University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master Degree in Computer Science

# Solving Systems of Fixpoint Equations via Strategy Iteration

*Master thesis*

*Supervisor*

Prof. Paolo Baldan

*Candidate*

Giacomo Stevanato

2078263

Academic year 2023-2024

“*Program testing can be used to show the presence of bugs, but never to show their absence*”
— Edsger W. Dijkstra

## Acknowledgements

I thank Prof. Paolo Baldan, my thesis supervisor, for his help, mentoring and all the insightful observations he gave me while writing this thesis.

I am particularly grateful to my parents Giuseppe and Maria Grazia and my sister Francesca for their love and support as they allowed me to chase my dreams.

I thank all my friends for they company and all the laughs we had together in these years.

# Summary

Fixpoint equations and, more generally, systems of fixpoint equations are ubiquitous in a number of formal verification tasks. This includes the model checking of specification logics, like the $\mu$-calculus, and the check of behavioral equivalence, like bisimilarity. Consequently, a recurring problem consists in conceiving and implementing algorithms aimed at determining the solution of these systems. It has been shown in the literature that a game-theoretic characterization of the solution of equational systems can be given in terms of a parity game (a two-player, zero-sum game), which is referred to as fixpoint game or powerset game. The game view opens the way for the development of local algorithms for characterizing the solution of such equation systems. Two classes of algorithms can be identified: global algorithms, aimed at computing the full solution, by deciding the game for all positions, and local algorithms, which instead aim at determining only some "component" of the solution. For instance, in the case of the $\mu$-calculus, of could be interested in checking whether a specific state enjoys or not a property, rather than determining all states satisfying the property. Similarly, for bisimilarity checking one might be interested in establishing whether two states are bisimilar, rather than computing the full equivalence.

A local algorithm for solving a system fo fixpoint equations has been already proposed, based on the local algorithm for parity games due to Stevens and Stirling. In this thesis we explore the possibility of exploiting a different local algorithm for parity games, due to Vöge and Jurdziński, based on local strategy iteration. The idea is to start from arbitrary strategies for game players and progressively provide a local solution for systems of fixpoint equations. We show how this algorithm can be adapted to provide a local solution for systems of fixpoint equations. This is non-trivial as it requires, in particular, to deal with a symbolic representation of the moves of the game, and with a lazy generation of such symbolic moves. An implementation in the language Rust is also provided, and a comparison is conducted with other existing tools.

# Index

# Index of figures

# Index of tables

# 1 Introduction

Systems of mixed least and greatest fixpoint equations over complete lattices are very common in the field of formal analysis and particularly in the field of model checking. A classic example is $\mu$-calculus [1], where liveness and safety properties can be expressed using potentially nested least and greatest fixpoints of functions over sets of states. Behavioral equivalences like many bisimilarities [2] can also be defined as the greatest fixpoint of an appropriate function over the lattice of the binary relations between states. Another example is Łukasiewicz $\mu$-calculus [3], a version of $\mu$-calculus which combines deterministic and probabilistic behavior by using continuous functions over the real numbers interval $[0, 1]$. Abstract interpretation [4] also extensively uses fixpoints of functions over functions representing the abstracted state of the program at various points.

It has thus been the focus of many papers in the literature to provide ways to solve fixpoint equations. Most notably the Knaster-Tarski theorem [5] is a key result for deriving the existence of fixpoints, including the uniqueness of a least and greatest one, while Kleene iteration [6] gives a constructive way to compute them, albeit generally not very efficient, by repeatedly applying the given function to the bottom or top element. However the mixing of least and greatest fixpoint equations into systems of fixpoint equations, while greatly increasing the expressiveness, also complicates the search for the solution. This is the case for example in the $\mu$-calculus, where the use of nested fixpoints is equivalent to a system of mixed least and greatest fixpoint equations.

In this thesis we will build upon the work in [7], which provides a way to characterize the solution of a system of mixed fixpoint equations over some complete lattice through the use of the *powerset game*, a particular parity game, which in turn is a kind of game where two players move a token on a directed graph with the winner being decided by the parity of the vertices that are visited. Due to the nature of the *powerset game*, the number of moves is linked to the powerset of the states, whose size grows really quickly. It is thus necessary to represent them in a compact way, by using *symbolic moves*, which also help reducing the number of moves to consider by allowing to ignore "useless" ones. Symbolic moves are represented using logic formulas, which can also be conveniently simplified when some position becomes known to be winning for one player, thus further reducing the number of moves to consider.

The powerset game can then be solved using existing parity games algorithms to solve the problem, which can be classified as either global or local: global algorithms aim to find the winners for all vertices of the graph, while local algorithms only focus on specific vertices. We can observe that often the interest is in only one local characteristic of the solution, for example in the case of the $\mu$-calculus we might be interested in whether a specific state satisfies a formula, rather than all the states that do so. As such we will focus on a local approach, like the one experimented in [8], rather than a global approach as in [9]. Many algorithms have been developed for solving parity game, but for our work we will use a strategy iteration algorithm [10], which works by iteratively improving a strategy for one player until it becomes optimal according to the related *play profiles*. This is a global algorithm, but a local variant based on it [11] has been developed, which works by solving subgames until it can infer the winner on the full game.

Our main contribution is an adaptation of these algorithms for the powerset game. This involved performing changes both to the powerset game and the local algorithm in order to satisfy some assumptions that would otherwise not hold. For example the strategy iteration algorithm assumes a so called "total" parity game, which the powerset game is not guaranteed to be, so we found a way to convert an arbitrary parity game to a "equivalent" total one, for some definition of "equivalent" we will give. We also generalized the local algorithm to work on subgames defined by a subset of the edges of the full game, rather than a subset of the vertices, due to the powerset game lazily generating those edges. Then we provided a more flexible way to simplify symbolic formulas while keeping track of which generated moves have already been considered, which was needed due to the lazier way we generated such moves. Our work also included some optimizations and improvements that became possible thanks to solving this specific kind of game, for example by computing the play profiles of the current strategy after expanding the subgame, which would otherwise require an expensive step. On top of this we translated some of the previously mentioned problems to systems of fixpoint equations and the corresponding symbolic formulas. These were then solved using our implementation, comparing the results to the existing work in [8].

The goal will ultimately be showing that we can solve generic systems of mixed fixpoint equations over some complete lattice, highlighting the flexibility of such approach, and in a way that is faster than the existing approach, though we will not be expecting performance to be necessarily competitive with state of the art specialized solvers.

The rest of this thesis sections are organized as follows:

- Section 2 introduces all the theoretical notions which we will build up on. In particular this includes the background needed to introduce systems of fixpoint equations, parity games and the powerset game. It also includes an explanation of $\mu$-calculus and bisimilarity, along with a way to convert them to instances we can work with. Finally, it includes descriptions of the parity game algorithms we will be adapting;

- Section 3 explains how we adapted the given parity game algorithms to the powerset game and also includes various optimizations that we found for these particular instances;
- Section 4 presents the implementation of our algorithm, along with its design choices and observations on its performance;
- Section 5 summarizes our contribution in this thesis along with possible future improvements or extensions.

# 2 Background

In this chapter we give an overview of the theoretical background used in the rest of this thesis. We will first recap some basic notions on order theory with special focus on (complete) lattices. Then we will define what a system of fixpoint equations over complete lattices is and what is its solution, along with a number of related concepts in order theory. We will then give a brief introduction to parity games and describe how to characterize the solution of a system of fixpoint equations using a parity game, with some care for efficiency issues. Finally we will introduce two algorithms used to solve parity games which we will be exploiting later on.

## 2.1 Partial orders, lattices and monotone functions

We start by defining what is a (complete) lattice and introducing some related concepts. This will be fundamental for defining systems of fixpoint equations, as their domain and codomain will be lattices. Moreover we are interested in least and greatest fixpoints, which intrinsically require a concept of order.

**Definition 2.1** (partial order, poset). *Let $X$ a set. A partial order $\sqsubseteq$ is a binary relation on $X$ which satisfies the following properties for all $x, y, z \in X$:*

- *(Antisymmetry): if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$;*
- *(Reflexivity): $x \sqsubseteq x$;*
- *(Transitivity): if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$.*

*A partially ordered set (poset, for short) is a pair $(X, \sqsubseteq)$.*

A common example of poset is $(\mathbb{N}, \leq)$, the set of natural numbers, and $\leq$ is the standard order relation.

**Example 2.1** (Posets and Hasse diagrams). Posets can conveniently be visualized using *Hasse diagrams*, like the ones in Figure 1. In such diagrams lines connecting two elements represent the one on top being greater than the one on the bottom. Lines that could be obtained by transitivity are instead left implicit due to the fact that the diagram represents a valid poset.

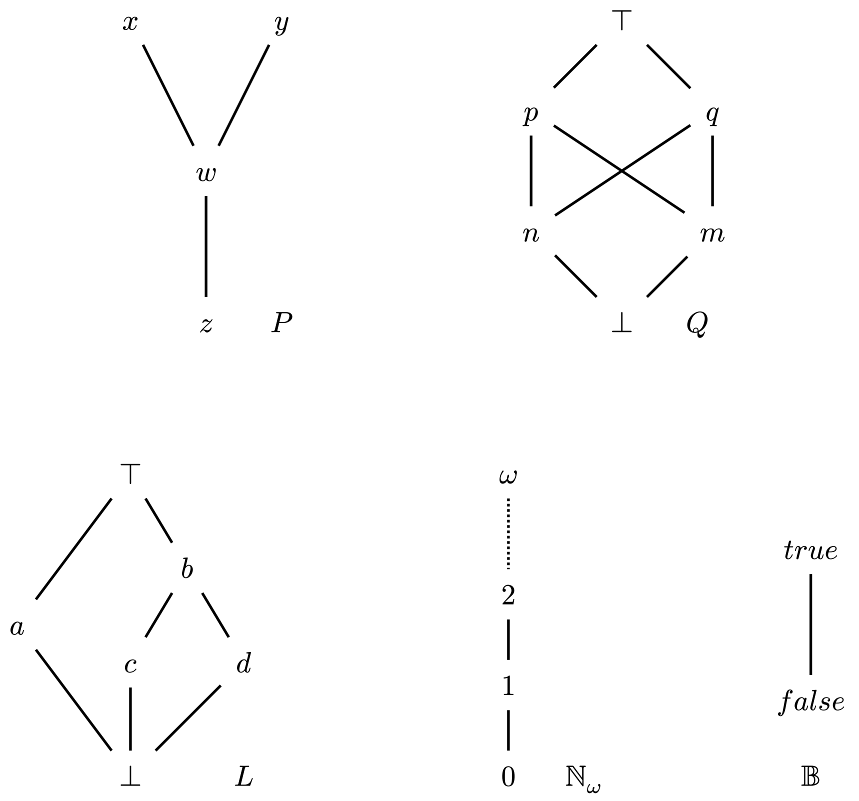

Figure 1: Hasse diagrams of five posets

**Definition 2.2** (join and meet). *Let $(X, \sqsubseteq)$ be a poset and $S \subseteq X$. The meet (respectively join) of $S$, written $\sqcap S$ (resp. $\sqcup S$), is the smallest (resp. greatest) element of $X$ that is bigger (resp. smaller) or equal to every element in $S$. Formally:*

- *(Meet): $\forall s \in S.\, s \sqsubseteq \sqcap S$ and $\forall t \in X.\, \forall s \in S.\, s \sqsubseteq t \Rightarrow \sqcap S \sqsubseteq t$*
- *(Join): $\forall s \in S.\, \sqcup S \sqsubseteq s$ and $\forall t \in X.\, \forall s \in S.\, t \sqsubseteq s \Rightarrow t \sqsubseteq \sqcup S$*

For example in Figure 1, in the poset $L$ the join between $c$ and $d$, that is $\sqcup\{c, d\}$, is $b$, while the join between $a$, $c$ and $d$ is $\sqcup\{a, c, d\} = \top$.

Meet and join do not always exist, for example in the poset $P$ the join between $x$ and $y$ does not exist because there is no element that is greater than both of them, while in the poset $Q$ the join between $n$ and $m$ does not exist because $p$, $q$ and $\top$ are all greater than both $n$ and $m$, but none of them is smaller than the others. It can however be proven that when a join or meet exists, it is unique. For our purposes we will however be interested in posets where meet and join always exists, also commonly called *lattices*. The posets $P$ and $Q$ are thus not lattices, while $L$, $\mathbb{N}_\omega$ and $\mathbb{B}$ are all lattices.

**Definition 2.3** (complete lattice). *Let $(L, \sqsubseteq)$ be a poset. It is also a lattice if meet and join exist for every pair of elements, that is given $x, y \in L$ both $\sqcap\{x, y\}$ and $\sqcup\{x, y\}$ are defined. It is a complete lattice if meet and join exist for every subset, that is given $S \subseteq L$ both $\sqcap S$ and $\sqcup S$ are defined.*

Observe that every complete lattice $L$ has a smallest element, called the *bottom* element $\bot = \sqcap\,\emptyset$, and a largest element, called the *top* element $\top = \sqcup L$. In particular, a complete lattice cannot be empty. For example in the three lattices in Figure 1 the top elements are $\top$, $\omega$ and *true*, while the bottom elements are $\bot$, $0$ and $false$, respectively.

From now on we will work with complete lattices. For most examples we will however use finite lattices, which can be proved to always be complete lattices.

**Lemma 2.1** (finite complete lattices). *Let $(L, \sqsubseteq)$ be a finite lattice, that is a lattice where $L$ is a finite set. Then it is also a complete lattice.*

In Figure 1 both $L$ and $\mathbb{B}$ are finite complete lattices. In particular $\mathbb{B}$ is called the *boolean lattice*, since it contains the two boolean literals *true* and $false$ and its join and meet operators are respectively the $\vee$ and $\wedge$ logical operators. The $\mathbb{N}_\omega$ lattice is instead an infinite complete lattice, since it contains all natural numbers equipped with a top element $\omega$. Note that the plain set of natural numbers $\mathbb{N}$ is not a complete lattice because $\sqcup\mathbb{N}$ is not defined, while in $\mathbb{N}_\omega$ it is $\omega$.

**Definition 2.4** (powerset). *Let $X$ be a set. Its powerset, written $2^X$, is the set of all subsets of $X$, that is $2^X = \{S \mid S \subseteq X\}$.*

**Example 2.2** (powerset lattice). Given a set $X$, the pair $(2^X, \subseteq)$ is a complete lattice.

The $\sqcup$ and $\sqcap$ operations are respectively the union $\cup$ and intersection $\cap$ operations on sets, while the $\top$ and $\bot$ elements are respectively $X$ and $\emptyset$.

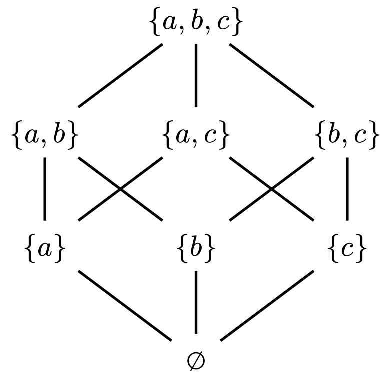

Figure 2: Hasse diagram of a powerset lattice

When we will later characterize the solutions of a system of fixpoint equations it will be convenient to consider a basis of the lattice involved. Intuitively a basis is a subset of elements which allows to express any other element as a join of a subset of such basis.

**Definition 2.5** (basis). *Let $(L, \sqsubseteq)$ be a lattice. A basis is a subset $B_L \subseteq L$ such that all elements $l \in L$, $l = \sqcup\{b \in B_L \mid b \sqsubseteq l\}$.*

To give an example of a basis, consider the lattice $L$ in Figure 1. A basis for it is the set $B_L = \{a, c, d\}$, where we can express the other elements with $\bot = \sqcup\, \emptyset$, $b = \sqcup\{c, d\}$ and $\top = \sqcup\{a, c, d\} = \sqcup\{a, c\} = \sqcup\{a, d\}$. Note that there may be more than one way to obtain an element by joining a subset of a basis, as shown with $\top$. The boolean lattice $\mathbb{B}$ instead admits the simple basis $\{true\}$, since $false = \sqcup\, \emptyset$ and $true = \sqcup\{true\}$. Another basis that we will use often is the basis of a powerset lattice, which we will now define.

**Definition 2.6** (basis of powerset). *Given a set $X$, a basis of the poset $(2^X, \subseteq)$ is the set of singletons $B_{2^X} = \{\{x\} \mid x \in X\}$.*

We now also define the concept of upward-closed set and upward-closure. This concept will become important later on.

**Definition 2.7** (upward-closed set). *Let $(X, \sqsubseteq)$ be a poset and $U \subseteq X$. $U$ is an upward-closed set if $\forall x, y \in X$, if $x \in U$ and $x \sqsubseteq y$ then $y \in U$.*

**Definition 2.8** (upward-closure). *Let $(L, \sqsubseteq)$ be a lattice and $l \in L$. The upward-closure of $l$ is $\uparrow l = \{l' \in L \mid l \sqsubseteq l'\}$.*

It can be proven that the upward-closure of a set is an upward-closed set.

**Definition 2.9** (fixpoint). *Let $(X, \sqsubseteq)$ be a complete lattice and $f : X \to X$ a function. Any element $x \in X$ such that $f(x) = x$ is a fixpoint of $f$.*

Given a function $f : L \to L$ where $(L, \sqsubseteq)$ is a complete lattice, it is not guaranteed that a fixpoint exists. However if we restrict ourself to *monotone* functions, then by the Knaster-Tarski theorem [5] there exists at least one fixpoint. Moreover the set of all fixpoints is also a complete lattice, which guarantees the existence and uniqueness the least and greatest fixpoints.

**Definition 2.10** (monotone function). *Let $(X, \sqsubseteq)$ be a poset and $f : X \to X$ a function. We say that $f$ is monotone if $\forall x, y \in X.\, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$*

**Theorem 2.1** (Knaster-Tarski [5]). *Let $(X, \sqsubseteq)$ be a complete lattice and $f : X \to X$ a monotone function. The set of fixpoint of $f$ forms a complete lattice with respect to $\sqsubseteq$.*

*The least fixpoint of $f$, written $\mu f$, is the bottom element of such lattice, while the greatest fixpoint of $f$, written $\nu f$, is the top element.*

Kleene iteration [6] also gives us a constructive way to obtain a least or greatest fixpoint by repeatedly iterating a function starting from respectively the bottom or top element of the lattice. In its most general form it may require a transfinite iteration, though with some stronger hypothesis it can be relaxed to a regular, possibly infinite, iteration, for example by requiring the function to be continuous instead of just monotone. In our case we can limit ourselves to finite lattices, in which case Kleene iteration can be shown to require only a finite amount of steps.

