472 ms
80	0.82058 ms
81	3.454883 ms
82	7.587814 ms
83	3.10014 ms
84	13126.034303 ms
85	3.565127 ms
86	4.315561 ms
87	2.283749 ms
88	52.009152 ms
89	0.493271 ms
90	38.726892 ms
91	5.756349 ms
92	1.065428 ms
93	1.30138 ms
94	5.638633 ms
95	1.014387 ms
96	5.851124 ms
97	5.45393 ms
98	4.119885 ms
99	0.43048 ms

A.5 Random games of type 3, scaling in λ

	Independent recursive
75	5.151231 ms
76	3.604603 ms
77	70.708286 ms
78	2.329638 ms
79	196.596803 ms
80	4.219329 ms
81	21.676263 ms
82	47.541397 ms
83	9.635504 ms
84	347.51953 ms
85	34.52527 ms
86	119.458258 ms
87	53.553651 ms
88	223.079026 ms
89	6.848982 ms
90	480.2348 ms
91	58.862406 ms
92	17.289855 ms
93	2.896246 ms
94	3.320768 ms
95	45.185803 ms
96	67.472313 ms
97	57.18095 ms
98	8.827342 ms
99	4.352249 ms

	Independent fixed-point iteration
75	25.575229 ms
76	5.925763 ms
77	243.831927 ms
78	109.736644 ms
79	11020.561653 ms
80	6.065144 ms
81	77.810369 ms
82	37.962145 ms
83	112.963567 ms
84	10825.315091 ms
85	93.401165 ms
86	683.16394 ms
87	33.900335 ms
88	35623.490809 ms
89	4.028757 ms
90	1495.140344 ms
91	198.317844 ms
92	29.820193 ms
93	20.077013 ms
94	3.824404 ms
95	251.811213 ms
96	730.404449 ms
97	663.050439 ms
98	264.842467 ms
99	7.13727 ms

	Independent recursive local
75	4.321471 ms
76	3.36463 ms
77	67.045277 ms
78	2.463787 ms
79	179.7972 ms
80	4.226262 ms
81	22.061602 ms
82	41.275329 ms
83	10.009246 ms
84	342.845079 ms
85	26.856433 ms
86	121.77033 ms
87	55.439681 ms
88	224.774085 ms
89	7.276428 ms
90	343.94985 ms
91	58.962511 ms
92	16.825969 ms
93	2.460236 ms
94	3.214327 ms
95	37.36455 ms
96	55.541583 ms
97	38.940735 ms
98	5.902098 ms
99	3.167523 ms

	Independent fixed-point iteration local
75	6.447583 ms
76	3.937245 ms
77	86.499632 ms
78	107.420124 ms
79	4505.261921 ms
80	6.09288 ms
81	77.817817 ms
82	16.59241 ms
83	113.320589 ms
84	10658.541404 ms
85	43.264458 ms
86	684.641534 ms
87	34.090233 ms
88	32877.686634 ms
89	4.090678 ms
90	534.077027 ms
91	199.14394 ms
92	29.995618 ms
93	7.908287 ms
94	3.828793 ms
95	156.78394 ms
96	638.498578 ms
97	332.57441 ms
98	99.536863 ms
99	2.512545 ms

	Collective recursive explicit
75	12.902371 ms
76	15.140991 ms
77	18.717939 ms
78	11.013356 ms
79	77.8986 ms
80	9.891766 ms
81	9.0947 ms
82	14.099915 ms
83	7.211647 ms
84	183.074633 ms
85	7.353104 ms
86	29.286211 ms
87	4.941745 ms
88	121.480495 ms
89	2.922741 ms
90	63.375097 ms
91	12.290787 ms
92	10.663925 ms
93	12.072567 ms
94	2.519906 ms
95	7.509088 ms
96	11.690614 ms
97	8.799267 ms
98	15.079385 ms
99	7.514165 ms

	Collective recursive symbolic
75	18.54137 ms
76	18.296847 ms
77	476.01134 ms
78	7.939144 ms
79	2959.308729 ms
80	7.872336 ms
81	55.342337 ms
82	273.237373 ms
83	20.358435 ms
84	6615.501783 ms
85	48.054967 ms
86	519.476449 ms
87	36.730716 ms
88	2886.350328 ms
89	6.518033 ms
90	1710.51928 ms
91	88.979346 ms
92	35.9586 ms
93	9.854976 ms
94	2.611188 ms
95	24.157741 ms
96	31.650372 ms
97	23.72306 ms
98	11.150034 ms
99	2.903731 ms

	Collective recursive explicit local
75	7.104126 ms
76	7.483722 ms
77	14.410184 ms
78	7.374659 ms
79	52.12921 ms
80	7.179271 ms
81	8.931317 ms
82	12.31199 ms
83	5.816688 ms
84	196.619454 ms
85	9.118515 ms
86	29.225048 ms
87	4.979079 ms
88	156.352514 ms
89	1.860649 ms
90	65.152861 ms
91	12.285151 ms
92	7.372765 ms
93	4.875889 ms
94	2.525663 ms
95	14.717426 ms
96	13.9454 ms
97	11.64738 ms
98	8.958566 ms
99	2.744124 ms

	Collective recursive symbolic local
75	8.698159 ms
76	15.287062 ms
77	404.360803 ms
78	7.927143 ms
79	2045.078 ms
80	7.938775 ms
81	54.964416 ms
82	250.826435 ms
83	20.280726 ms
84	6554.989919 ms
85	59.498466 ms
86	513.878872 ms
87	36.316144 ms
88	3821.29137 ms
89	6.040122 ms
90	1821.926626 ms
91	88.742239 ms
92	36.185988 ms
93	8.150113 ms
94	2.62318 ms
95	49.1887 ms
96	40.688449 ms
97	33.738615 ms
98	6.778539 ms
99	1.288989 ms

	Incremental pre-solve
75	73.734811 ms
76	28.641868 ms
77	3754.799599 ms
78	123.479098 ms
79	29096.679104 ms
80	33.534004 ms
81	459.50578 ms
82	2065.343313 ms
83	172.782392 ms
84	31559.535147 ms
85	859.952952 ms
86	3885.281168 ms
87	1055.404786 ms
88	34179.411694 ms
89	56.631884 ms
90	84285.964489 ms
91	1855.988683 ms
92	188.596517 ms
93	31.280134 ms
94	13.42136 ms
95	369.221645 ms
96	1124.808958 ms
97	948.932646 ms
98	89.644036 ms
99	15.072768 ms

	Incremental pre-solve local
75	43.558063 ms
76	27.014061 ms
77	2196.073022 ms
78	107.994854 ms
79	22233.181687 ms
80	7.830664 ms
81	271.692867 ms
82	1071.624026 ms
83	60.29467 ms
84	26370.072606 ms
85	303.293351 ms
86	1325.389916 ms
87	343.330116 ms
88	33579.322758 ms
89	9.660728 ms
90	13009.351329 ms
91	307.125961 ms
92	76.306882 ms
93	21.666715 ms
94	7.713631 ms
95	207.979102 ms
96	363.974039 ms
97	191.276935 ms
98	42.050393 ms
99	4.143319 ms

A.6 Random games of type 1, scaling in the number of features

	Independent recursive
2.0	0.831682 ms
2.25	0.787779 ms
2.5	0.630085 ms
2.75	0.768894 ms
3.0	1.808512 ms
3.25	0.873079 ms
3.5	1.303374 ms
3.75	0.847835 ms
4.0	1.681708 ms
4.25	3.279441 ms
4.5	3.66918 ms
4.75	1.558638 ms
5.0	4.309323 ms
5.25	2.893653 ms
5.5	5.59452 ms
5.75	5.736842 ms
6.0	12.759288 ms
6.25	6.920441 ms
6.5	12.602162 ms
6.75	8.326259 ms
7.0	12.128731 ms
7.25	14.289059 ms
7.5	16.616108 ms
7.75	14.929757 ms
8.0	51.708592 ms
8.25	29.056171 ms
8.5	38.009515 ms
8.75	65.541624 ms
9.0	83.306977 ms
9.25	77.116324 ms
9.5	88.795178 ms
9.75	104.625002 ms
10.0	109.839213 ms
10.25	201.184011 ms
10.5	71.639255 ms
10.75	173.372201 ms
11.0	278.696424 ms
11.25	190.420726 ms
11.5	166.337719 ms
11.75	212.147761 ms
12.0	446.367466 ms
12.25	411.10256 ms
12.5	611.590006 ms
12.75	376.966971 ms
13.0	1048.267745 ms
13.25	1460.488019 ms
13.5	1159.976893 ms
13.75	1428.482043 ms
14.0	1319.015823 ms
14.25	2558.800215 ms
14.5	2547.022879 ms
14.75	1702.58316 ms
15.0	3637.486788 ms
15.25	3077.368185 ms
15.5	4212.603012 ms
15.75	4828.130663 ms