It should be noted however that it may not be efficient enough to compute a fixpoint in such a way, because it may require too many iterations (potentially an infinite amount in case of non-finite lattices) or because computing and representing the full solution may require too much space, and we are interested only in some specific characteristics of it.

**Theorem 2.2** (Kleene iteration [6] for finite lattices). *Let $(X, \sqsubseteq)$ be a finite lattice and $f : X \to X$ a monotone function. Consider the ascending chain $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$, it converges to $\mu f$ in a finite amount of steps, that is there exists a $k$ such that $\mu f = f^k(\bot)$. Similarly $\nu f = f^k(\top)$ for some $k$.*

## 2.2 Tuples

In order to define systems of fixpoint equations we will need to refer to multiple equations/variables/values together, and to do that we will use $n$-tuples. We now introduce some basic notions regarding tuples, along with some convenient notation for referring to them or their elements and constructing new ones.

**Definition 2.11** (set of $n$-tuples). *Let $A$ be a set. Given $n \geq 1$ the set of $n$-tuples of $A$, written $A^n$, is inductively defined as $A^0 = \{()\}$, $A^1 = A$ and $A^{n+1} = A \times A^n$.*

**Notation 2.1** ($n$-tuple). *Let $A^n$ be a set of $n$-tuples. We will refer to its elements using boldface lowercase letters, like $\boldsymbol{a}$. Given $\boldsymbol{a} \in A^n$ we will refer to its $i$-th element with the non-boldface $a_i$.*

**Notation 2.2** (concatenation). *Let $\boldsymbol{a_1}, ..., \boldsymbol{a_k}$ be either $n$-tuples or single elements of $A$. The notation $(\boldsymbol{a_1}, ..., \boldsymbol{a_k})$ represents a $n$-tuple obtained by concatenating the* *elements in the tuples $\boldsymbol{a_1}, ..., \boldsymbol{a_k}$. Single elements are considered as 1-tuples for this purpose.*

We will also often refer to intervals over natural numbers, typically in order to index the elements of a tuple.

**Notation 2.3** (range). *We will refer to the set $\{1, ..., n\}$ with the shorthand $\underline{n}$.*

Given a poset $X$ we can extend its order to $X^n$ by having the $\sqsubseteq^\wedge$ relation hold for two tuples when the $\sqsubseteq$ relation holds for all the pair of elements. This is called *pointwise order*, and we will use it often later on.

**Definition 2.12** (pointwise order). *Let $(X, \sqsubseteq)$ be a poset. The pointwise order $\sqsubseteq^\wedge$ on $X^n$ is defined, for $\boldsymbol{x}, \boldsymbol{x}' \in X^n$, by $\boldsymbol{x} \sqsubseteq^\wedge \boldsymbol{x}'$ if $\forall i \in \underline{n}.\, x_i \sqsubseteq x_i'$.*

*It can be proven that $(X^n, \sqsubseteq^\wedge)$ is also a poset. Moreover if $(X, \sqsubseteq)$ is a (complete) lattice then $(X^n, \sqsubseteq^\wedge)$ is also a (complete) lattice, where $\sqcup \boldsymbol{X} = (\sqcup X_1, \sqcup X_2, ..., \sqcup X_n)$ and similarly for $\sqcap \boldsymbol{X}$.*

## 2.3 Systems of fixpoint equations

We will now define what is a system of fixpoint equations and what is its solution, following the definition given in [7]. Intuitively this will be very similar to a normal system of equations, except for the fact that each equation is interpreted as a fixpoint equation. Since there can be more than one fixpoint we will also need to specify which kind of fixpoint the equation refers to, which we will do by using respectively the symbols $\mu$ and $\nu$ in subscript after the equal sign to denote the fact that we refer to the least or greatest fixpoint, respectively.

**Definition 2.13** (system of fixpoint equation). *Let $(L, \sqsubseteq)$ be a complete lattice. A system of fixpoint equations $E$ over $L$ is a system of the following shape:*

$$\begin{cases} x_1 =_{\eta_1} f_1(x_1, ..., x_n) \\ x_2 =_{\eta_2} f_2(x_1, ..., x_n) \\ \quad\vdots \\ x_n =_{\eta_n} f_n(x_1, ..., x_n) \end{cases}$$

*where $\forall i \in \underline{n}$, $f_i : L^n \to L$ is a monotone function and $x_i$ ranges over $L$. Each subscript $\eta_i$ must be either $\mu$ or $\nu$, representing respectively a least or a greatest fixpoint equation.*

**Notation 2.4** (system of fixpoint equations as tuple). *The above system of fixpoint equations can be written as $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$, where:*

- *$\boldsymbol{x} = (x_1, ..., x_n)$;*
- *$\boldsymbol{\eta} = (\eta_1, ..., \eta_n)$;*
- *$\boldsymbol{f} = (f_1, ..., f_n)$ but can also be seen as $\boldsymbol{f} : L^n \to L^n$ with $\boldsymbol{f}(x_1, ..., x_n) = (f_1(x_1), ..., f_n(x_n))$.*

**Notation 2.5** (empty system of fixpoint equations). *A system of equations with no equations or variables is conveniently written as $\varnothing$.*

**Definition 2.14** (substitution). *Let $(L, \sqsubseteq)$ be a complete lattice and $E$ be a system of $n$ fixpoint equations over $L$ and variables $x_i$ for $i \in \underline{n}$. Let $j \in \underline{n}$ and $l \in L$. The substitution $E\left[x_j := l\right]$ is a new system of equation where the $j$-th equation is removed and any occurrence of the variable $x_j$ in the other equations is replaced with the element $l$.*

We can now define the solution for a system of fixpoint equations recursively, starting from the last variable, which is replaced in the rest of the system by a free variable representing the fixed parameter. Then one obtains a parametric system with one equation less. This is inductively solved and its solution, which is a function of the parameter, is replaced in the last equation. This produces a fixpoint equation with a single variable, which can be solved to determine the value of the last variable.

**Definition 2.15** (solution). *Let $(L, \sqsubseteq)$ be a complete lattice and $E$ be a system of $n$ fixpoint equations over $L$ and variables $x_i$ for $i \in \underline{n}$. The solution of $E$ is $s = \mathrm{sol}(E)$, with $s \in L^n$ inductively defined as follows:*

$$\begin{aligned} \mathrm{sol}(\varnothing) &= () \\ \mathrm{sol}(E) &= (\mathrm{sol}(E[x_n := s_n]), s_n) \end{aligned}$$

*where $s_n = \eta_n(\lambda x.\, f_n(\mathrm{sol}(E[x_n := x]), x))$.*

**Example 2.3** (solving a fixpoint system). Consider the following system of fixpoint equations $E$ over the boolean lattice $\mathbb{B}$:

$$\begin{cases} x_1 =_\mu x_1 \vee x_2 \\ x_2 =_\nu x_1 \wedge x_2 \end{cases}$$

To solve this system of fixpoint equations we apply the definition of its solution, getting $\mathrm{sol}(E) = (\mathrm{sol}(E[x_2 := s_2]), s_2)$ with $s_2 = \nu(\lambda x.\, \mathrm{sol}(E[x_2 := x]) \wedge x)$. In order to find $s_2$ we will need to solve $E[x_2 := x]$, that is the system of the single fixpoint equation $x_1 =_\mu x_1 \vee x$ and parameterized over $x$. To do this we apply the definition again, getting $\mathrm{sol}(E[x_2 := x]) = (\mathrm{sol}(\varnothing), s_1)$ with $s_1 = \mu(\lambda x'.\, x' \vee x)$. At this point we have hit the base case with $\mathrm{sol}(\varnothing)$, which is just $()$, while we can find $s_1$ by solving the given fixpoint equation, getting $s_1 = x$ because $x$ is the smallest value that is equal to itself when joined with $x$. We thus get $\mathrm{sol}(E[x_2 \coloneqq x]) = (x)$, and we are back to find $s_2$, whose definition can now be simplified to $\nu(\lambda x.\, x \wedge x)$. Thus fixpoint equation can now be solved, getting $s_2 = \mathit{true}$ because $\mathit{true}$ is the greatest element of $\mathbb{B}$ that also satisfies the given equation. Finally, we can get $\mathrm{sol}(E[x_2 \coloneqq s_2]) = s_2 = \mathit{true}$ by substituting $s_2$ in place of $x$ in $\mathrm{sol}(E[x_2 \coloneqq x])$, and with this we get $\mathrm{sol}(E) = (\mathit{true}, \mathit{true})$.

To recap, the steps performed were:
- $\mathrm{sol}(E) = (\mathrm{sol}(E[x_2 \coloneqq s_2]), s_2)$ with $s_2 = \nu(\lambda x.\, \mathrm{sol}(E[x_2 \coloneqq x]) \wedge x)$
- $\mathrm{sol}(E[x_2 \coloneqq x]) = (\mathrm{sol}(\emptyset), s_1)$ with $s_1 = \mu(\lambda x'.\, x' \vee x)$
- solving $s_1$ gives $s_1 = x$
- solving $s_2$ gives $s_2 = \nu(\lambda x.\, x \wedge x) = \mathit{true}$
- $\mathrm{sol}(E) = (\mathit{true}, \mathit{true})$

Notice that the way the solution of a system of fixpoint equations is defined depends on the order of the equations. Indeed different orders can result in different solutions.

**Example 2.4** (different order of equations). Consider a system of equations $E'$ containing the same fixpoint equations as $E$ from Example 2.3, but with their order swapped:

$$\begin{cases} x_1 =_\nu x_1 \wedge x_2 \\ x_2 =_\mu x_1 \vee x_2 \end{cases}$$

This time the steps needed will be the following:
- $\mathrm{sol}(E') = (\mathrm{sol}(E'[x_2 \coloneqq s_2]), s_2)$ with $s_2 = \mu(\lambda x.\, \mathrm{sol}(E'[x_2 \coloneqq x]) \vee x)$
- $\mathrm{sol}(E'[x_2 \coloneqq x]) = (\mathrm{sol}(\emptyset), s_1)$ with $s_1 = \nu(\lambda x'.\, x' \wedge x)$
- solving $s_1$ gives $s_1 = x$
- solving $s_2$ gives $s_2 = \mu(\lambda x.\, x \wedge x) = \mathit{false}$
- $\mathrm{sol}(E') = (\mathit{false}, \mathit{false})$

Notice that $\mathrm{sol}(E) = (\mathit{true}, \mathit{true}) \neq (\mathit{false}, \mathit{false}) = \mathrm{sol}(E')$, meaning that the different order of the equations in the two systems does indeed influence the solution.

## 2.4 Applications

In this section we discus two classical verification problems, model checking behavioral properties expressed in the $\mu$-calculus and checking behavioral equivalence formalized as bisimilarity. We show that both can be seen as instances of the solution of a system of fixpoint equations.

### 2.4.1 $\mu$-calculus

The $\mu$-calculus is a propositional modal logic extended with support for least and greatest fixpoints. It was first introduced by Dana Scott and Jaco de Bakker and later developed by Dexter Kozen in [1]. Its main use is to help describing properties of (labelled) transition systems to be verified.

Consider a labelled transition system over a set of states $\mathbb{S}$, a set of actions $\mathit{Act}$ and a set of transitions $\rightarrow \subseteq \mathbb{S} \times \mathit{Act} \times \mathbb{S}$ (usually written $s \xrightarrow{a} t$ to mean $(s, a, t) \in \rightarrow$). Also, let $\mathit{Prop}$ be a set of propositions and $Var$ be a set of propositional variables. A $\mu$-calculus formula for such system is defined inductively in the following way, where $A \subseteq \mathit{Act}$, $p \in \mathit{Prop}$, $x \in Var$ and $\eta$ is either $\mu$ or $\nu$:

$$\varphi, \psi := \mathit{true} \mid \mathit{false} \mid p \mid x \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid [A]\varphi \mid \langle A \rangle \varphi \mid \eta x.\, \varphi$$

**Example 2.5** (lack of deadlocks). For example the liveness property, or lack of deadlocks, which expresses the fact that all reachable states can make at least one transition, can be expressed with the formula $\nu x.\, \langle \mathit{Act} \rangle \mathit{true} \wedge [\mathit{Act}]x$. This can be read as requiring a state $s$ to be able to make at least one transition, that is it satisfies $\langle \mathit{Act} \rangle \mathit{true}$, and that after every single possible step transition the same property should hold, that is it satisfies $[\mathit{Act}]x$, where $x$ is equivalent to the initial formula. Intuitively the fixpoint is extending the first requirement to any state reachable after a number of transitions.

The semantics of a formula are given by the set of states that satisfy the formula in an environment. Given $\rho : \mathit{Prop} \cup Var \rightarrow 2^{\mathbb{S}}$, we define:

$$\begin{aligned}
[\![\mathit{true}]\!]_\rho &= \mathbb{S} \\
[\![\mathit{false}]\!]_\rho &= \varnothing \\
[\![p]\!]_\rho &= \rho(p) \\
[\![x]\!]_\rho &= \rho(x) \\
[\![\varphi \vee \psi]\!]_\rho &= [\![\varphi]\!]_\rho \cup [\![\psi]\!]_\rho \\
[\![\varphi \wedge \psi]\!]_\rho &= [\![\varphi]\!]_\rho \cap [\![\psi]\!]_\rho
\end{aligned}
\qquad
\begin{aligned}
[\![[A]\varphi]\!]_\rho &= \left\{ s \in \mathbb{S} \mid \forall a \in A, t \in \mathbb{S}.\, s \xrightarrow{a} t \Rightarrow t \in [\![\varphi]\!]_\rho \right\} \\
[\![\langle A \rangle \varphi]\!]_\rho &= \left\{ s \in \mathbb{S} \mid \exists a \in A, t \in \mathbb{S}.\, s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]_\rho \right\} \\
[\![\mu x.\, \varphi]\!]_\rho &= \mu X.\, [\![\varphi]\!]_{\rho[x:=X]} = \bigcap \left\{ S \subseteq \mathbb{S} \mid [\![\varphi]\!]_{\rho[x:=S]} \subseteq S \right\} \\
[\![\nu x.\, \varphi]\!]_\rho &= \nu X.\, [\![\varphi]\!]_{\rho[x:=X]} = \bigcup \left\{ S \subseteq \mathbb{S} \mid S \subseteq [\![\varphi]\!]_{\rho[x:=S]} \right\}
\end{aligned}$$

We will thus say that a state $s$ satisfies a $\mu$-calculus formula $\varphi$ if it is contained in its semantics, that is if $s \in [\![\varphi]\!]_{\rho_0}$, where $\rho_0$ is initially irrelevant for all $x \in Var$ and with some fixed value for all $p \in \mathit{Prop}$.

Intuitively the $\mu$ calculus enriches the common propositional logic with the modal operators $[\_]$ and $\langle \_ \rangle$, often called respectively box and diamond, which require a formula to hold for respectively all or any state reachable by the current state through a transition with one of the given actions. On top of this fixpoints then allow to express recursive properties, that is properties that hold on a certain state and also on the states reached after certain sequences of transitions. This can be used for example to propagate some requirements across any number of transitions.

It is possible to translate $\mu$-calculus formulas into systems of fixpoint equations [12] over $2^{\mathbb{S}}$, the powerset lattice of its states. Such system can be obtained by extracting each fixpoint subformula into its own equation and replacing it with its variable, assuming that no variable is used in multiple fixpoints. Since the order of equations matter, outer fixpoints must appear later in the system of equations. It can be shown that each function in the system is monotone, and so it always admits a solution.

**Example 2.6** (fixpoint equations for a $\mu$-calculus formula). For example the formula $\mu x.\, \langle Act \rangle x \vee (\nu y.\, [a]y \wedge x)$ would be translated into the following system, where for simplicity we used formulas instead of their semantics:

$$\begin{cases} y =_{\nu} [a]y \wedge x \\ x =_{\mu} \langle Act \rangle x \vee y \end{cases}$$

### 2.4.2 Bisimilarity

Bisimilarity [2] is a binary relation on states of a labelled transition system, where two states are in the relation if they are indistinguishable by only looking at some kind of behavior. We will focus on the strong bisimilarity $\cong$, where such behavior is identified with the possible transitions from a state. Bisimilarity is usually defined in terms of bisimulations, which are also binary relations on states. For the strong bisimilarity the associated bisimulations $R$ have the following requirement:

**Definition 2.16** (bisimulation). *Let* $(\mathbb{S}, Act, \rightarrow)$ *be a labelled transition system. A relation* $R \subseteq \mathbb{S} \times \mathbb{S}$ *is a bisimulation if for all* $s, t \in \mathbb{S}$ *the following holds:*

$$(s,t) \in R \qquad \Leftrightarrow \qquad \begin{cases} \forall a, s'.\, s \xrightarrow{a} s' \Rightarrow \exists t'.\, t \xrightarrow{a} t' \wedge (s', t') \in R \\ \forall a, t'.\, t \xrightarrow{a} t' \Rightarrow \exists s'.\, s \xrightarrow{a} s' \wedge (s', t') \in R \end{cases}$$

Bisimilarity is then defined to be the largest bisimulation, that is the bisimulation that contains all other bisimulations, or equivalently the union of all bisimulations.

**Example 2.7** (fixpoint equations for a bisimilarity problem).

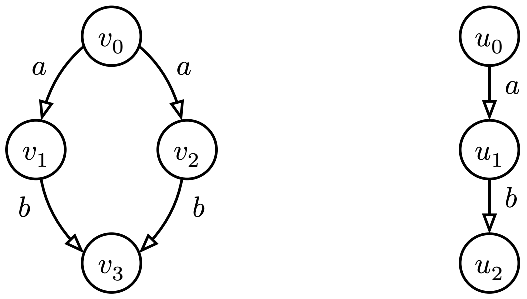

Figure 3: Example of a strong bisimilarity problem

Consider for example the two labelled transition systems given above in Figure 3. They are obviously different, but by only looking at the possible transitions it is impossible to distinguish $v_0$ from $u_0$, hence they are bisimilar. It is instead possible to distinguish $v_1$ from $u_0$, because the former can perform one transition with action $b$ while the latter can only perform a transition with action $a$, and thus they are not bisimilar.

For our purposes however there is an alternative formulation based on a greatest fixpoint. We can in fact define the following function $F : 2^{\mathbb{S}\times\mathbb{S}} \to 2^{\mathbb{S}\times\mathbb{S}}$ over the powersets of binary relations over between states:

$$\begin{aligned} F(R) = \Big\{(s,t) \in R \mid \Big(&\forall a, s'.\, s \overset{a}{\to} s' \Rightarrow \exists t'.\, t \overset{a}{\to} t' \wedge (s',t') \in R\Big) \\ \wedge \Big(&\forall a, t'.\, t \overset{a}{\to} t' \Rightarrow \exists s'.\, s \overset{a}{\to} s' \wedge (s',t') \in R\Big)\Big\} \end{aligned}$$

$F$ can be thought as "refining" a relation by ensuring that the bisimulation property holds for another step. This can be shown to be a monotonic operation, guaranteeing the existence of at least one fixpoint, including for our purposes the greatest fixpoint. Bisimulations can then be seen as the post-fixpoints of $F$, since for them the bisimulation property always holds after any amount of steps and thus no pair needs to be removed to make the property hold. Bisimilarity, being the greatest bisimulation, is thus the greatest fixpoint of $F$.

$$\cong \; = \nu R.\, F(R)$$

## 2.5 Parity games

Parity games [13, 14] are games with two players, typically denoted by 0 and 1 and referred as the existential and universal players, performed on directed graphs. A token is placed in a position, represented by nodes, and the two players move it along the edges of the graph. The set of nodes is partitioned in two sets and the player that chooses the move is determined by the subset in which the node for the current position is in. Each node is also associated with a *priority*, usually represented by a natural number. A maximal sequence of positions visited in a game is called a *play*. A play can be finite or infinite, depending on whether a position with no moves is reached or not. In case of a finite play the player who cannot move loses, otherwise if the play is infinite the priorities of the positions that are visited infinitely many times are considered: if the largest one is even then player 0 wins, otherwise player 1 is the winner. Players are also sometimes called $\exists$ and $\forall$ or $\Diamond$ and $\Box$ due to their meaning when using parity games in relation to $\mu$-calculus or fixpoint equations.

**Example 2.8** (parity game). In Figure 4 we can see an example of a parity game with 5 vertices. Circles represent vertices controlled by player 0 while squares represent vertices controlled by player 1. Each vertex is shown with its name and its priority. The vertices have been divided in two groups based on the winner on the vertices in them. The left one is winning for player 0 because from $v_0$ it can always go downwards to $v_1$, from which the only possible response possible for player 1 is to go back to $v_0$. Player 0 can thus force such play in which the higher infinitely visited priority is 2, hence the vertices are winning for player 0. In the right group a similar situation happens where player 1 can force any play to go through vertex $v_3$ infinitely often and thus winning the game. Notice that the edges from $v_0$ to $v_2$ and from $v_2$ to $v_1$ are never a good choice for the players, since they lead from a vertex that is winning for the player to one that is losing.

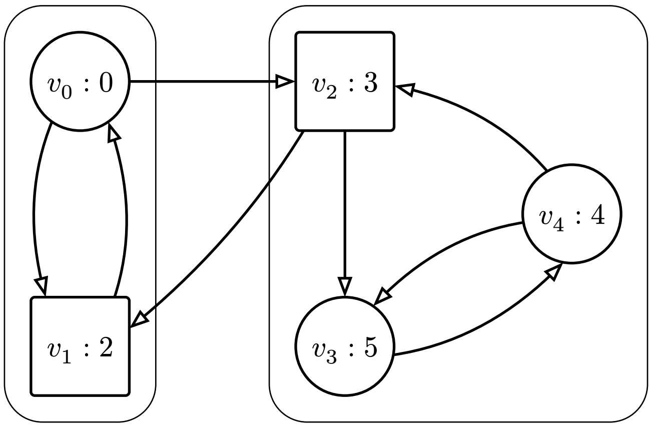

Figure 4: Example of a parity game

We will now introduce graphs and some convenient notation for them. Moreover we will also need a formal notion for infinitely recurring elements in a sequence in order to describe the winner of a parity game.

**Notation 2.6** (graph, successors and predecessors). *A simple graph is a pair $(V, E)$ where $V$ is the set of vertices and $E \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$ is the set of edges. It is called finite if $V$ is finite.*

*Given $u, v \in V$ we write $uEv$ if $(u, v) \in E$, that is if the graph contains an edge from $u$ to $v$. We also write $uE$ to denote the set of successors of $v$ in $G$, i.e. $\{v \in V \mid uEv\}$, and $Ev$ to denote the set of predecessors of $v$, i.e. $\{u \in V \mid uEv\}$.*

**Definition 2.17** (sink vertices). *Let $G = (V, E)$ be a graph. The set of sink vertices is $S_G = \{vE = \varnothing\}$, that is the set of vertices without successors.*

**Definition 2.18** (infinitely recurring elements). *Let $\pi = v_0 v_1 v_2 ...$ an infinite sequence of elements. We define* $\inf(\pi)$ *as the set of infinitely recurring elements of* $\pi$*, that is* $\inf(\pi) = \{v \mid \forall n.\, \exists i \geq n.\, v_i = v\}$.

We can now introduce parity games, which consist of a graph partitioned into two set of vertices, representing the positions controlled by each player, along with a priority function.

**Definition 2.19** (parity graph, parity game). *A parity graph is a triple $G = (V, E, p)$ where $(V, E)$ is a finite graph and $p : V \to \mathbb{N}$ is a so called priority function. A parity graph is a triple $G = (V, E, p)$. Let $V$ be partitioned into two sets $V_0$ and $V_1$. The tuple $G = (V_0, V_1, E, p)$ is a parity game.*

A particular game played on a parity game is called a *play*. Each play starts with the token on a given vertex and proceeds by moving the token to one of the successors of the current vertex, as chosen by the player controlling it. A play can eventually reach a vertex which has no successors, in which case the player controlling that vertex loses the play. Otherwise, the play can be infinite, in which case the winner of the play is determined by the highest priority of the vertices that are visited infinitely often: if that is even the winner is player 0, otherwise it is player 1.

**Definition 2.20** (play). *Let $G = (V_0, V_1, E, p)$ be a parity game. A play in $G$ from a vertex $v_0 \in V_0 \cup V_1$ is a potentially infinite sequence $\pi = v_0 v_1 ...$ such that $\forall i.\, v_i E v_{i+1}$. If the play is finite, that is $\pi = v_0 v_1 ... v_n$, then $v_n \in S_G$ is required.*

**Definition 2.21** (winner of a play). *Let $G = (V_0, V_1, E, p)$ be a parity game and let $\pi = v_0 v_1 ...$ be a play. The winner of $\pi$ is:*

- *if $\pi$ is finite, that is $\pi = v_0 v_1 ... v_n$ with $v_n \in V_i$ then the winner is player $1 - i$;*
- *if $\pi$ is infinite then consider* $\max \inf(p(v_0)p(v_1)...)$*: if it is even the winner is player 0, otherwise it is player 1.*

From now on we will focus on a subclass of parity games, which for convenience we will call *bipartite parity games* and *total parity games.* Bipartite parity games are games whose graph is bipartite, forcing players to perfectly alternate their moves. Total parity games instead require every vertex to have at least one successor, thus forcing every play to be infinite.

The parity games we will generate will be bipartite by construction, though not necessarily total. We will however mostly deal with total parity games since, as we will show, we can convert any parity game to a "compatible" total parity game.

**Definition 2.22** (bipartite parity game). *Let $G = (V_0, V_1, E, p)$ be a parity game. It is called bipartite if the graph $(V_0, V_1, E)$ is bipartite, that is $\forall v \in V_i.\, vE \cap V_i = \varnothing$.*

**Definition 2.23** (total parity game). *Let $G = (V_0, V_1, E, p)$ be a parity game. It is called total if $\forall v \in V_0 \cup V_1 .\, v \notin S_G$, that is there is no sink vertex.*

### 2.5.1 Strategies

By the well-known determinacy of parity games [13, 14] we know that each vertex is winning for exactly one of the two players, that is that player can force every play to be winning for them. Moreover it is known that the winning player also has a so-called memoryless winning strategy, that is a way to choose the next vertex in the play without considering the previous ones such that any resulting play is winning for them. From now on we will focus only on strategies and plays induced by strategies, as they are finite and easier to reason about.

**Definition 2.24** (strategy). *Let $G = (V_0, V_1, E, p)$ be a parity game. A (memoryless) strategy for player $i$ is a function $\sigma : V_i \setminus S_G \to V$ such that $\forall v.\, \sigma(v) \in vE$.*

Strategies for player 0 will usually be denoted by $\sigma$ while those for player 1 by $\tau$.

It is also worth mentioning that the domain of a strategy for player $i$ on a total parity game will be exactly $V_i$, since the set of sink vertices $S_G$ will be empty.

**Definition 2.25** (strategy induced instance). *Let $G = (V_0, V_1, E, p)$ be a parity game, $\sigma$ be a strategy for player 0 and $\tau$ be a strategy for player 1. An instance of the game $G$ induced by the strategies $\sigma$ and $\tau$ is a tuple $(G, \sigma, \tau)$.*

*Given a starting vertex $v_0 \in V_0 \cup V_1$, an instance also uniquely defines a play, called an induced play, where if $v_i \in S_G$ then the play is finite and stops at $v_i$, otherwise $v_{i+1} = \sigma(v_i)$ if $v_i \in V_0$ and $v_{i+1} = \tau(v_i)$ if $v_i \in V_1$.*

It can be proven that if an induced play is infinite then it will eventually reach a cycle and repeatedly visit those vertices in the same order, that is the play will be of the kind $v_0 ... v_k v_{k+1} ... v_n v_{k+1} ... v_n ...$.

**Definition 2.26** (winning strategy). *Let $G = (V_0, V_1, E, p)$ be a parity game. A strategy $\sigma_i$ for player $i$ is called winning on vertex $v$ if for any strategy $\sigma_{1-i}$ for the opposing player, the induced play starting from vertex $v$ in the instance $(G, \sigma_0, \sigma_1)$ is winning for player $i$.*

**Lemma 2.2** (determinacy of parity games). *Given a parity game $G = (V_0, V_1, E, p)$, for every vertex $v \in V_0 \cup V_1$ one and only one of the players can force a winning play from $v$. The set of vertices $V$ can thus be partitioned in two* ***winning sets*** *$W_0$ and $W_1$ of the vertices where player 0 (resp. player 1) has a winning strategy starting from vertices in that set.*

**Example 2.9** (strategy). For example in the parity game in Figure 5 the winning strategy, represented as whole lines, for player 0 on vertex $v_0$ would be going to the vertex $v_1$, while for player 1 on vertex $v_2$ it would be going to the vertex $v_3$. For all the other vertices the strategy is not relevant, since it will always be losing for their controlling player, so it has not been displayed.

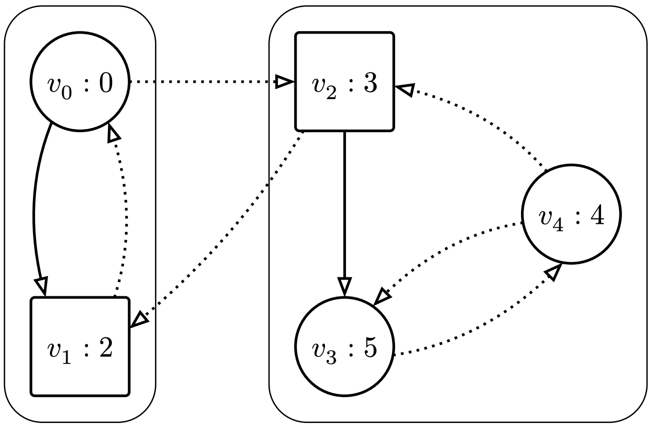

Figure 5: Example of a parity game along with its winning strategies

## 2.6 Game characterization of the solution of systems of equations

### 2.6.1 Game definition

The solution of systems of fixpoint equations can be characterized using a parity game [7], also called a powerset game. This characterization in particular allows to determine whether some element of a basis is under the solution for one of the variables of the system. This makes sense because in practice the actual solution of the system may include lot of informations we are not interested about, for example for the $\mu$-calculus it would include all the states that satisfy the given formula, while we might be only interested in knowing whether one particular state is included, or for bisimilarity it would include all pairs of processes that are bisimilar, when again we are only interested in a single pair.

**Definition 2.27** (powerset game). *Let $(L, \sqsubseteq)$ be a complete lattice and $B_L$ a basis of $L$. Let $E = \boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ be a system of $n$ fixpoint equations.*

*The powerset game is a parity game associated with $E$ defined as follows:*

- *the vertices for player 0 are $V_0 = B_L \times \underline{n} = \{(b, i) \mid b \in B_L \land i \in \underline{n}\}$*
- *the vertices for player 1 are $V_1 = (2^{B_L})^n = \{(X_1, ..., X_n) \mid X_i \in 2^{B_L}\}$*
- *the moves from player 0 vertices are $E(b, i) = \{\boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^n \land b \sqsubseteq f_i(\sqcup \boldsymbol{X})\}$*
- *the moves from player 1 vertices are $A(\boldsymbol{X}) = \{(b, i) \mid i \in \underline{n} \land b \in X_i\}$*

- *the priority function is defined such that:*
  - $p(\boldsymbol{X}) = 0$*;*
  - $p((b,i))$ *is even if* $\eta_i = \nu$ *and odd if* $\eta_i = \mu$*;*
  - $p((b,i)) < p((b',j))$ *if* $i < j$.

Intuitively each vertex $(b,i)$, owned by player 0, represents the fact that the basis element $b$ is under the $i$-th component of the solution. Its moves then are all the possible assignments to the tuple of variables $\boldsymbol{x}$. These are expressed as tuples of subsets $X_1, ..., X_n$ of the basis, with the requirement that $b$ is under the result of $f_i(\sqcup X_1, ..., \sqcup X_n)$. Player 1 then can challenge player 0 by claiming that one of those subsets contains an element of the basis that is not actually under the solution, and this continues either infinitely or until one of the two players has no move possible.

The priority function is not fully specified, but it can be shown that there exist a mapping to $\mathbb{N}$ that satisfies the given order and partition into even/odd. An intuitive way would be to just list the priorities in order and give to map each of them to the next available even or odd natural number.

It has been proven in [7] that such characterization is both correct and complete, allowing us to solve generic systems of fixpoint equations with it.

**Theorem 2.3** (correctness and completeness of the powerset game). *Let $E$ be a system of $n$ fixpoint equations over a complete lattice $L$ with solution $s$. For all $b \in B_L$ and $i \in \underline{n}$, we have $b \sqsubseteq s_i$ if and only if the player 0 has a winning strategy on the powerset game associated to $E$ starting from the vertex $(b,i)$.*

**Example 2.10** (game characterization). Consider for example the system of equations given in Example 2.3 over the boolean lattice $\mathbb{B}$:

$$\begin{cases} x_1 =_\mu x_1 \lor x_2 \\ x_2 =_\nu x_1 \land x_2 \end{cases}$$

The corresponding game characterization would be the following:

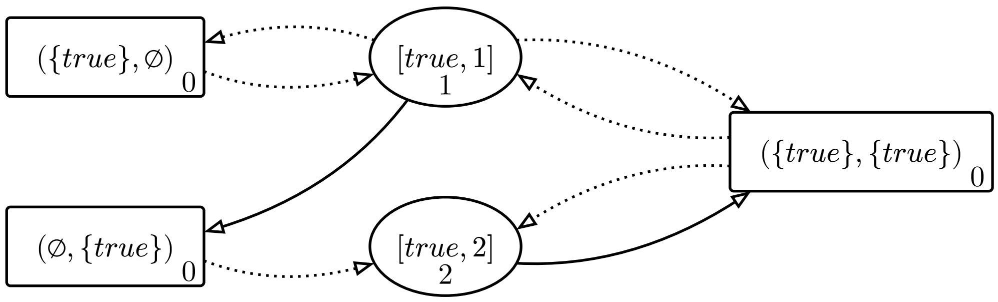

Figure 6: Example of a game characterization

As before, elliptic vertices represent player 0 positions while rectangular vertices represent player 1 positions. The priorities are now represented with the numbers on the bottom center or right, while the non-dotted edges correspond to the winning strategies.

The way this is obtained is by starting with the player 0 positions, which are the ones we care about, since if one wanted to prove whether *true* is under the solution for $x_1$ or $x_2$ they would have to check whether $[\mathit{true}, 1]$ or $[\mathit{true}, 2]$ are winning or not. From those vertices we then have the following moves:

$$E(\mathit{true}, 1) = \{\ (\{\mathit{true}\}, \emptyset),\ \ (\emptyset, \{\mathit{true}\}),\ \ (\{\mathit{true}\}, \{\mathit{true}\})\ \}$$
$$E(\mathit{true}, 2) = \{\ (\{\mathit{true}\}, \{\mathit{true}\})\ \}$$

Note that the remaining position of player 1 $(\emptyset, \emptyset)$ is not reachable, and thus was omitted from the figure.

The game is ultimately won by player 0 on every position, since it can force every play to go through the position $[\mathit{true}, 2]$ over and over. This position has the highest priority, at 2, thus being the highest of every play, and since it is even it makes player 0 the winner. Hence we can infer that $\mathit{true} \sqsubseteq x_1^*$ and $\mathit{true} \sqsubseteq x_2^*$, which implies $x_1^* = \mathit{true}$ and $x_2^* = \mathit{true}$.

One can also see that swapping the equations would result in the same parity graph, except for position $[\mathit{true}, 1]$ which now would have a higher odd priority than $[\mathit{true}, 2]$. This makes the game losing for player 0 on all positions, since player 1 can force every play to go through $[\mathit{true}, 1]$ and win. We thus get $\mathit{true} \not\sqsubseteq x_1^*$ and $\mathit{true} \not\sqsubseteq x_2^*$, which imply $x_1^* = \mathit{false}$ and $x_2^* = \mathit{false}$, like we already saw in Example 2.4.

### 2.6.2 Selections

In practice it is not convenient to consider all the possible moves for player 0. For instance in Example 2.10 the move from $[\mathit{true}, 1]$ to $(\{\mathit{true}\}, \{\mathit{true}\})$ is never convenient for player 0, since the moves to $(\{\mathit{true}\}, \emptyset)$ and $(\emptyset, \{\mathit{true}\})$ would give player 1 strictly less choices. In fact going from $[\mathit{true}, 2]$ to $(\{\mathit{true}\}, \{\mathit{true}\})$ would be a losing move for player 0, and the only way to win is to go to $(\emptyset, \{\mathit{true}\})$. In general, for player 0, it will be always convenient to play moves consisting of sets of elements with limited cardinality and as small as possible in the order. We will now see a formalization of this idea.