	Independent recursive local
2.0	0.845191 ms
2.25	0.767179 ms
2.5	0.53975 ms
2.75	0.572876 ms
3.0	1.332794 ms
3.25	0.708959 ms
3.5	1.291144 ms
3.75	0.843332 ms
4.0	1.614021 ms
4.25	3.231037 ms
4.5	2.814344 ms
4.75	1.333007 ms
5.0	3.510366 ms
5.25	2.605022 ms
5.5	5.960058 ms
5.75	6.280421 ms
6.0	12.824149 ms
6.25	6.878454 ms
6.5	8.105308 ms
6.75	7.782662 ms
7.0	10.466472 ms
7.25	13.976853 ms
7.5	13.401955 ms
7.75	15.546727 ms
8.0	47.294253 ms
8.25	26.072257 ms
8.5	38.181981 ms
8.75	51.229874 ms
9.0	58.960801 ms
9.25	77.765224 ms
9.5	91.086372 ms
9.75	106.955251 ms
10.0	105.642766 ms
10.25	149.697167 ms
10.5	59.093611 ms
10.75	161.967224 ms
11.0	260.206377 ms
11.25	188.520127 ms
11.5	148.423159 ms
11.75	211.188823 ms
12.0	329.366103 ms
12.25	326.42036 ms
12.5	617.843715 ms
12.75	377.525177 ms
13.0	1062.828274 ms
13.25	1193.730158 ms
13.5	879.727791 ms
13.75	1365.846105 ms
14.0	1337.705727 ms
14.25	2004.668309 ms
14.5	1998.261304 ms
14.75	1701.711136 ms
15.0	3586.237182 ms
15.25	2993.832268 ms
15.5	3772.95595 ms
15.75	3938.494334 ms

	Independent fixed-point iteration
2.0	1.222122 ms
2.25	1.133871 ms
2.5	0.526085 ms
2.75	1.256012 ms
3.0	10.007389 ms
3.25	0.653871 ms
3.5	1.128714 ms
3.75	6.658314 ms
4.0	1.222348 ms
4.25	4.570886 ms
4.5	9.554861 ms
4.75	2.521594 ms
5.0	74.04515 ms
5.25	8.378165 ms
5.5	28.951916 ms
5.75	45.63253 ms
6.0	1884.595558 ms
6.25	4.029901 ms
6.5	176.488428 ms
6.75	230.247336 ms
7.0	6.192419 ms
7.25	9.909952 ms
7.5	155.378584 ms
7.75	10.296701 ms
8.0	876.484578 ms
8.25	57.978683 ms
8.5	172.483793 ms
8.75	1186.537532 ms
9.0	62.05414 ms
9.25	450.213645 ms
9.5	271.981159 ms
9.75	6887.116 ms
10.0	124.2075 ms
10.25	31039.306543 ms
10.5	30.166836 ms
10.75	8019.228783 ms
11.0	2837.548451 ms
11.25	123.43605 ms
11.5	60.914666 ms
11.75	111.798733 ms
12.0	3111.775565 ms
12.25	209.116113 ms
12.5	584.160629 ms
12.75	193.744241 ms
13.0	35438.150725 ms
13.25	69412.506047 ms
13.5	50077.86549 ms
13.75	27055.392409 ms
14.0	403.672671 ms
14.25	2860.536252 ms
14.5	37655.061565 ms
14.75	786.531295 ms
15.0	6134.357818 ms
15.25	2955.288551 ms
15.5	118306.907779 ms
15.75	29027.486125 ms

	Independent fixed-point iteration local
2.0	1.221726 ms
2.25	1.122385 ms
2.5	0.33584 ms
2.75	0.472064 ms
3.0	5.498123 ms
3.25	0.314695 ms
3.5	1.111351 ms
3.75	6.665386 ms
4.0	1.226083 ms
4.25	4.582422 ms
4.5	3.885753 ms
4.75	1.623997 ms
5.0	32.555541 ms
5.25	4.824165 ms
5.5	28.753541 ms
5.75	45.750097 ms
6.0	1585.701822 ms
6.25	4.08783 ms
6.5	67.935102 ms
6.75	230.142028 ms
7.0	4.958423 ms
7.25	9.922013 ms
7.5	71.667973 ms
7.75	10.286904 ms
8.0	630.285925 ms
8.25	43.19819 ms
8.5	173.03034 ms
8.75	562.239159 ms
9.0	30.276466 ms
9.25	359.964972 ms
9.5	273.467268 ms
9.75	6883.883355 ms
10.0	124.974115 ms
10.25	13847.918176 ms
10.5	13.333751 ms
10.75	8027.709909 ms
11.0	2840.919689 ms
11.25	123.803546 ms
11.5	32.688601 ms
11.75	111.654487 ms
12.0	1491.764149 ms
12.25	103.405101 ms
12.5	588.127372 ms
12.75	195.841903 ms
13.0	35425.29469 ms
13.25	34923.642069 ms
13.5	21415.153714 ms
13.75	27084.473 ms
14.0	403.123418 ms
14.25	2700.289123 ms
14.5	9553.674124 ms
14.75	789.869555 ms
15.0	6135.838701 ms
15.25	1845.041556 ms
15.5	74896.828066 ms
15.75	20444.306506 ms

	Collective recursive explicit
2.0	3.689382 ms
2.25	3.421375 ms
2.5	1.321109 ms
2.75	2.18316 ms
3.0	4.987218 ms
3.25	0.983535 ms
3.5	1.607078 ms
3.75	1.423996 ms
4.0	1.740855 ms
4.25	9.460022 ms
4.5	9.151982 ms
4.75	1.586286 ms
5.0	4.564488 ms
5.25	2.189078 ms
5.5	6.413069 ms
5.75	6.948876 ms
6.0	3.662662 ms
6.25	1.214467 ms
6.5	6.181864 ms
6.75	6.432064 ms
7.0	1.070284 ms
7.25	1.480217 ms
7.5	2.620133 ms
7.75	2.205924 ms
8.0	5.118212 ms
8.25	3.582384 ms
8.5	4.342961 ms
8.75	8.397231 ms
9.0	7.123557 ms
9.25	4.121446 ms
9.5	6.071836 ms
9.75	7.492039 ms
10.0	3.968228 ms
10.25	16.948431 ms
10.5	1.864118 ms
10.75	15.662477 ms
11.0	13.343244 ms
11.25	9.390556 ms
11.5	5.05927 ms
11.75	5.823042 ms
12.0	18.187711 ms
12.25	16.342897 ms
12.5	21.5048 ms
12.75	10.785761 ms
13.0	35.854223 ms
13.25	48.255687 ms
13.5	37.936213 ms
13.75	34.426066 ms
14.0	27.098822 ms
14.25	80.789609 ms
14.5	108.803807 ms
14.75	54.611687 ms
15.0	107.030578 ms
15.25	45.12552 ms
15.5	111.968127 ms
15.75	124.825594 ms