To start we will need to consider a new order, called *Hoare preorder*:

**Definition 2.28** (Hoare preorder). *Let $(P, \sqsubseteq)$ be a poset. The Hoare preorder, written $\sqsubseteq_H$, is a preorder on the set $2^P$ such that, $\forall X, Y \subseteq P.\, X \sqsubseteq_H Y \Leftrightarrow \forall x \in X.\, \exists y \in Y.\, x \sqsubseteq y$.*

We also consider the pointwise extension $\sqsubseteq_H^\wedge$ of the Hoare preorder on the set $(2^{B_L})^n$, that is $\forall X, Y \in (2^{B_L})^n, \boldsymbol{X} \sqsubseteq_H^\wedge \boldsymbol{Y} \Leftrightarrow \forall i \in \underline{n}.\, X_i \sqsubseteq_H Y_i$, and the upward-closure with respect to it, that is given $T \subseteq (2^{B_L})^n$ then $\uparrow_H T = \{\boldsymbol{X} \mid \exists \boldsymbol{Y} \in T \wedge \boldsymbol{Y} \sqsubseteq_H^{\wedge} \boldsymbol{X}\}$.

The idea will then be for player 0 to avoid playing a move $\boldsymbol{Y}$ if there exist another move $\boldsymbol{X}$ such that $\boldsymbol{X} \sqsubset_H^{\wedge} \boldsymbol{Y}$. More formally, it can be proven that any set of moves whose upward-closure with respect to $\sqsubseteq_H^{\wedge}$ is equal to $E(b, i)$ is equivalent to it for the purpose of the game. That is, we can replace the moves for that player 0 position and it would not change the winners compared to the original game. We call such sets of moves *selections*, and a point of interest will be finding small selections in order to reduce the size of the game.

**Definition 2.29** (selection). *Let $(L, \sqsubseteq)$ be a lattice. A selection is a function $\sigma : (B_L \times \underline{n}) \to 2^{(2^{B_L})^m}$ such that $\forall b \in B_L, i \in \underline{n}. \uparrow_H \sigma(b, i) = E(b, i)$.*

### 2.6.3 Logic for upward-closed sets

Ideally we would be interested in the least selection; this can be shown to always exist in finite lattices, but not in infinite ones. Moreover when it exists it might be exponential in size.

**Example 2.11** (least selection may not exist). Consider for example the complete lattice $\mathbb{N}_\omega$ seen in Figure 1, and consider a system of fixpoint equations with only the equation $x =_\mu f(x)$ where $f(n) = n + 1$ if $n \in \mathbb{N}$ and $f(\omega) = \omega$. We will pick the lattice itself as its basis and we will want to prove $\omega \sqsubseteq x^*$ with $x^*$ being the solution of this equation. This will generate a powerset game starting from position $\omega$ with moves $E(\omega)$, for which it can be shown that the move $\mathbb{N}$ is winning for player 0. We are however interested in selections for $E(\omega)$, and it can be shown that any $\{X\}$ where $X \subseteq \mathbb{N}$ and $X$ is infinite is a valid selection for $E(\omega)$. In fact $\omega \sqsubseteq f(\sqcup X)$ can only be satisfied if $f(\sqcup X) = \omega$ and thus $\sqcup X = \omega$, which is true for all and only the infinite $X$. There are however infinitely many such sets, and there is no smallest one, since it is always possible to get a smaller one by removing one element. Thus there cannot be a smallest selection.

**Example 2.12** (least selection can be exponential). The least selection can be exponential with respect to the number of variables and basis size. Take for example the function $f(x_1, ..., x_{2n}) = (x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge ... \wedge (x_{2n-1} \vee x_{2n})$ over the boolean lattice. The corresponding minimal selection would be $\{(\{true\}, \emptyset, \{true\}, \emptyset, ...), ..., (\emptyset, \{true\}, \emptyset, \{true\}, ...)\}$, which lists all the ways to satisfy each $x_{2i-1} \vee x_{2i}$ without making them both *true*, which is $2^n$ and thus exponential in the number of variables. A similar construction can be made for the basis size, by taking as domain the set of $n$-tuples over the boolean lattice.

For these reasons a logic for upward-closed sets is used to represent the $E(b, i)$ set in a more compact way. Additionally this allows us to generate relative selections which are typically small, even if they are not the least ones. From now on we will refer to formulas in such logic with "logic formulas".

**Definition 2.30** (logic for upward-closed sets). *Let $(L, \sqsubseteq)$ be a complete lattice and $B_L$ a basis of $L$. Given $n \in \mathbb{N}$ we define the following logic, where $b \in B_L$ and $i \in \underline{n}$:*

$$\varphi := [b, i] \mid \bigwedge_{k \in K} \varphi_k \mid \bigvee_{k \in K} \varphi_k$$

The formulas *true* and *false* are then implicitly defined as $\wedge_{k \in \varnothing} \varphi_k$ and $\vee_{k \in \varnothing} \varphi_k$.

We now give the semantics of a logic formula, which consist in the set of moves that the formula is representing. We will be interested in formulas whose semantics will be equal to the set $E(b, i)$.

**Definition 2.31** (logic formulas semantics). *Let $(L, \sqsubseteq)$ be a complete lattice, $B_L$ a basis of $L$, $n \in \mathbb{N}$, $i \in \underline{n}$ and $\varphi$ a logic formula. The semantics of $\varphi$, that is the set of player 1 vertices is represents, is a upward-closed set $[\![\varphi]\!] \subseteq (2^{B_L})^n$ with respect to $\sqsubseteq_H^\wedge$, define as follows:*

$$\begin{aligned} [\![[b, i]]\!] &= \{\boldsymbol{X} \mid b \in \boldsymbol{X}_i\} \\ [\![\bigwedge_{k \in K} \varphi_k]\!] &= \bigcap_{k \in K} [\![\varphi_k]\!] \\ [\![\bigvee_{k \in K} \varphi_k]\!] &= \bigcup_{k \in K} [\![\varphi_k]\!] \end{aligned}$$

Given a logic formula we can however define a generator for symbolic moves, which is a selection for the set represented by the logic formula semantics. This will be the set of moves that we will use in practice when solving the parity game.

**Definition 2.32** (generator for symbolic moves). *Let $(L, \sqsubseteq)$ be a complete lattice, $B_L$ a basis of $L$, $n \in \mathbb{N}$, $i \in \underline{n}$ and $\varphi$ a logic formula. The moves generated by $\varphi$, written $M(\varphi)$ are:*

$$\begin{aligned} M([b, i]) &= \{\boldsymbol{X}\} \text{ with } X_i = \{b\} \text{ and } \forall j \neq i.\, X_j = \varnothing \\ M(\bigwedge_{k \in K} \varphi_k) &= \left\{ \bigcup X \mid X \in \prod_{k \in K} M(\varphi_k) \right\} \\ M(\bigvee_{k \in K} \varphi_k) &= \bigcup_{k \in K} M(\varphi_k) \end{aligned}$$

Another advantage of representing selections using such formulas is that they can be simplified when it becomes known that some position $[b, i]$ for player 0 is winning or losing. This corresponds to the assigning either true or false to the atom $[b, i]$ in the formula and propagating that through the operators it is contained in. In the parity game this would translate to either removing some moves for player 0, due to them being winning for player 1, or replacing ,moves for player 0 moves with a smaller number of them that do not give player 1 the option to reach positions that are winning for player 0. This is already exploited in the existing implementation of the symbolic algorithm [8] to potentially remove lot of edges at once, thus simplifying the game, while preserving the winners on all positions.

An alternative intuition for this logic is to see it as splitting the variables $x_i$ and functions $f_i$ of the original system of equations into multiple $x_{ib_1}, x_{ib_2}, ...$ and $f_{ib_1}, f_{ib_2}, ...$, with one for each element of the basis. Logic formulas then represent these functions, while moves for an original position $(b, i)$ for player 0 can be seen as partial assignments to the new boolean variables such that $f_{ib}\left(x_{1b_1}, ...\right)$ is true. Finally, generating symbolic moves is equivalent to extracting approximately minimal assignments from the formulas representing the functions. We will leave further exploration of this topic to the future, but for our purposes it still provides a nice intuition for how to define logic formulas for our implementations.

### 2.6.4 Translating $\mu$-calculus formulas

As seen in Section 2.4.1, $\mu$-calculus formulas can be translated into systems of fixpoint equations. The functions appearing in such systems can also be automatically translated into logic formulas for upward-closed sets. Consider a system of fixpoint equations generated by a $\mu$-calculus formula:

$$\begin{cases} x_1 =_{\eta_1} \varphi_1(x_1, ..., x_n) \\ \quad \vdots \\ x_n =_{\eta_n} \varphi_n(x_1, ..., n_n) \end{cases}$$

We need to define a logic formula representing the moves for player 0 for each vertex $(b, i)$ for a basis element $b$ and a variable index $i$. Recall that the system of equations is defined over $2^{\mathbb{S}}$, the powerset lattice of its states, while the basis is $B_{2^{\mathbb{S}}}$, consisting of singletons, given in Definition 2.6. We thus need to define a formula for each state $s$ and index $i$ such that the formula is true when the state $s$ satisfies the formula $\varphi_i(\boldsymbol{x}^*)$, with $\boldsymbol{x}^*$ representing the actual solution of the system. Moreover we are allowed to refer to any vertex controlled by player 0 in this formula, which is equivalent to being able to require that any another state $s'$ satisfies one of the formulas $\varphi_j(\boldsymbol{x}^*)$.

We can then define the logic formula for the vertex $(s, i)$ as $F(s, \varphi_i(x_1, ..., x_n))$, where $F$ is in turn defined by structural induction on its second argument:

$$
\begin{aligned}
F(s, true) &= true & F(s, x_i) &= [b, i] \\
F(s, false) &= false & F(s, \langle A \rangle \psi) &= \bigwedge_{a \in [\![A]\!]} \bigwedge_{s \xrightarrow{a} t} F(t, \psi) \\
F(s, p) &= \begin{cases} true & \text{if } s \in \rho_0(p) \\ false & \text{if } s \notin \rho_0(p) \end{cases} & F(s, [A]\psi) &= \bigvee_{a \in [\![A]\!]} \bigvee_{s \xrightarrow{a} t} F(t, \psi) \\
F(s, \psi_1 \vee \psi_2) &= F(s, \psi_1) \vee F(s, \psi_2) & F(s, \psi_1 \wedge \psi_2) &= F(s, \psi_1) \wedge F(s, \psi_2)
\end{aligned}
$$

It is interesting to note that the cases for $\langle A \rangle \psi$ and $[A]\psi$ are effectively taking the respective semantics definition, which use universal and existential quantifiers, and translating them to finite sequence of respectively conjunctions and disjunctions between the elements that satisfy such quantifiers.

The definition also did not include fixpoint formulas since those were already been removed when translating to a system of fixpoint equations.

### 2.6.5 Translating bisimilarity

Likewise for bisimilarity we have seen in Section 2.4.2 that it can be translated to a fixpoint equation, which in turn can be seen as a system of a single fixpoint equation. As with $\mu$-calculus formulas the domain is the powerset lattice $2^{\mathbb{S} \times \mathbb{S}}$, and thus its basis is $B_{2^{\mathbb{S} \times \mathbb{S}}}$, which can also be expressed as $\{\{(s_1, s_2)\} \mid s_1, s_2 \in \mathbb{S}\}$. Since there is just one variable and equation we will only define $F(s_1, s_2)$, representing the formula for the player 0 vertex $((s_1, s_2), 1)$:

$$
F(s_1, s_2) = \left( \left( \bigwedge_{a \in Act} \bigwedge_{s_1 \xrightarrow{a} t_1} \bigvee_{s_2 \xrightarrow{a} t_2} [(t_1, t_2), 1] \right) \wedge \left( \bigwedge_{a \in Act} \bigwedge_{s_2 \xrightarrow{a} t_2} \bigvee_{s_1 \xrightarrow{a} t_1} [(t_1, t_2), 1] \right) \right)
$$

**Example 2.13** (logic formulas for bisimilarity). For example the formulas for the pair of states in the labelled transition systems shown in Figure 3 are the following:

$$
\begin{aligned}
F(v_0, u_0) &= ([(v_1, u_1), 1] \wedge [(v_2, u_1), 1]) \wedge ([(v_1, u_1), 1] \vee [(v_2, u_1), 1]) \\
&= [(v_1, u_1), 1] \wedge [(v_2, u_1), 1] \\
F(v_0, u_1) &= false \wedge false = false \\
F(v_0, u_2) &= false \wedge true = false \\
\\
F(v_1, u_0) &= false \wedge false = false \\
F(v_1, u_1) &= [(v_3, u_2), 1] \wedge [(v_3, u_2), 1] = [(v_3, u_2), 1] \\
F(v_1, u_2) &= false \wedge true = false \\
\\
F(v_2, u_0) &= false \wedge false = false \\
F(v_2, u_1) &= [(v_3, u_2), 1] \wedge [(v_3, u_2), 1] = [(v_3, u_2), 1] \\
F(v_2, u_2) &= false \wedge true = false
\end{aligned}
$$

$$F(v_3, u_0) = true \land false = false$$
$$F(v_3, u_1) = true \land false = false$$
$$F(v_3, u_2) = true \land true = true$$

### 2.6.6 Translating parity games

It is known that parity games can also be translated to nested fixpoints [15], which in turn are equivalent to systems of fixpoint equations. We will later use this fact to generate simple problems for testing our implementation.

In particular, given a parity game $G = (V_0, V_1, E, p)$ we can define a system of fixpoint equations on the boolean lattice $\mathbb{B}$, where *true* represents a vertex being winning for player 0 while *false* is winning for player 1. Then for each vertex $v \in V_0 \cup V_1$ a variable $x_v$ will be defined along with the following equation:

$$x_v =_\eta \begin{cases} \bigsqcup_{u \in vE} x_u & \text{if } v \in V_0 \\ \bigsqcap_{u \in vE} x_u & \text{if } v \in V_1 \end{cases} \qquad \text{with } \eta = \begin{cases} \nu & \text{if } p(v) \text{ even} \\ \mu & \text{if } p(v) \text{ odd} \end{cases}$$

Intuitively, a vertex in $V_0$ is winning for player 0 if any of its successors is also winning for them, because they can choose to move to that successor and keep winning. Meanwhile, a vertex in $V_1$ is winning for player 0 if all its successors are winning for them, because otherwise player 1 would have the option to move to any successor that is not winning for player 0 and win.

The priority of vertices must however also be taken into account in order to determine the winner of infinite plays, which we can reduce to plays ending with a cycle. If one happens the last equation corresponding to a vertex of the cycle will have both *true* and *false* as fixpoint, and will thus decide the winner for the entire cycle, hence why equations corresponding with vertices with higher priorities have to be sorted last. The winner is then chosen by whether the fixpoint equation is a greatest fixpoint or a least fixpoint: if it is a greatest fixpoint the solution will be *true* and player 0 will win, otherwise it will be *false* and player 1 will win. This is the reason why the fixpoint type was chosen according to the priority of the vertex: if it is even then player 0 wins the cycle in the parity game and hence the equation must be a greatest fixpoint, otherwise player 1 wins and the equation must be a least fixpoint.

These functions can be trivially converted to logic formulas. Notice that the atom $(true, i)$, where $i$ is the index of the equation with variable $x_u$, is true if and only if the solution for $x_u$ is *true*, otherwise if the atom is false then the solution is *false*. As such the equations of the system can be converted to logic formulas by replacing each variable $x_u$ with the atom $(true, i)$, where $i$ is the index of variable the $x_u$, each $\sqcup$ with $\lor$ and each $\sqcap$ with $\land$.

**Example 2.14** (translation and logic formulas for a parity game). For example the parity game in Figure 4 would be translated to the following system of fixpoint equations:

$$\begin{cases} v_0 =_\nu v_1 \sqcup v_2 \\ v_1 =_\nu v_0 \\ v_2 =_\mu v_1 \sqcap v_3 \\ v_4 =_\nu v_2 \sqcup v_3 \\ v_3 =_\mu v_4 \end{cases}$$

Which can then be translated to the following formulas:

$$\begin{aligned} F(true, 1) &= [true, 2] \vee [true, 3] \\ F(true, 2) &= [true, 1] \\ F(true, 3) &= [true, 2] \wedge [true, 5] \\ F(true, 4) &= [true, 3] \vee [true, 5] \\ F(true, 5) &= [true, 4] \end{aligned}$$

## 2.7 Local strategy iteration

### 2.7.1 Strategy iteration

Strategy iteration [10] is one of the oldest algorithms that computes the winning sets and the optimal strategies for the two players of a bipartite and total parity game. The algorithm starts with a strategy for player 0 and repeats *valuation* phases, during which it computes a *play profile* for each vertex, and *improvement* phases, during which it uses such play profiles to improve the strategy. This continues until the strategy can no longer be improved, at which point it is guaranteed to be optimal.

We will start introducing some concepts that will help characterize how favorable a vertex is for a given player. We will start by giving the definition of a *relevance ordering*, which is a total order over the vertices where bigger vertices correspond to bigger priorities. This will be important in determining which vertices are more impactful on the winner of a play. We then define the sets of *positive and negative vertices*, which are a different way to partition the set of vertices. In particular the set of positive vertices contains vertices whose priority is even, and thus more favorable to player 0, while the negative vertices will be those with odd priority. We also introduce a *reward ordering*, which instead expresses how favorable to player 0 a vertex is. In particular a positive vertex has a bigger reward than a negative one. Positive vertices are also more rewarding if they have a bigger priority, while negative vertices are less rewarding in that case. Finally, the reward ordering is extended to sets of vertices, where the reward of the most relevant vertex decides which set is more rewarding.

**Definition 2.33** (relevance ordering). *Let $G = (V_0, V_1, E, p)$ be a parity game. A relevance ordering $<$ is a total order that extends the partial order induced by the $p$ function. In particular $<$ is such that $\forall u, v.\, p(u) < p(v) \Rightarrow u < v$.*

It should be noted that in general multiple relevance orderings can exist for a given parity game, and usually an arbitrary one can be picked. The specific choice can affect the efficiency, but it is currently unclear how different choices impact on efficiency and if some heuristic can be devised to guide this choice.