	Collective recursive explicit local
2.0	2.348458 ms
2.25	2.902567 ms
2.5	1.277929 ms
2.75	1.322573 ms
3.0	2.258321 ms
3.25	0.573683 ms
3.5	1.592866 ms
3.75	1.074024 ms
4.0	1.246714 ms
4.25	3.953101 ms
4.5	3.315987 ms
4.75	0.991203 ms
5.0	2.639893 ms
5.25	2.348003 ms
5.5	5.33634 ms
5.75	6.770864 ms
6.0	3.722354 ms
6.25	1.217468 ms
6.5	1.967965 ms
6.75	3.643114 ms
7.0	1.624517 ms
7.25	1.481051 ms
7.5	1.409988 ms
7.75	1.724534 ms
8.0	6.181395 ms
8.25	3.130761 ms
8.5	3.732697 ms
8.75	4.951618 ms
9.0	2.317672 ms
9.25	4.256602 ms
9.5	6.089713 ms
9.75	7.518627 ms
10.0	3.889302 ms
10.25	5.623036 ms
10.5	1.731465 ms
10.75	8.713679 ms
11.0	8.556716 ms
11.25	5.561564 ms
11.5	4.815343 ms
11.75	5.759282 ms
12.0	8.295555 ms
12.25	6.372578 ms
12.5	21.595398 ms
12.75	10.797121 ms
13.0	35.756249 ms
13.25	85.921009 ms
13.5	22.788833 ms
13.75	34.238522 ms
14.0	24.262727 ms
14.25	69.111176 ms
14.5	161.253008 ms
14.75	54.125483 ms
15.0	101.85229 ms
15.25	40.104211 ms
15.5	109.339547 ms
15.75	122.412531 ms

	Collective recursive symbolic
2.0	3.262736 ms
2.25	3.854871 ms
2.5	1.814205 ms
2.75	3.178633 ms
3.0	4.480284 ms
3.25	1.007125 ms
3.5	1.653495 ms
3.75	1.176028 ms
4.0	1.00418 ms
4.25	2.88306 ms
4.5	3.737277 ms
4.75	1.301316 ms
5.0	2.344471 ms
5.25	0.530157 ms
5.5	2.207829 ms
5.75	3.078832 ms
6.0	2.665448 ms
6.25	1.391205 ms
6.5	1.386358 ms
6.75	2.07436 ms
7.0	0.815236 ms
7.25	1.202583 ms
7.5	1.919336 ms
7.75	1.494987 ms
8.0	4.263079 ms
8.25	1.262287 ms
8.5	1.984775 ms
8.75	5.37509 ms
9.0	1.876328 ms
9.25	1.941336 ms
9.5	3.968792 ms
9.75	3.593196 ms
10.0	1.476037 ms
10.25	1.898132 ms
10.5	0.556618 ms
10.75	2.581225 ms
11.0	1.116342 ms
11.25	0.891594 ms
11.5	1.062457 ms
11.75	1.371136 ms
12.0	1.555298 ms
12.25	1.145364 ms
12.5	2.259401 ms
12.75	1.16684 ms
13.0	2.010952 ms
13.25	2.045104 ms
13.5	2.090693 ms
13.75	1.513717 ms
14.0	0.311262 ms
14.25	1.851603 ms
14.5	5.755441 ms
14.75	1.625853 ms
15.0	1.560027 ms
15.25	0.486548 ms
15.5	1.175458 ms
15.75	1.356144 ms

	Collective recursive symbolic local
2.0	3.258418 ms
2.25	3.882941 ms
2.5	1.664378 ms
2.75	1.686219 ms
3.0	2.449441 ms
3.25	0.518552 ms
3.5	1.737923 ms
3.75	1.191066 ms
4.0	1.016765 ms
4.25	2.89878 ms
4.5	2.043714 ms
4.75	0.695672 ms
5.0	1.209631 ms
5.25	0.85079 ms
5.5	2.20691 ms
5.75	3.098205 ms
6.0	2.692335 ms
6.25	1.387245 ms
6.5	0.492061 ms
6.75	2.08924 ms
7.0	1.536968 ms
7.25	1.206975 ms
7.5	1.185762 ms
7.75	1.490256 ms
8.0	4.162072 ms
8.25	2.033141 ms
8.5	2.011455 ms
8.75	2.991366 ms
9.0	0.921848 ms
9.25	1.948832 ms
9.5	3.939665 ms
9.75	3.552001 ms
10.0	1.490848 ms
10.25	1.212432 ms
10.5	0.525402 ms
10.75	2.566581 ms
11.0	1.104146 ms
11.25	0.889719 ms
11.5	1.027237 ms
11.75	1.348293 ms
12.0	0.827223 ms
12.25	0.630874 ms
12.5	2.271923 ms
12.75	1.166266 ms
13.0	1.996657 ms
13.25	4.203315 ms
13.5	1.258266 ms
13.75	1.530374 ms
14.0	0.315723 ms
14.25	1.606094 ms
14.5	8.342462 ms
14.75	1.65994 ms
15.0	1.553522 ms
15.25	0.500135 ms
15.5	1.062274 ms
15.75	1.276518 ms