**Definition 2.34** (positive and negative vertices). *Let $G = (V_0, V_1, E, p)$ be a parity game. We define $V_+ = \{v \in V \mid p(v) \text{ is even}\}$ and $V_- = \{v \in V \mid p(v) \text{ is odd}\}$.*

**Definition 2.35** (reward ordering). *Let $G = (V_0, V_1, E, p)$ be a parity game with a relevance ordering $<$, and let $v, u \in V$. We write $u \prec v$ when $u < v$ and $v \in V_+$ or $v < u$ and $u \in V_-$.*

$$u \prec v \Leftrightarrow (u < v \land v \in V_+) \lor (v < u \land u \in V_-)$$

**Definition 2.36** (reward ordering on sets). *Let $G = (V_0, V_1, E, p)$ be a parity game with a relevance ordering $<$ and let $P, Q \subseteq 2^V$ be two different sets of vertices. We write $P \prec Q$ if the following holds:*

$$P \neq Q \land \max_< P \,\Delta\, Q \in (P \cap V_-) \cup (Q \cap V_+)$$

Intuitively $P \prec Q$ represents the reward for $P$ being less than the one for $Q$. The way this is determined is by looking at the vertices that are in either $P$ or $Q$ but not both, namely the symmetric set difference $P \,\Delta\, Q$. The vertices that are in both are ignored because they will equally contribute to the reward of the two sets. From the symmetric difference it is then selected $v = \max_< P \,\Delta\, Q$, the greatest remaining vertex according to the relevance ordering. Then $P \prec Q$ holds when $v \in P$ and $v \in V_-$, representing the situation where $v$ is not favorable to player 0 and thus makes the reward of the left set worse, or when $v \in Q$ and $v \in V_+$, representing the situation where $v$ is favorable to player 0 and thus makes the reward of the right set better.

At the core of the algorithm there is the valuation phase computing the *play profiles*, which helps understanding how favorable a play is for each player. Moreover an ordering between play profiles is defined, with bigger values being more favorable to player 0 and lower ones being more favorable to player 1. In particular play profiles are based on three key values:

- the most relevant vertex that is visited infinitely often, which we will refer to as $w$, which directly correlates to the winner of the play;
- the vertices visited before $w$ that are more relevant than it;
- the number of vertices visited before $w$.

Recall that the game is total, thus every play is infinite, and plays induced by an instance that are infinite always consists of a prefix followed by a cycle. Thus in this case $w$ coincides with the most relevant vertex of the cycle that is reached in a play.

Intuitively in this context the last two values are linked to the chances that changing strategy would change either the value of $w$ or the cycle itself, thus more relevant vertices before $w$ or a longer prefix are more beneficial for the losing player.

**Definition 2.37** (play profile and valuation). *Let $G = (V_0, V_1, E, p)$ be a parity game with a relevance ordering $<$ and $\pi = v_0 v_1 \dots$ a play on $G$. Let $w = \max_< \inf(\pi)$ be the most relevant vertex that is visited infinitely often in the play and $\alpha = \{u \in V \mid \exists i \in N.\, v_i = u \wedge \forall j < i.\, v_j \neq w\}$ be the set of vertices visited before the first occurrence of $w$. Let $P = \alpha \cap \{v \in V \mid v > w\}$ and $e = |\alpha|$. The play profile of the play $\pi$ is the tuple $(w, P, e)$.*

*Given an instance $(G, \sigma, \tau)$ a valuation $\varphi$ is a function that associates to each vertex the play profile $(w, P, e)$ of the play induced by the instance.*

Given a valuation, we are then interested in determining whether a strategy for player 0 is optimal. It can be shown [10] that if there exist a winning strategy for a player then the *optimal* strategy is winning, otherwise it must be losing. The problem thus reduces to determining whether the current player 0 strategy is optimal, and if not improve it until it is. This can be done by looking at the play profiles of the successors of each vertex: if one of them is greater than the one of the successor chosen by the current strategy then it is not optimal. In other words the optimal strategy chooses the successor with the greatest play profile. If the strategy is not optimal then a new strategy is determined by picking for each vertex the successor with the greatest play profile. This will however change the optimal strategy for player 1 and thus the valuation, which must be recomputed, leading to another iteration. It has been shown in [10] that each new strategy “improves” upon the previous one, and eventually this process will reach the optimal strategy. This can however require $O\left(\Pi_{v \in V_0} \text{out-deg}(v)\right)$ improvement steps in the worst case. Intuitively this is because each of the $\Pi_{v \in V_0}$ out-deg$(v)$ strategies for player 0 could end up being considered.

**Definition 2.38** (play profile ordering). *Let $G = (V_0, V_1, E, p)$ be a parity game with a relevance ordering $<$, and $(u, P, e)$ and $(v, Q, f)$ be two play profiles. Then we define:*

$$(u, P, e) \prec (v, Q, f) \Leftrightarrow \begin{cases} \quad u \prec v \\ \vee \, (u = v \wedge P \prec Q) \\ \vee \, (u = v \wedge P = Q \wedge u \in V_- \wedge e < f) \\ \vee \, (u = v \wedge P = Q \wedge u \in V_+ \wedge e > f) \end{cases}$$

**Theorem 2.4** (optimal strategies). *Let $G = (V_0, V_1, E, p)$ be a parity game with a relevance ordering $<$, $\sigma$ and $\tau$ be two strategies for respectively player 0 and 1 and $\varphi$ a valuation function for $(G, \sigma, \tau)$. The strategy $\sigma$ is optimal against $\tau$ if $\forall u \in V_0.\, \forall v \in uE.\, \varphi(v) \preccurlyeq \varphi(\sigma(u))$. Dually, $\tau$ is optimal against $\sigma$ if $\forall u \in V_1.\, \forall v \in uE.\, \varphi(\tau(u)) \preccurlyeq \varphi(v)$.*

Finally, an algorithm is given in [10] to compute, given a strategy for player 0, an optimal counter-strategy for player 1 along with a valuation for them.

```
1: function valuation($H$)
2:     for $v \in V$ do
3:         $\varphi(v) = \perp$
4:     for $w \in V$ (ascending order with respect to $\prec$) do
5:         if $\varphi(w) = \perp$ then
6:             $L = reach\Big(H|_{\{v \in V \mid v \leq w\}}, w\Big)$
7:             if $E_H \cap \{w\} \times L \neq \emptyset$ then
8:                 $R = reach(H, w)$
9:                 $\varphi|_R = subvaluation(H|_R, w)$
10:                $E|_H = E|_H \setminus (R \times (V \setminus R))$
11:    return $\varphi$
12:
13: function subvaluation(K, w)
14:    for $v \in V_K$ do
15:        $\varphi_0(v) = w$
16:        $\varphi_1(v) = \emptyset$
17:    for $u \in \{v \in V_K \mid v > w\}$ (descending order with respect to $<$) do
18:        if $u \in V_+$ then
19:            $\overline{U} = reach\Big(K|_{V_K \setminus \{u\}}, w\Big)$
20:            for $v \in V_K \setminus \overline{U}$ do
21:                $\varphi_1(v) = \varphi_1(v) \cup \{u\}$
22:            $E_K = E_K \setminus \Big(\Big(\overline{U} \cup \{u\}\Big) \times \Big(V \setminus \overline{U}\Big)\Big)$
23:        else
24:            $U = reach\Big(K|_{V_K \setminus \{w\}}, u\Big)$
25:            for $v \in U$ do
26:                $\varphi_1(v) = \varphi_1(v) \cup \{u\}$
27:            $E_K = E_K \setminus ((U \setminus \{u\}) \times (V \setminus U))$
28:    if $w \in V_+$ then
```

```
29:             $\varphi_2 = maximal\_distances(K, w)$
30:         **else**
31:             $\varphi_2 = minimal\_distances(K, w)$
32:         **return** $\varphi$
```

The algorithm works by determining from which vertices player 1 can force a play to reach that vertex again, resulting in a cycle. This is done by considering the vertices with lowest reward first, as those are the ones that are more favorable to player 1. For each one that is found the algorithm then forces every vertex that can reach it to do so, by removing the edges that would allow otherwise, and hence fixing the $w$ component of their play profile. Then for this set of vertices it computes the *subvaluation*, whose goal is to find the value of the optimal player 1 strategy for them by minimizing the $P$ and $e$ components of the play profiles of these vertices. In particular this step goes through each vertex that has a higher relevance than $w$ from the one with highest relevance to the one with lowest, which are exactly those that will influence the $P$ component and its role in the play profile ordering. For each of these, if they are favorable to player 0 then it will prevent all vertices that can reach them before reaching $w$ from doing so, again by removing the edges that would allow that. If instead they are favorable to player 1 then the algorithm will force any vertex that can reach them before reaching $w$ to do so. Finally, depending on whether $w$ is favorable to player 0 or not, to each vertex is forced the longest or shortest path to reach $w$, thus fixing the $e$ component of the play profile. Ultimately this will leave each vertex with only one outgoing edge, representing the strategy for its controlling player.

It has been proven in [10] that this algorithm has a complexity of $O(|V| \times |E|)$.

### 2.7.2 Local algorithm

The strategy improvement algorithm has the downside of requiring to visit the whole graph. In some cases this might be an inconvenience, as the graph could be very large but only a small portion may need to be visited to determine the winner of a specific vertex of interest. For an extreme example, consider a disconnected graph, in which case the winner of a vertex only depends on its connected component and not on the whole graph.

The local strategy iteration algorithm [11] fills this gap by performing strategy iteration on a *subgame*, a parity game defined as a subgraph of the main game, and providing a way to determine whether this is enough to infer the winner in the full game. It may happen that the winner is not immediately decidable, in which case the subgame would have to be *expanded*. To do this we will need to define what a subgame is, how to expand it and what is the condition that decides the winner on a vertex.

**Definition 2.39** ($U$-induced subgames). *Let $G = (V_0, V_1, E, p)$ be a parity game and $U \subseteq V$. The $U$-induced subgame of $G$, written $G|_U$, is a parity game $G' = (V_0 \cap U, V_1 \cap U, E \cap (U \times U), p|_U)$, where $p|_U$ is the function $p$ with domain restricted to $U$.*

**Definition 2.40** (partially expanded game). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = G|_U$ a subgame of $G$. If $G'$ is still a total parity game it is called a partially expanded game.*

Given a partially expanded game, two optimal strategies and its winning sets, the local algorithm has to decide whether vertices winning for a player in this subgame are also winning in the full game. Recall that a strategy is winning for a player $i$ if any strategy for the opponent results in an induced play that is winning for $i$. However the fact that plays are losing in the subgame does not necessarily mean that all plays in the full game will be losing too, as they might visit vertices not included in the subgame. Intuitively, the losing player might have a way to force a losing play for them to reach one of the vertices outside the subgame, called the *$U$-exterior* of the subgame, and thus lead to a play that is not possible in the subgame. The set of vertices that can do this is called the *escape set* of the subgame, and for such vertices no conclusions can be made. For the other vertices instead the winner in the subgame is also the winner in the full game and they constitute the definitely winning sets.

**Definition 2.41** ($U$-exterior). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G|_U$ a subgame of $G$. The $U$-exterior of a vertex $v \in U$, also written $D_G(U, v)$, is the set of its that successors that are not themselves in $U$. That is, $D_G(U, v) = vE \cap (V \setminus U)$. The $U$-exterior of of the subgame $G|_U$ is instead the union of all $U$-exteriors of its vertices, that is:*

$$D_G(U) = \bigcup_{v \in U} vE \cap (V \setminus U)$$

In order to define the concept of *escape set* we will use the notion of *strategy restricted edges.* These are needed because we are interested in plays that are losing for a player, and to do that we have to restrict the moves of the opposing player to the ones represented by its optimal strategy.

**Definition 2.42** (strategy restricted edges). *Let $G = (V_0, V_1, E, p)$ be a parity game and $\sigma$ a strategy for player $i$ in $G$. The set of edges restricted to the strategy $\sigma$ is $E_\sigma = \{(u, v) \mid u \in V_i \Rightarrow \sigma(u) = v\}$.*

**Definition 2.43** (escape set). *Let $G = (V_0, V_1, E, p)$ be a parity game, $U \subseteq V$ and $G|_U$ the induced subgame of $G$. Let $L = (G|_U, \sigma, \tau)$ be an instance of the subgame.*

*Let $E^*_\sigma$ (resp. $E^*_\tau$) be the transitive-reflexive closure of $E_\sigma$ (resp. $E_\tau$). The escape set for player 0 (resp. 1) from vertex $v \in U$ is the set $E^0_L(v) = vE^*_\sigma \cap D_G(U)$ (resp. $E^1_L(v) = vE^*_\tau \cap D_G(U)$).*

**Definition 2.44** (definitive winning set). *Let $G = (V_0, V_1, E, p)$ be a parity game, $U \subseteq V$ and $G|_U$ the induced subgame of $G$. Let $L = (G|_U, \sigma, \tau)$ be an instance of the subgame with $\sigma$ and $\tau$ optimal strategies, and let $\varphi$ be the valuation for this instance. The definitive winning sets $W'_0(L)$ and $W'_1(L)$ are defined as follows:*

$$W'_0(L) = \left\{v \in U \mid E^1_L(v) = \varnothing \land (\varphi(v))_1 \in V_+\right\}$$
$$W'_1(L) = \left\{v \in U \mid E^0_L(v) = \varnothing \land (\varphi(v))_1 \in V_-\right\}$$

In practice we will however not compute the full escape sets, but instead we will find for which vertices they are empty. We can do this by considering all the vertices in $U_i$ that can reach vertices in the unexplored part of the game. Then we compute the set of vertices that can reach said vertices when the edges are restricted according to the strategy for player $1 - i$. This will result in the set of all vertices which have a non-empty escape set, so we just need to consider their complement when computing the definitive winning sets.

**Lemma 2.3** (definitive winning set soundness). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G|_U$ a subgame of $G$ with an instance $L = (G, \sigma, \tau)$. Then $W'_0(L) \subseteq W_0$ and $W'_1(L) \subseteq W_1$.*

As previously mentioned, if the winner of a vertex cannot be determined in a subgame, that is the vertex is not in a definitive winning set, then the subgame must be *expanded* to a larger subgame, which is then solved, repeating the process.

Given a partially expanded game $G|_U$, the expansion process starts by selecting new vertices in the $U$-exterior to include in the set $U$, creating a new set $U'$. However $G|_{U'}$ might not be a total parity game, so the expansion process must continue to include new vertices in $U'$ until the $U'$-induced subgame becomes total. More formally, an *expansion scheme* is made up of a *primary expansion function* $\varepsilon_1$ and a *secondary expansion function* $\varepsilon_2$, and the new subgame will be decided through a combination of them. In particular the primary expansion function will select a non-empty set of vertices in the $U$-exterior to add to the current game, while the secondary expansion function will be used to recursively select elements from the $U$-exterior of new vertices until the game becomes total.

**Definition 2.45** (expansion scheme). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G|_U$ a subgame of $G$. An expansion scheme is a pair of functions $\varepsilon_1 : 2^V \to 2^V$ and $\varepsilon_2 : 2^V \times V \to 2^V$ such that:*

- $\varnothing \subsetneq \varepsilon_1(U) \subseteq D_G(U)$
- $\forall v \in U.\, \varepsilon_2(U, v) \subseteq D_G(U, v)$
- $\forall v \in U.\, vE = D_G(U, v) \Rightarrow \varepsilon_2(U, v) \neq \varnothing$

The expansion is then computed by first applying $\varepsilon_1$ to get the set of new vertices, and then by inductively applying $\varepsilon_2$ to each new vertex until there is no new vertex produced:

$$\textsc{Expand}(U) = \textsc{Expand}_2(U, \varepsilon_1(U))$$

$$\textsc{Expand}_2(U, A) = \begin{cases} U & \text{if } A = \varnothing \\ \textsc{Expand}_2\Big(U \cup A, \bigcup_{v \in A} \varepsilon_2(U \cup A, v)\Big) & \text{otherwise} \end{cases}$$

Two expansion schemes are provided in [11], a *symmetric scheme* and an *asymmetric scheme*.

Both start by expanding the game to one of the vertices in the escape set of the vertex of interest $v^*$ for the currently losing player $i$ on it. Formally, $\varepsilon_1(U) = \{w\}$ for some $w \in E_L^i(v^*)$ where $p\Big(\big(\varphi(v^*)\big)_1\Big) \bmod 2 \equiv 1 - i$. The idea is that player $i$ has the ability to force a play from $v^*$ to reach the new vertex, which might be winning for them and thus could change the winner on $v^*$ in the new subgame. On the other hand if that does not happen then the escape set of $v^*$ for player $i$ might reduce, eventually becoming empty and thus making $v^*$ definitely winning for player $1 - i$.

The two expansion schemes differ however in the secondary expansion function. Both choose not to expand any new vertex if the just expanded vertex already has a successor in the current subgame, as doing otherwise may be wasteful. However the symmetric scheme chooses to expand only one of the successors, that is $\varepsilon_2(U, v) = \{w\}$ for some $w \in vE$. Instead the asymmetric scheme performs a different choice depending on whether $v$ is controlled by player 0 or 1. If it is controlled by player 1 it chooses to expand all the $U$-exterior of $v$, that is $\varepsilon_2(U, v) = D_G(U, v)$ if $v \in V_1$, otherwise if it is controlled by player 0 it chooses to expand only one successor like in the symmetric scheme, that is $\varepsilon_2(U, v) = \{w\}$ for some $w \in vE$ if $v \in V_0$.

Intuitively, the symmetric scheme makes no assumption about the winner and expands vertices for both players in the same way. Instead the asymmetric scheme assumes that player 0 will win, and thus tries to expand more vertices controlled by player 1 in order to reduce its escape set. Ultimately there are different tradeoffs involved, since the symmetric scheme expands relatively few vertices and thus may require solving more subgames, while the asymmetric scheme is eager, but in doing so it might expand to larger subgames that could otherwise be avoided.

Finally, the algorithm performs an initial expansion to get a total subgame that includes the vertex of interest. Then it repeatedly solves the current subgame, using an *improve* subroutine, until the vertex $v^*$ becomes definitely winning for either player, and expands it, using an *expand* subroutine, when no conclusion can be made on it.

```
1:  function local-strategy-iteration($G$, $v^*$)
2:      $U = expand(\{v^*\})$
3:      $\sigma$ = arbitrary player 0 strategy on $G|_U$
4:      $\tau$ = optimal player 1 strategy against $\sigma$ on $G|_U$
5:      $L = (G, \sigma, \tau)$
6:      while $v^* \notin W_0(L) \cup W_1(L)$ do
7:          if $\sigma$ is improvable w.r.t $L$ then
8:              $L = improve(L)$
9:          else
10:             $L = expand(L)$
11:     return $\sigma, \tau, W_0(L), W_1(L)$
```

The complexity of the algorithm depends on the specific expansion scheme used. For the two expansion schemes provided it has been proven in [11] that the asymmetric scheme will require at most $O\big(|V|^{|V_0|}\big)$ iterations, while the symmetric one will require at most $O\big(|V| \cdot |V|^{|V_0|}\big)$.