	Incremental pre-solve
2.0	3.027823 ms
2.25	3.056003 ms
2.5	2.420596 ms
2.75	2.903228 ms
3.0	8.968049 ms
3.25	2.66567 ms
3.5	3.25113 ms
3.75	3.013486 ms
4.0	5.42656 ms
4.25	12.377157 ms
4.5	14.507135 ms
4.75	4.765442 ms
5.0	18.986578 ms
5.25	5.661775 ms
5.5	15.943753 ms
5.75	22.621107 ms
6.0	77.486745 ms
6.25	19.285423 ms
6.5	28.655507 ms
6.75	40.345489 ms
7.0	28.69868 ms
7.25	35.097079 ms
7.5	51.9112 ms
7.75	52.303305 ms
8.0	45.751888 ms
8.25	82.996192 ms
8.5	134.40839 ms
8.75	269.843533 ms
9.0	121.9773 ms
9.25	204.60547 ms
9.5	262.423825 ms
9.75	301.717134 ms
10.0	121.304941 ms
10.25	594.192081 ms
10.5	8.949736 ms
10.75	860.301552 ms
11.0	158.94577 ms
11.25	113.17543 ms
11.5	0.159143 ms
11.75	602.142005 ms
12.0	156.559805 ms
12.25	944.023468 ms
12.5	434.253602 ms
12.75	1196.079578 ms
13.0	2445.847367 ms
13.25	4712.871414 ms
13.5	4129.468355 ms
13.75	1190.968878 ms
14.0	0.143011 ms
14.25	4019.764436 ms
14.5	6998.876242 ms
14.75	2540.239206 ms
15.0	3314.935058 ms
15.25	2982.548442 ms
15.5	1379.077967 ms
15.75	12668.325032 ms

	Incremental pre-solve local
2.0	0.798992 ms
2.25	2.494454 ms
2.5	0.483175 ms
2.75	0.740626 ms
3.0	2.077046 ms
3.25	0.274311 ms
3.5	0.421767 ms
3.75	0.870524 ms
4.0	0.260408 ms
4.25	0.72037 ms
4.5	1.196638 ms
4.75	0.330133 ms
5.0	3.168872 ms
5.25	0.449824 ms
5.5	1.352182 ms
5.75	1.725661 ms
6.0	63.53004 ms
6.25	0.248895 ms
6.5	3.852187 ms
6.75	3.642727 ms
7.0	10.454824 ms
7.25	0.250003 ms
7.5	1.462712 ms
7.75	0.275768 ms
8.0	4.069187 ms
8.25	1.050522 ms
8.5	1.035266 ms
8.75	5.397 ms
9.0	0.382111 ms
9.25	3.839694 ms
9.5	1.096205 ms
9.75	19.257683 ms
10.0	0.264334 ms
10.25	37.33974 ms
10.5	0.12137 ms
10.75	9.524515 ms
11.0	2.748464 ms
11.25	0.196986 ms
11.5	0.153325 ms
11.75	0.20295 ms
12.0	1.377195 ms
12.25	0.21173 ms
12.5	0.431508 ms
12.75	0.187851 ms
13.0	5.65634 ms
13.25	444.761634 ms
13.5	6.973845 ms
13.75	16.897137 ms
14.0	0.147741 ms
14.25	0.479535 ms
14.5	4.16456 ms
14.75	0.249741 ms
15.0	0.364773 ms
15.25	7.931287 ms
15.5	4.174871 ms
15.75	1.291211 ms