# 3 Symbolic local algorithm

## 3.1 Adapting the algorithm

Our goal will be to adapt and improve the local strategy iteration algorithm to solve systems of fixpoint equations expressed as parity games using the symbolic formulation.

### 3.1.1 Handling finite plays

The parity game formulation of a system of fixpoint equations admits positions where a player has no available moves, namely it is not a total parity game. However the strategy improvement algorithm requires a total parity game, so we need to convert a generic parity game into a "compatible" total parity game that can be handled by it, for some definition of "compatible.

The way we do this transformation is by extending the parity game, inserting auxiliary vertices that will be used as successors for those vertices that do not have one. We call this the *extended total parity game*, for short *extended game*, since it extends the original parity game to make it total. In particular we will add two vertices $w_0$ and $w_1$ representing vertices that are both controlled by and winning for respectively player 0 and 1. The vertices $w_0$ and $w_1$ will in turn also need successors, and these will be respectively $l_1$ and $l_0$, representing vertices that are controlled by and losing for respectively player 1 and 0. Likewise, the vertices $l_0$ and $l_1$ will need at least one successor, at that will be respectively $w_1$ and $w_0$. The vertices $w_0$ and $l_1$ will thus form a forced cycle, as well as $w_1$ and $l_0$. This, along with priorities chosen as favorable for the player that should win these cycles, will guarantee that the winner will actually be the expected one. Then, vertices that have no successors in the general game, meaning they are losing for the player controlling them, in the game will have as successor $w_0$ or $w_1$, that is controlled by and winning for the opposing player.

**Definition 3.1** (extended total parity game). *Let $G = (V_0, V_1, E, p)$ be a parity game. The extended total parity game of $G$ is the parity game $G' = (V_0', V_1', E', p')$ where:*

- $V_0' = V_0 \cup \{w_0, l_0\}$
- $V_1' = V_1 \cup \{w_1, l_1\}$
- $E' = E \cup \{(v, w_i) \mid i \in \{0,1\} \land v \in V_{1-i} \land v \in S_G\}$
  $\cup \{(w_0, l_1), (l_1, w_0), (w_1, l_0), (l_0, w_1)\}$

- $p'(v) = \begin{cases} p(v) & \text{if } v \in V \\ 0 & \text{if } v \in \{w_0, l_1\} \\ 1 & \text{if } v \in \{w_1, l_0\} \end{cases}$

We now want to prove that this new parity game is "compatible" with the original one, for a suitable definition of "compatible". In particular for our purposes we are interested that in the new game the winner for vertices which were already in the old game remains unchanged.

**Definition 3.2** (compatible parity games). *Let $G = (V_0, V_1, E, p)$ and $G' = (V_0', V_1', E', p')$ be two parity games with $V_i \subseteq V_i'$. Let $W_0$ and $W_1$ be the winning sets for $G$ and $W_0'$ and $W_1'$ the winning sets for $G'$. We say that $G'$ is compatible with $G$ if $W_0 \subseteq W_0'$ and $W_1 \subseteq W_1'$.*

**Definition 3.3** (extended strategies). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = (V_0', V_1', E', p')$ be the extended game from $G$. Let $\sigma$ a strategy on $G$ for player $i$. We say that $\sigma$ induces the following extended strategy $\sigma'$ on $G'$:*

$$\sigma'(v) = \begin{cases} \sigma(v) & \text{if } v \in V_i \land v \notin S_G \\ W_{1-i} & \text{if } v \in V_i \land v \in S_G \\ W_{1-i} & \text{if } v = L_i \\ L_{1-i} & \text{if } v = W_i \end{cases}$$

It can be observed that strategies on a parity game and their extended counterparts create a bijection. In fact notice that the condition $v \in V_i \land v \notin S_G$ in the first case of $\sigma'$ is equivalent to requiring $v \in \text{dom}(\sigma)$, meaning that restricting $\sigma'$ to $\text{dom}(\sigma)$ will result in $\sigma$ itself.

The bijection is not only limited to this. It can be showed that strategies that are related by this bijection will also induce plays with the same winner.

**Theorem 3.1** (plays on extended strategies). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = (V_0', V_1', E', p')$ be the extended game from $G$. Let $\sigma_0$ and $\sigma_1$ be two strategies on $G$ and $\sigma_0'$ and $\sigma_1'$ be the unique corresponding strategies on $G'$. Let $v \in V_0 \cup V_1$ and consider the plays starting from $v_0$ on the instances $I = (G, \sigma_0, \sigma_1)$ and $I' = (G', \sigma_0', \sigma_1')$. The two plays have the same winner.*

*Proof.* We will prove that for all $i$ the play induced by $I$ is won by player $i$ if and only if the induced play by $I'$ is also won by player $i$:

- $\Rightarrow$): We distinguish two cases on the play induced by $I$:
  - the play is infinite: $v_0 v_1 v_2 \ldots$, then every vertex is in $\text{dom}(\sigma_i)$ for some $i$ and thus $\sigma_i'$ are defined to be equal to $\sigma_i$ and will induce the same play, which is won by player $i$;

  - the play is finite: $v_0 v_1 ... v_n$, with $v_n \in V_{1-i}$ because the play is won by player $i$. For the same reason as the previous point the two induced plays are the same until $v_n$, which is not in $\mathrm{dom}(\sigma_{1-i})$ but is in $\mathrm{dom}(\sigma'_{1-i})$. The play induced by $I'$ is $v_0 v_1 ... v_n w_i l_{1-i} w_i ...$ which is also won by player $i$ because only the vertices $w_i$ and $l_{1-i}$ repeat infinitely often, and they have both priority favorable to player $i$.
- $\Leftarrow$): We distinguish the following cases on the play induced by $I'$:
  - the play never reaches the $w_0, w_1, l_0$ or $l_1$ vertices: $v_0 v_1 v_2 ...$, then only the first case of $\sigma'_i$ is ever used and thus every vertex is in $\mathrm{dom}(\sigma_i)$. Thus $I$ induces the same play, which is won by player $i$;
  - the play reaches $w_i$: $v_0 v_1 ... v_n w_i l_{1-i} w_i ...$, then $v_n$ is not in $\mathrm{dom}(\sigma_{1-i})$ and $I$ induces the finite play $v_0 v_1 ... v_n$ which is won by player $i$ because $v_n \in V_{1-i}$ due to its successor being controlled by player $i$;
  - the play reaches $w_{1-i}$: this is impossible because it would be winning for player $1 - i$, which contradicts the hypothesis;
  - the play reaches $l_i$ or $l_{1-i}$ before $w_i$ or $w_{1-i}$: this is impossible because the only edges leading to $l_i$ or $l_{1-i}$ start from $w_{1-i}$ and $w_i$. □

**Theorem 3.2** (compatibility of extended games). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = (V'_0, V'_1, E', p')$ be the extended game from $G$. Then $G'$ is compatible with $G$, that is $\forall i.\, W_i \subseteq W'_i$.*

*Proof.* Let $v \in W_i$, then there exist a winning strategy $\sigma_i$ for player $i$. We claim that the extended strategy $\sigma'_i$ for player $i$ on $G'$ is also winning. In fact consider any strategy $\sigma'_{1-i}$ for player $1 - i$ on $G'$, then it is the extended strategy of some strategy $\sigma_{1-i}$ on $G$. We know that the play starting from $v$ on the instance $(G', \sigma'_0, \sigma'_1)$ is won by the same player as the play starting from $v$ on the instance $(G, \sigma_0, \sigma_1)$. Moreover since $\sigma_i$ is a winning strategy for player $i$ we know that these plays are won by player $i$, thus $v \in W'_i$ and so $W_i \subseteq W'_i$. □

### 3.1.2 Generalizing subgames with subset of edges

The local strategy improvement algorithm gives a way to consider only a subset of the vertices, but still assumes all edges between such vertices to be known. However this is not necessarily true in the symbolic formulation, as the list of successors of vertices in $V_0$ is computed lazily, and this might include vertices already in the subgame. We thus have to update the local algorithm to handle this case by extending the idea of escape set. Instead of identifying those vertices that can reach the $U$-exterior we will identify those vertices that can reach an "unexplored" edge, that is an edge present in the full game but not in the subgame. We will call the vertices directly connected to such edges *incomplete vertices*. Note that the resulting set will be a superset of the $U$-exterior, since edges that lead outside $U$ cannot be part of the subgame.

**Definition 3.4** (subgame). *Let $G = (V_0, V_1, E, p)$ be a parity game, $U \subseteq V$ and $E' \subseteq E \cap (U \times U)$, then $G' = (V_0 \cap U, V_1 \cap U, E', p|_U)$ is a subgame of $G$, where $p|_U$ is the function $p$ with domain restricted to $U$. We will write $G' = (G, U, E')$ for brevity.*

**Definition 3.5** (escape set (updated)). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = (G, U, E')$ a subgame of $G$. Let $L = (G|_U, \sigma, \tau)$ be an instance of the subgame. Let $E^*_\sigma$ (resp. $E^*_\tau$) be the transitive-reflexive closure of $E_\sigma$ (resp. $E_\tau$) and $I_G = \{v \mid vE \neq vE'\}$ the set of vertices that have unexplored outgoing edges. The (updated) escape set for player 0 (resp. 1) from vertex $v \in U$ is the set $E^0_L(v) = vE^*_\sigma \cap I_G$ (resp. $E^1_L(v) = vE^*_\tau \cap I_G$).*

**Theorem 3.3** (definitive winning set is sound). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = (G, U, E')$ a subgame of $G$. Let $G = (V_0, V_1, E, p)$ be a parity game and $U \subseteq V$. Let $L = (G|_U, \sigma, \tau)$ be an instance of the subgame where $\sigma$ and $\tau$ are optimal strategies. Then $W'_0 \subseteq W_0$ and $W'_1 \subseteq W_1$.*

*Proof.* Let $v \in W'_i$, then there exist a strategy $\sigma_i$ on $G'$ the for player $i$ such that for any strategy $\sigma_{1-i}$ for player $1-i$ on $G'$ the resulting play is winning for player $i$. Moreover $E^{1-i}_L(v) = \emptyset$ by definition of $W'_i$, meaning that in the graph restricted to the strategy $\sigma_i$, any vertex controlled by player $1-i$ that has unexplored edges is not reachable. This in turn means that on the full game $G$ the strategy $\sigma_i$ is still winning, because for any strategy $\sigma'_{1-i}$ for player $1-i$ on $G$ the resulting play will still be within the subgame, since no unexplored edge can be reached, and any such play is winning for player $i$, hence $v \in W_i$. □

### 3.1.3 Expansion scheme

In the local strategy iteration the expansion scheme is based on the idea of expanding the subgame by adding new vertices. In our adaptation it will instead add new edges, and vertices will be implicitly added if they are the endpoint of a new edge. This does not however change much of the logic behind it, since the expansion schemes defined in [11] are all based on picking some unexplored successor, which is equivalent to picking the unexplored edge that leads to it.

More formally, the $\varepsilon_1$ and $\varepsilon_2$ functions now take the set of edges in the subgame and output a set of new edges to add to the subgame. The requirements remain similar, in that $\varepsilon_1$ must return a non-empty set of edges that are not already in the subgame and $\varepsilon_2$ must return a set of outgoing edges from the given vertex. Moreover if a vertex has no successor then $\varepsilon_2$ must also be non-empty in order to given that vertex a successor and make the game total.

**Definition 3.6** (expansion scheme (updated)). *Let $G = (V_0, V_1, E, p)$ be a parity game and $G' = (G, U, E')$ a subgame of $G$. An expansion scheme is a pair of functions $\varepsilon_1 : 2^E \to 2^E$ and $\varepsilon_2 : 2^E \times V \to 2^E$ such that:*

- $\varnothing \subsetneq \varepsilon_1(E') \subseteq E \setminus E'$
- $\varepsilon_2(E', v) \subseteq (\{v\} \times vE) \setminus E'$
- $vE = D_G(U, v) \Rightarrow \varepsilon_2(E', v) \neq \varnothing$

As before the expansion is computed by first applying $\varepsilon_1$ and then by repeatedly applying $\varepsilon_2$.

$$\textsc{Expand}(E') = \textsc{Expand}_2(E', \varepsilon_1(E'))$$

$$\textsc{Expand}_2(E', E'') = \begin{cases} E & \text{if } E'' = \varnothing \\ \textsc{Expand}_2\Big(E' \cup E'', \bigcup_{(u,v)\in E''} \varepsilon_2(E' \cup E'', v)\Big) & \text{otherwise} \end{cases}$$

For our implementation we decided to adapt the symmetric expansion scheme from [11]. The adapted $\varepsilon_1$ picks any edge from a vertex in the escape set of $v^*$ for the losing player $i$, that is, $\varepsilon_1(E') = \{e\}$ for $e \in (v \times vE') \setminus E'$, some $v \in E_L^i(v^*)$ and $p\Big(\big(\varphi(v^*)\big)_1\Big) \bmod 2 \equiv 1 - i$, while the adapted $\varepsilon_2$ picks any unexplored edge from the given vertex $v$ if it has no successors, that is $\varepsilon_2(E', v) = \{e\}$ for some $e \in (v \times vE') \setminus E'$ if $vE' = \varnothing$, otherwise $\varepsilon_2(E', v) = \varnothing$. The choices it makes are almost the same as those of the original symmetric algorithm if each chosen edge is replaced with its head vertex, with the exception that it may select edges that lead to already explored vertices.

It should be noted that the upper bound on the number of expansions grows from $O(|V|)$, caused by when each expansions adds only a single vertex to the subgame, to $O(|E|)$, now caused by when each expansion adds only one edge to the subgame. As shown in [11], a big number of expansions might not be ideal because each will require at least one strategy iteration, which in the long run can end up being slower than directly running the global algorithm.

On the other hand a lazier expansion scheme can take better advantage of the ability to perform simplifications on symbolic moves, which allows to remove lot of edges with little work. A eager expansion scheme may instead visit all those edges, just to ultimately find out that they were all losing for the same reason. There is thus a tradeoff between expanding too much in a single step, which loses some of the benefits of using symbolic moves, and expanding too little, which instead leads to too many strategy iterations.

### 3.1.4 Symbolic formulas iterators and simplification

Differently from the implementation in [8], we need to generate symbolic moves lazily in order to take advantage of the local algorithm and the simplification of formulas. To do this we represent the generator for symbolic moves described in Section 2.6.3 as a sequence of moves rather than as a set. Then, we can generate moves in the same order they appear in the sequence, and keep track of which point we have reached.

For sake of simplicity we assume that every $\wedge$ and $\vee$ operator with a single subformula can be first simplified to that subformula itself, while $\wedge$ and $\vee$ operators with more than two subformulas can be rewritten to nested $\wedge$ and $\vee$ operators each with exactly two subformulas using the associative property. We thus define the sequence of moves for each type of formula as follows, where for the recursive case we take $M(\varphi_i) = (\boldsymbol{X}_{i1}, \boldsymbol{X}_{i2}, ..., \boldsymbol{X}_{in})$:

$$\begin{aligned}
M([b,i]) &= (\boldsymbol{X}) \text{ with } X_i = \{b\} \text{ and } \forall j \neq i.\, X_j = \varnothing \\
M(true) &= (\boldsymbol{X}) \text{ with } \forall i.\, X_i = \varnothing \\
M(false) &= () \\
M(\varphi_1 \vee \varphi_2) &= (\boldsymbol{X}_{11}, \boldsymbol{X}_{12}, ..., \boldsymbol{X}_{1n} \boldsymbol{X}_{21}, \boldsymbol{X}_{22}, ..., \boldsymbol{X}_{2m}) \\
M(\varphi_1 \wedge \varphi_2) &= (\boldsymbol{X}_{11} \cup \boldsymbol{X}_{21}, ..., \boldsymbol{X}_{11} \cup \boldsymbol{X}_{2m}, \boldsymbol{X}_{12} \cup \boldsymbol{X}_{21}, ..., \boldsymbol{X}_{1n} \cup \boldsymbol{X}_{2m})
\end{aligned}$$

Intuitively a formula $[b,i]$ represents a sequence consisting of a single element, $true$ also represents a sequence of a single winning move for player 0, while $false$ represents an empty sequence which is thus losing for player 0. The $\vee$ operator represent concatenating the two (or more) sequences, with the left one first, and the $\wedge$ operator is equivalent to the cartesian product of the two (or more) sequences, by fixing an element of the first sequence and joining it with each element of the second sequence, then repeating this for all elements of the first sequence.

In practice the implementation is based on *formula iterators*, on which we define three operations:

- getting the current move;
- advancing the iterator to the next move, optionally signaling the end of the moves sequence;
- resetting the iterator, thus making it start again from the first move.

These are implemented for every type of formula:

- for $[b,i]$ formula iterators:
  - the current move is always $\boldsymbol{X}$ with $X_i = \{b\}$ and $\forall j \neq i.\, X_j = \varnothing$;
  - advancing the iterator always signals that the sequence has ended, since there is ever only one move;
  - resetting the iterator always does nothing, since the first move is always the end represented by the iterator.
- for $\varphi_1 \vee \varphi_2$ formula iterators:
  - the current move is the current move of the currently active subformula iterator, which is kept as part of the iterator state;

- ‣ advancing the iterator means advancing the iterator of the currently active subformula, and if that signals the end of the formula then the next subformula becomes the active one. If there is no next subformula then the end of the sequence is signaled;
  - ‣ resetting the iterator means resetting the iterators for both subformulas and making $\varphi_1$ the currently active subformula.
- for $\varphi_1 \land \varphi_2$ formula iterators:
  - ‣ the current move is always the union of the current move of the two subformula iterators;
  - ‣ advancing the iterator means advancing the iterator of the right subformula, and if that reports the end of the sequence then it is resetted and the iterator for the left subformula is advanced. If that also reports the end of its sequence then this iterator also reports the end of its sequence;
  - ‣ resetting the iterator means resetting the iterators of both subformulas.

**Example 3.1** (formula iterator). Consider for example the formula $(a \lor b) \land (c \lor d)$, where for sake of simplicity we have represented atoms by a single variable letter. The sequence of its moves would then be $\{a, c\}$, $\{a, d\}$, $\{b, c\}$ and $\{b, d\}$. Initially the formula iterator would start with the following state, where red edges represent the currently active subformula of an $\lor$ formula:

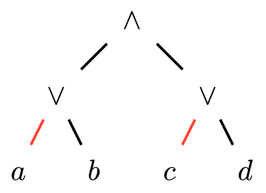

Figure 7: Example of formula iterator

The current move would then be $\{a, c\}$, since the $\land$ formula performs the union of the moves of its two subformulas, while the two $\lor$ subformulas select their left subformula as active.

Advancing the iterator would result in advancing the iterator for the right subformula of the $\land$, which happens without reaching its end, thus resulting in the following formula iterator:

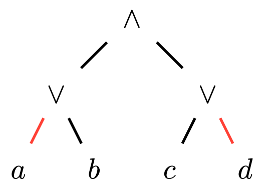

Figure 8: Example of formula iterator after one step

The current formula would then be $\{a, d\}$, which is also the next move in the original sequence.

Advancing again the iterator would result in the right subformula signaling it has reached its end, and thus the $\wedge$ subformula advances the left subformula and resets the right one, resulting in the following iterator:

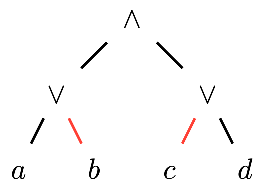

Figure 9: Example of formula iterator after two steps

This time the current move is $\{b, c\}$, the next one in the sequence.

Advancing would again advance the right subformula:

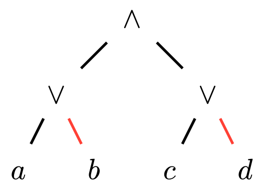

Figure 10: Example of formula iterator after two steps

The current move is now $\{b, d\}$, the last move in the sequence. In fact advancing again would result in the right subformula signaling it has reached its end, causing the left subformula to also advance and reach its end, ultimately resulting in the whole formula iterator reaching its end.

As mentioned briefly in Section 2.6.3, in LCSFE [8] formulas are simplified once before exploring their moves according to the assumptions on the winner for each vertex made at that point in the exploration. This is however not applicable to our case since we lazily explore moves, and thus have to simplify formulas whose moves have already been partially explored. An option would be performing simplifications anyway, losing the information about which moves have already been explored and thus needing to explore them again. We however want to preserve this information to avoid exploring moves over and over, and thus need a way to simplify formulas while tracking the effects on their iterator.

The way we do this is by considering how the operation of simplifying a formula iterator can be seen on their sequence. It turns out that simplifying a formula is equivalent to removing some elements from its sequence, in particular simplifying a formula to *false* removes all the moves from its sequence, while simplifying a formula to *true* removes all the moves from its sequence except the first winning one. Simplifying a formula iterator then requires simplifying its subformula iterators, which in turn might remove moves from the parent formula iterator. Most importantly, the current move may also be among those removed moves, in which case the iterator needs to be advanced to the next remaining move, potentially reaching its end. Note that a formula iterator might also need to be adjusted depending on whether a subformula has been advanced or reached its end after being simplified; for example if the left subformula of an $\wedge$ formula is advanced, even if once, then from the point of view of the sequence of moves of the $\wedge$ formula a lot of moves might have been skipped, corresponding to all the pairs between the skipped move on the left subformula and all the moves in the right subformula.

**Example 3.2** (formula iterator simplification). Consider again an iterator for the formula $(a \vee b) \wedge (c \vee d)$ in the following state:

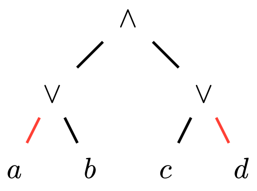

Figure 11: Example of formula iterator simplification

If it becomes known that the position represented by the atom $c$ is winning, then we might want to simplify the $c \vee d$ branch to just $c$, since $c$ will always be a best move for player 0. This is similar to assigning to *true* to $c$, resulting in $c \vee d$ also being *true*, though, from the point of view of the sequence of moves, keeping $c$ is more intuitive since we ultimately want a winning move. Note however that we also want to update its current move, and since $c$ was already considered due to appearing on the left side of the $\vee$ formula, the new iterator is thus considered as having reached its end.

From the point of view of the sequence of moves for the $\wedge$ formula however, this is equivalent to discarding all the moves derived from the $d$ in the right subformula and instead considering only those derived from $c$, thus the original sequence with $\{a, c\}$, $\{a, d\}$, $\{b, c\}$ and $\{b, d\}$ would become just $\{a, c\}$ and $\{b, c\}$. Notice however how the iterator has already considered the move $\{a, c\}$, and thus it should advance to the next move $\{b, c\}$. This can be inferred by the fact that the right subformula has reached its end, so just like when advancing the $\wedge$ formula, the left subformula is advanced to $b$ and the right subformula is resetted, which for a formula iterator consisting of just $c$ does nothing. Thus we end up with the following simplified formula iterator:

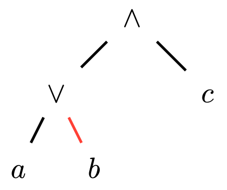

Figure 12: Example of formula iterator simplification

Notice how this iterator would consider exactly the moves $\{a, c\}$ and $\{b, c\}$ if restarted, but instead is currently considering the move $\{b, c\}$ because the original iterator already considered the move $\{a, c\}$ and would be a waste to consider it again.

When simplifying we will be interested, for every subformula, about whether it has been simplified to $true$, $false$ or whether its truth value is still unknown. In case it has not been simplified to $false$ we will also care about whether it has reached the end of its sequence after the simplification, and if not whether the current move has changed or not. This will be useful to update the current move of the parent formula iterators. In particular:

- for $[b, i]$ formulas, simplifying them depends on whether it is known that the position for player 0 corresponding to that atom is definitely winning or not:
  - if it is definitely winning the iterator remains unchanged, since the only move in the sequence it represents is winning, while the information that it is winning is propagated to the parent formula iterator;
  - if it is definitely losing the iterator is replaced with $false$, effectively removing all moves from the sequence;
  - if it is neither of them then the iterator is not changed.
- $true$ and $false$ formulas do not need to be simplified, since they are already as much simplified as possible;
- for $\lor$ formulas, each subformula is simplified, thus any move that is removed from those subformulas sequences is also removed from the $\lor$ sequence. Then:
  - if one of the subformulas is simplified to $true$ then this formula simplifies to $true$. The current move is updated based on whether the winning move was before the current move, in which case the iterator reaches its end, the current move itself, in which case it remains the same, or after the current move, in which case the current move it updated to the winning move.
  - if all the subformulas are simplified to $false$ then this formula is also simplified to $false$ and reaches its end;
  - otherwise the current move is updated to the new current move of the current subformula if it has not reached its end, to the first move of the next subformula if that exists, which becomes the new active subformula, or the iterator signals having reached the end of the sequence.

- for $\wedge$ formulas each subformula is simplified and moves that use removed moves from any subformulas are removed. Then:
  - if any subformula has been simplified to $false$ then the whole formula also simplified to $false$ and reaches its end;
  - if all subformulas have been simplified to $true$ then this formula also simplifies to $true$. If the current move is the winning one nothing changes, otherwise if the first subformula whose winning move is not the current one had already considered that move the iterator reaches its end, if not the current move is advanced until the winning one;
  - otherwise the first subformula from the left that has reached its end causes the advance of the subformula on its left and the reset of itself and all the ones on its right. If there is no subformula on its left the whole iterator has reached its end.

## 3.2 Improvements

### 3.2.1 Graph simplification

In the local strategy iteration it may happen that we learn about the winner on a vertex that is not the one we are interested in. When this happens we will do a lot of wasted work in the subsequent valuations steps, since it will have to visit its edges again and again. We now propose a transformation that produces a compatible graph and reduces the amount of edges of vertices in the definitely winning sets, thus decreasing the amount of work that the valuation step needs to perform. Informally, the idea will be to replace all outgoing edges from vertices in a definitely winning set with one pointing to one of the four auxiliary vertices $w_0$, $l_0$, $w_1$ or $l_1$ in such a way that its winner is preserved and the graph remains bipartite.

**Definition 3.7** (simplified graph). *Let $G = (V_0', V_1', E', p)$ be the extended game of some game $(V_0, V_1, E, p)$, let $G' = (G, U, E'')$ be a partially expanded game with $\{w_0, l_0, w_1, l_1\} \in U$ and let $W_0'$ and $W_1'$ be the definitely winning sets of $G'$. Let $v \in (V_0 \cup V_1) \cap (W_0' \cup W_1')$, then $G$ can be simplified to the graph $G'' = (V_0', V_1', E''', p)$ where:*

- *if $v \in V_0 \cap W_0'$ then $E''' = E' \setminus vE'' \cup \{(v, l_1)\}$;*
- *if $v \in V_0 \cap W_1'$ then $E''' = E' \setminus vE'' \cup \{(v, w_1)\}$;*
- *if $v \in V_1 \cap W_1'$ then $E''' = E' \setminus vE'' \cup \{(v, l_0)\}$;*
- *if $v \in V_1 \cap W_0'$ then $E''' = E' \setminus vE'' \cup \{(v, w_0)\}$;*

**Theorem 3.4** (simplified graph compatible). *Let $G = (V_0, V_1, E, p)$ be an extended parity game which has been simplified to $G'' = (V_0', V_1', E', p)$ according to the previous definition. Then $G''$ is compatible with $G$.*

*Proof.* We want to prove that the winning sets in $G$ are equal to the ones in $G''$, that is $\forall i.\, W_i = W_i''$. Without loss of generality we assume the simplification has happened on a vertex $v \in W_0'$. Consider now any vertex $u \in W_i$, that is winning for some player $i$ in $G$. We want to prove that $u \in W_i''$ too. Consider any winning strategy for player $i$ and any other strategy for player $1 - i$ in $G$. Any play in $G$ induced by these two strategies will be winning for player $i$ since we have $v \in W_i$. We now distinguish two cases:

- $i = 0$, then these plays could reach vertex $v$. The corresponding play in $G''$ would then also reach $v$, but would then only be able to reach $l_1$, $w_0$ and loop between them. The resulting play would is however also won by player 0, hence $v \in W_0''$.
- $i = 1$, then it is not possible for the play in $G$ to reach vertex, since otherwise player 0 would have a strategy to continue the play and win it, resulting in $u \in W_0$ instead of $W_1$. Hence all plays in $G$ do not go through $v$ and remain the same in $G'$, thus remaining winning for player 1 and $u \in W_1''$.

$\square$

### 3.2.2 Computing play profiles of the expansion

Each game expansion is normally followed by a strategy iteration step, which computes the play profile of each vertex and then tries to improve the current strategy. We can notice however that the play profiles of all the vertices are known right before the expansion, and if we keep the current strategies fixed, both for player 0 and 1, then the newer vertices cannot influence the play profiles for the existing vertices, since the existing strategies will force any play to remain within the edges in the old subgame. Hence, we can compute the play profiles for the newer vertices in isolation, and only then determine if the existing strategies can be improved given the newer vertices.

It is known that a play profile on a vertex depends on the vertex itself and on the play profile of its successor according to the strategy for the player controlling that vertex. In particular, given a vertex $x$ and its successor $y$ we know the following about its play profile components $\varphi_0$, $\varphi_1$ and $\varphi_2$:

$$\varphi_0(x) = \varphi_0(y)$$

$$\varphi_1(x) = \begin{cases} \varphi_1(y) & \text{if } x < \varphi_0(x) \\ \varnothing & \text{if } x = \varphi_0(x) \\ \varphi_1(y) \cup \{x\} & \text{if } x > \varphi_0(x) \end{cases}$$

$$\varphi_2(x) = \begin{cases} \varphi_2(y) + 1 & \text{if } x \neq \varphi_0(x) \\ 0 & \text{if } x = \varphi_0(x) \end{cases}$$

Notice however how this can result in a cyclic dependency if we need to compute the play profiles of multiple vertices creating a cycle. We thus distinguish two cases:

- if the expansion stops by reaching an existing vertex then its play profile was already known and there is no cyclic dependency. Each play profile can be computed based on the one of the successor, starting with the play profile of the last new vertex found;
- if the expansion stops by reaching a vertex found in the current expansion then there is a cyclic dependency. The cyclic dependency can however be broken by finding the most relevant vertex of the cycle $w$, for which we know that $\varphi(w) = (w, \varnothing, 0)$. This then breaks the cyclic dependency, since we know the play profile of one of the vertices in the cycle, and we can compute the play profiles of the rest like in the previous case.

By computing the play profiles after an expansion step we can thus perform an improvement right away without having to go through a valuation step to recompute the play profiles. We can further improve this by noticing that the play profiles of existing vertices did not change, thus allowing us to skip the improvement check for any vertex that did not have an outgoing edge just added.

Ultimately this allows us to skip a lot of valuation steps, which are relatively expensive. This also allows to reduce some of the downsides of the local algorithm, among which there is an increased amount of valuation steps required.

### 3.2.3 Exponentially increasing expansions

While lazier expansion schemes are intuitively better when paired with symbolic moves simplification, and the incremental play profiles computation helps often removes the need to perform an expensive valuation step, it can still happen that games fall into the worst case of expanding only a handful of edges in each iteration without being able to perform significant simplifications. This can be avoided by expanding more eagerly, like in the asymmetric expansion scheme for the local strategy improvement algorithm, but ideally we would like to be lazier when possible.

We thus changed the expansion logic to repeatedly expand until a minimum amount of edges has been added to the game. We choose this number to be initially pretty small in order to favour the locality of the algorithm, but made it increase to favour more eager expansions once it becomes clear that the winner cannot be quickly determined locally.

There are multiple ways to perform this increase, and this will influence the final complexity of the algorithm. In our case we choose to increase this number exponentially, thus guaranteeing that the maximum number of expansions is logarithmic in the amount of edges and keeping the cost of the worst cases under control.

To see why this is the case consider the sum of the number of edges $e_i$ added in each expansion $i$. We require each $e_i$ to be at least $a \times b^i$ for some constants $a > 0$

and $b > 1$. This creates a geometric progression, whose sum is known to be $a\frac{b^n - 1}{b - 1}$, though for our purposes we can focus only on bounding it by $ab^n$.

$$\begin{aligned}
\#\text{edges added} &= e_0 + e_1 + e_2 + ... + e_n \\
&\geq a + ab + ab^2 + ... + ab^n \\
&\geq ab^n
\end{aligned}$$

Then we know that in the worst case we can add at most $|E|$, since those are all the edges. This gives the equation $|E| \geq ab^n$, which if we solve for $n$ given $n \leq \log_b \frac{|E|}{a}$.

# 4 Implementation

The final goal of this thesis was a concrete implementation of the algorithms explained in the previous sections. The implementation partly relies on the work done in LCSFE [8] which, as mentioned in the introduction, was based on a different algorithm for parity games. The final implementation is available in the repository https://github.com/SkiFire13/master-thesis-code

In this section we will explain our design choices, what was actually implemented, and we will present a performance comparison with some existing tools.

## 4.1 Technologies used

Just like LCSFE, our implementation is written in Rust [16], a modern systems programming language, focused on performance and correctness and whose goal is to rival languages like C and C++ while offering memory safety. Just like C and C++, Rust mainly follows the imperative paradigm, allowing mutations, loops and general side effects, but it also includes lot of functional programming related features, like algebraic data structures and most notably *enums*, pattern matching, which allows to exhaustively inspect those enums, and *closures*, which are anonymous function that can capture their outer environment, although with some limitations due to how the memory management works. Among other features there are *traits*, which work similarly to type classes in Haskell and fill the same use cases as interfaces in popular OOP languages like Java. It should also be mentioned that Rust programs are organized in *crates*, which make up the unit of compilation, and *modules*, which are a hierarchical division internal to a crate and help organize code and avoid name clashes.

The most interesting features however are its *ownership* system and its borrow checker, which allow the compiler to guarantee memory safety without a garbage collection or other kind of runtime support. The ownership system enforces that every value has exactly one *owner*, which is responsible for freeing up its resources, making classes of issues like use-after-free impossible, and others like memory leaking much more difficult to hit. The borrow checker instead rules how borrows can be created and used. Every variable can be borrowed, creating either a shared reference or an exclusive references, which are pointers with a special meaning for the compiler. The borrow checker ensures that at any point in time there can be either multiple shared references or one exclusive reference pointing to a variable, but not both. Coupled with the fact that only exclusive references allow mutations, this system guarantees that references always point to valid data.

The borrowing rules however can become an obstacle when writing programs that perform lot of mutations, especially for programmers used to other imperative languages. It has been found however that data oriented designs in practice work pretty well with the borrow checker, due to the ability to replace pointers with indexes and thus restricting the places where borrows need to be created. This paradigm also helps creating cache efficient programs, which can often be faster. For this reason we tried to implement out algorithm with a data oriented design, which was mainly done by associating an auto-incrementing index to each vertex. Then informations associated with vertices, like their successors or remaining moves, was each stored in its own array indexed by the same index on vertices.

## 4.2 Structure of the implementation

The implementation was split in multiple crates, just like in the original LCSFE implementation. It consists of one main *solver* crate implementing the solving algorithm and multiple dependent crates, that translate specific problems into systems of fixpoint equations with logic formulas ready to be solved by the solver crate and offer a CLI interface for testing such functionalities.

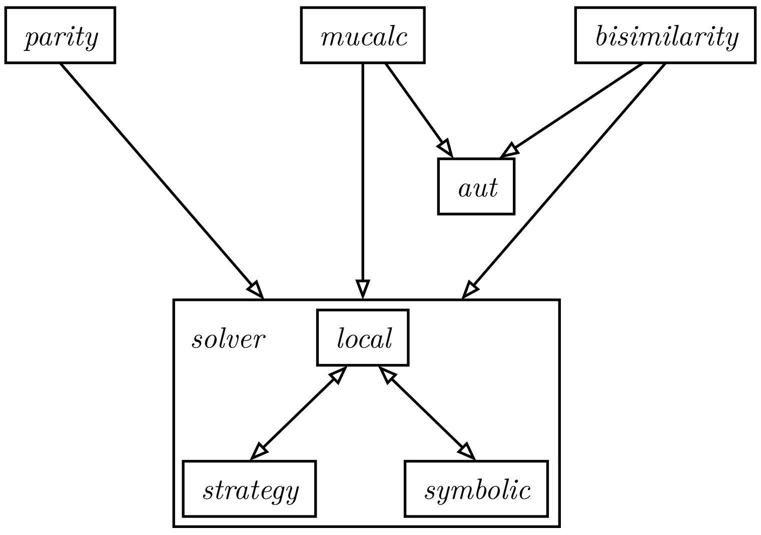

Figure 13: Crates tree of the implementation

The crates involved are the following:

- *parity*, which implements the parsing and translation from parity games to a system of fixpoint equations, which we saw in section Section 4.3, and a binary crate for the associated CLI;
- *aut*, which implements the parsing of labelled transition system files from the AUT format (also called Aldebaran) and is consumed by both the *mucalc* and *bisimilarity* crates;

- *mucalc*, which implements the parsing of a subset of $\mu$-calculus formulas, followed by their translation to a system of fixpoint equations and logic formulas as shown in Sections 2.4.1 and 2.6.4, and along with a binary crate for the associated CLI;
- *bisimilarity*, which implements the translation from a bisimilarity problem between two states of two different labelled transition systems to a system of one fixpoint equation and then logic formulas as shown in Sections 2.4.2 and 2.6.5, along with a binary crate for the associated CLI.

The *solver* crate is also internally split into three main modules implementing the major pieces of functionality:

- *symbolic*, which defines the structures for systems of fixpoint equation and logic formulas, and more importantly implements formula iterators their simplification;
- *strategy*, which implements the strategy iteration algorithm;
- *local*, which implements the local algorithm and the expansion scheme, along with the improvement we made to them, connecting to the *symbolic* module to generate new moves when necessary and to the *strategy* module to perform the valuation and improvement steps.

## 4.3 Testing with parity games

As mentioned in Section 2.6.6 parity games can be translated to systems of fixpoint equations, and we used this fact to generate simple problems for testing our implementation.

The *parity* crate implements this conversion from parity games to systems of fixpoint equations and then logic formulas, along with a parser for parity games specified in the pgsolver [17] format, according to the following grammar:

$$\begin{aligned}
\langle \mathit{parity\_game} \rangle &::= [\texttt{parity}\ \langle \mathit{identifier} \rangle\ \ ;]\ \ \langle \mathit{node\_spec} \rangle^{+} \\
\langle \mathit{node\_spec} \rangle &::= \langle \mathit{identifier} \rangle\ \ \langle \mathit{priority} \rangle\ \ \langle \mathit{owner} \rangle\ \ \langle \mathit{successors} \rangle\ \ [\langle \mathit{name} \rangle]\ \ ; \\
\langle \mathit{identifier} \rangle &::= \mathbb{N} \\
\langle \mathit{priority} \rangle &::= \mathbb{N} \\
\langle \mathit{owner} \rangle &::= \texttt{0} \mid \texttt{1} \\
\langle \mathit{successors} \rangle &::= \langle \mathit{identifier} \rangle\ \ (,\ \langle \mathit{identifier} \rangle)^{*} \\
\langle \mathit{name} \rangle &::= \texttt{"}\ \ \text{(any ASCII string not containing '"')}\ \ \texttt{"}
\end{aligned}$$

For example the parity game shown in Figure 4 would be specified in the following way:

```
parity 5
0 0 0 1,2;
```

```
1 2 1 0;
2 3 1 1,3;
3 5 0 4;
4 4 0 2,3;
```

The format consists of a header containing the identifier `parity` followed by a number indicating how many vertices will be specified, which can be used to speed up the parsing of the file. Then each of the following lines specifies a vertex with, in order, its identifier, priority, controlling player, edges and optionally a name. For the sake of simplicity we assumed the names to never be present, since they are not required for solving the game and were not present in the games we exploited for our testing activity.

We used the parity game instances included in the Oink [18] collection of parity game solvers to test our implementation. These tests are pretty small, reaching a maximum of 24 vertices and 90 edges, but they include lot of tricky cases which help getting empiric evidence of the correctness of our implementation.

## 4.4 Testing with $\mu$-calculus

As mentioned in Sections 2.4.1 and 2.6.4, $\mu$-calculus formulas can be translated to systems of fixpoint equations and then to logic formulas. We implemented this in the *mucalc* crate, which performs this translation after parsing a labeled transition system and a $\mu$-calculus formula from two given files.

The labelled transition system is expected to be in the AUT (Aldebaran) format, according to the following grammar, which based on the one given in [19]:

$$
\begin{aligned}
\langle \textit{aut} \rangle &::= \langle \textit{header} \rangle \ \langle \textit{transition} \rangle^* \\
\langle \textit{header} \rangle &::= \texttt{des} \ ( \ \langle \textit{initial-state} \rangle \ , \ \langle \textit{transitions-count} \rangle \ , \ \langle \textit{states-count} \rangle ) \\
\langle \textit{initial-state} \rangle &::= \mathbb{N} \\
\langle \textit{transitions-count} \rangle &::= \mathbb{N} \\
\langle \textit{states-count} \rangle &::= \mathbb{N} \\
\langle \textit{transition} \rangle &::= ( \ \langle \textit{from-state} \rangle \ , \ \langle \textit{label} \rangle \ , \ \langle \textit{to-state} \rangle \ ) \\
\langle \textit{from-state} \rangle &::= \mathbb{N} \\
\langle \textit{label} \rangle &::= \langle \textit{unquoted-label} \rangle \mid \langle \textit{quoted-label} \rangle \\
\langle \textit{unquoted-label} \rangle &::= (\text{any character except } \texttt{"}\ ) \ (\text{any character except } ,)^* \\
\langle \textit{quoted-label} \rangle &::= \texttt{"} \ (\text{any character except } \texttt{"}\ )^* \ \texttt{"} \\
\langle \textit{to-state} \rangle &::= \mathbb{N}
\end{aligned}
$$

The grammar consists of a header containing the literal “des” followed by the initial state number, the number of transitions and the number of states. After that, are all the transitions, encoded as a triple $(s, \mathit{label}, t)$, where the first and last components

$s$ and $t$ are the source and target state of the transition, while the second component is the label, which can be quoted or not. For the sake of simplicity we have diverged from the specification at [19] by considering labels as either a sequence of characters until the first comma or as sequence of characters delimited by quotes. In particular we have ignored character escaping and any restrictions on which characters are allowed to be used.

The given grammar for a $\mu$-calculus formula mostly follows the definition previously given in Section 2.4.1:

$$
\begin{aligned}
\langle \mathit{expr} \rangle &::= \langle \mathit{fix\text{-}expr} \rangle \mid \langle \mathit{or\text{-}expr} \rangle \\
\langle \mathit{fix\text{-}expr} \rangle &::= (\mathsf{mu} \mid \mathsf{nu})\ \langle \mathit{var} \rangle\ .\ \langle \mathit{or\text{-}expr} \rangle \\
\langle \mathit{var} \rangle &::= (\text{any identifier}) \\
\langle \mathit{or\text{-}expr} \rangle &::= \langle \mathit{and\text{-}expr} \rangle\ (\ ||\ \langle \mathit{and\text{-}expr} \rangle\ )^* \\
\langle \mathit{and\text{-}expr} \rangle &::= \langle \mathit{modal\text{-}expr} \rangle\ (\ \&\&\ \langle \mathit{modal\text{-}expr} \rangle\ )^* \\
\langle \mathit{modal\text{-}expr} \rangle &::= (<\ \langle \mathit{action} \rangle\ >\ \langle \mathit{atom} \rangle) \mid (\ [\ \langle \mathit{action} \rangle\ ]\ \langle \mathit{atom} \rangle) \mid \mathit{atom} \\
\langle \mathit{action} \rangle &::= \mathsf{true} \mid \langle \mathit{label} \rangle \mid !\ \langle \mathit{label} \rangle \\
\langle \mathit{label} \rangle &::= (\text{any character except } > \text{ and } ]\ ) \\
\langle \mathit{atom} \rangle &::= \mathsf{true} \mid \mathsf{false} \mid \langle \mathit{var} \rangle \mid (\ \langle \mathit{expr} \rangle\ )
\end{aligned}
$$

Compared to the definition given in Section 2.4.1 we have omitted support for arbitrary propositions. Arbitrary subsets of labels are also not supported, but are instead limited to singleton sets containing a label, their complement, signaled by a ! character preceding a label, or the set of all labels, represented by the true action. From now on we will use $\mu$-calculus formulas that follow this syntax. Several mathematical symbols have also been replaced with similar ASCII characters, and precedence rules have been encoded in the grammar.

The two grammars for labelled transition systems and $\mu$-calculus formulas have been chosen to be mostly compatible with the ones used in LCSFE, from which their limitations also come from, in order to simplify a comparison between the two implementation. However the grammar for labelled transition systems has also been extended in order to allow for quoted labels in the labelled transition system grammar, which appeared in some instances used for testing, and more convenient precedence rules for the $\mu$-calculus grammar, which helped when writing some more complex formulas.

### 4.4.1 Performance comparison

We compared the performance of our implementation with respect to LCSFE and mCRL2 on the mCRL2 examples used originally in [8]. All the tests were performed on a computer equipped with an AMD Ryzen 3700x and 32GB of DDR4 RAM running Windows 10. LCSFE and our implementation were compiled using the Rust release profile, which applies optimizations to the code produced.

We started with the “bridge referee” example from mCRL2, modeling the crossing of a bridge by a group of 4 adventurers with different speeds, with the additional restrictions that only 2 explorers can cross the bridge at a time and that they have to carry their only flashlight at every crossing. This leads to a labelled transition system with 102 states and 177 transitions, representing all the possible ways they can try crossing such bridge. The formula to check is $\mu x.\, \langle \mathrm{report}(17) \rangle\ \mathit{true} \vee \langle \mathit{true} \rangle\ x$, representing the fact that all 4 adventurers reach the other side in 17 minutes, which is signaled by the transition report(17). The formula thus checks if it is possible to ever execute such transition.

Using the workflow suggested by mCRL2 we first converted the mCRL2 specification into its internal lps format using the `mcrl22lps` utility:

```
> mcrl22lps bridge-referee.mcrl2 bridge.lps --timings
```

```
- tool: mcrl22lps
  timing:
    total: 0.024
```

Then, we bundled together the lps file and a file holding the formula specified above into a pbes file, another internal format, using the `lps2pbes` utility.

```
> lps2pbes bridge.lps --formula=bridge_report_17.mcf \
  bridge_report_17.pbes --timings
```

```
- tool: lps2pbes
  timing:
    total: 0.016
```

Finally, the `pbes2bool` was used to convert the pbes file into a boolean parity game and solve it. It should be noted that $\mu$-calculus also admits an ad-hoc translation to parity games, which we would expect to be better than our generic approach.

```
> pbes2bool bridge_report_17.pbes -rjittyc --timings
```

```
true
- tool: pbes2bool
  timing:
    instantiation: 0.009495
    solving: 0.000028
    total: 0.038349
```

We then verified the same formula with LCSFE and our implementation. We used mCRL2 again to convert the mCRL2 machine specification to a labelled transition system in AUT format we can use. To do this we reused the lps file previously generated to produce a lts file using the `lps2lts` utility:

```
> lps2lts bridge.lps bridge.lts -rjittyc --timings
```

```
- tool: lps2lts
  timing:
    total: 0.035608
```

The lts file was then converted to an AUT file using the `ltsconvert` utility, which converts between different labelled transition systems formats:

```
> ltsconvert bridge.lts bridge.aut --timings
```

```
- tool: ltsconvert
  timing:
    reachability check: 0.000
    total: 0.002
```

Finally we verified the formula using LCSFE and our implementation

```
> lcsfe-cli mu-ald bridge.aut bridge_report_17.mcf 0
```

```
Preprocessing took: 0.0004837 sec.
Solving the verification task took: 0.0000129 sec.
Result: The property is satisfied from state 0
```

```
> mucalc bridge.aut bridge_report_17.mcf
```

```
Preprocessing took 432.1µs
Solve took 1.1076ms
The formula is satisfied
```

We used this small example to get some empirical evidence that our implementation for $\mu$-calculus is correct, as it gives the same result as the other tools, and to also show the process we used to run all the tools involved. From now on we will omit the specific commands we ran and instead will only report the time required to run them.

We then tested the second formula that was used in [8], which uses the bigger "gossip" labelled transition system, also an example from mCRL2 which models a group of $n$ girls sharing gossips through phone calls. We tested up to $n = 5$, which leads to 9152 states and 183041 transitions, after which the transition system began growing too big. The formula tested was $\nu x.\, \langle \mathit{true} \rangle \mathit{true} \land [\mathit{true}]x$, which represents the lack of deadlocks. It should be noted that formulas checking for absence of deadlock that are satisfied, like this one, are a worst case for local algorithms because they require visiting the whole graph, thus vanishing the advantage of local algorithms which consists in the possibility of visiting only the states that are relevant.

| $n$ | **mCRL2** | **AUT generation** | **Our solver** | **LCSFE** |
|---|---|---|---|---|
| 2 | 67.8 ms | 54.7 ms | 132 µs | 65.5 µs |
| 3 | 68.5 ms | 59.2 ms | 212 µs | 195 µs |
| 4 | 72.0 ms | 117 ms | 2.30 ms | 4.38 ms |
| 5 | 1.47 s | 2.05 s | 202 ms | 5.90 s |

Table 1: Gossips benchmark results

Our implementation scales much better than LCSFE, confirming that the different parity game solving algorithm does make a difference in this case, to the point where the bottleneck becomes the generation of the AUT file, which takes an order of magnitude more time than solving the parity game itself. Compared with mCRL2 our implementation overall takes a similar amount of time, most of which is however spent doing conversions to produce the AUT file using mCRL2 itself. This suggests that lazily generating the labelled transition system might be beneficial, though this was considered out of scope for our work. Overall the pure mCRL2 approach is slightly faster, probably due to the costs of the intermediate conversions to produce the AUT file or the overhead of using a local algorithm in a case where all states must be explored regardless.

We also compared our tool with LCSFE on a set of randomly generated transition systems given the number of states, the number of transitions for each state, and the number of labels. For sake of simplicity the labels have been given a natural number starting from 0 as their name. We used the two tools to test a *fairness* formula on these transition systems, that is a formula in the shape $\nu x.\, \mu y.\, (P \land \langle Act \rangle x) \lor \langle Act \rangle y$, which is satisfied when there exist a path in the labelled transition system where $P$ is true infinitely often. We choose such formula because it represents a common property to verify, it actually uses nested fixpoints, and also because it does not require exploring the whole transition system to verify, hence favoring local solvers. As a formula $P$ we choose $\langle 0 \rangle \mathit{true} \land \langle 1 \rangle \mathit{true} \land \langle 2 \rangle \mathit{true}$, that is we require a state to be able to do three transitions with respectively the labels 0, 1 and 2, because it is an arbitrary condition that we can manipulate how often it is satisfied by changing the number of transitions and labels. We then tested on a number of states ranging from 1000 to 10000, while the number of transitions and labels tested was respectively 10/10 and 20/100, representing a case where the condition $P$ was satisfied quite often and a bit rarer.

| **Size (states/ transitions/labels)** | **Our solver** | **LCSFE** |
|---|---|---|
| 1000 / 10 / 10 | 2.74 ms | 21.8 ms |
| 2500 / 10 / 10 | 5.10 ms | 59.9 ms |
| 5000 / 10 / 10 | 10.2 ms | 120 ms |
| 10000 / 10 / 10 | 18.5 ms | 250 ms |
| 1000 / 20 / 100 | 5.63 ms | 26.6 ms |
| 2500 / 20 / 100 | 13.8 ms | 67.7 ms |
| 5000 / 20 / 100 | 40.1 ms | 142 ms |
| 10000 / 20 / 100 | 48.6 ms | 298 ms |

Table 2: Random LTS benchmark results

Again we can see our tool improving compared to LCSFE, though this time by not so much. This could be attributed to a difference in either the efficiency of the algorithm of the one of the implementation though.

Finally, we also ran our solver on some of the instances in the VLTS benchmark suite to understand the limitations and the strengths of our implementation. For each chosen instance we verified the $\mu$-calculus formulas $\nu x.\, \langle true \rangle true \wedge [true]x$, which checks for absence of deadlocks, and $\mu x.\, \langle true \rangle x \vee (\nu y.\, \langle \text{tau} \rangle y)$, which checks for the presence of livelocks, that is cycles consisting of only tau transitions. For each instance we ran the solver 5 times, ignored the slowest and quickest ones and reported a mean of the remaining 3.

| **Name** | **States count** | **Trans. count** | **Dead-locks?** | **Deadlock solve time** | **Live-locks?** | **Livelock solve time** |
|---|---|---|---|---|---|---|
| `vasy_0_1` | 289 | 1224 | no | 4.93 ms | no | 6.98 ms |
| `cwi_1_2` | 1952 | 2387 | no | 8.74 ms | no | 72.9 ms |
| `vasy_52_318` | 52268 | 318126 | no | 443 s | yes | 75.2 ms |
| `vasy_69_520` | 69754 | 520633 | yes | 122 ms | no | 6.59 s |
| `vasy_720_390` | 720247 | 390999 | yes | 82 ms | no | 3.64 s |

Table 3: VLTS benchmark results

The various labelled transition systems reported in Table 3 have different sizes, and some have deadlocks and livelocks while others do not, which greatly influences the results and makes the various results not directly comparable to one another. We can for example see that checking for the absence of deadlocks when they are not present quickly becomes very slow, like in `vasy_52_318` where in particular we observed that even single iterations of the strategy iteration algorithm become quite slow.

Checking for the presence of livelocks also becomes pretty slow when they are not present, however when they are the local nature of the algorithm allows us to skip checking a lot of positions, ultimately making the algorithm much faster.

In the `cwi_1_2` we observed the computation of play profiles for newly expanded vertices to be especially effective, allowing the valuation step to be performed only once.

The `vasy_720_390` instance is also interesting because it is not connected, with only 87740 states which are actually reachable from the initial one. This is a favorable case for local algorithms, and in fact the time required to verify the formulas is proportional to the number of actually reachable states rather than the full amount.

## 4.5 Testing with bisimilarity

We also briefly tested performance of our bisimilarity checker implementation. For that we used some of the instances mentioned above, in particular `vasy_0_1` and `cwi_1_2` because bigger instances were too slow to check. For each instance we obtained a reduced version of them according to strong bisimilarity and then used our implementation to check whether random states in the original instance were bisimilar with the ones in the reduced one.

| **Name** | **Bisimilar** | **Non bisimilar** | |
|---|---|---|---|
| | | **Min** | **Max** |
| `vasy_0_1` | 15.4 ms | 540 µs | 80 ms |
| `cwi_1_2` | 5.63 s | 1.17 ms | 5.73 s |

Table 4: Bisimilarity benchmark results

We splitted the results based on whether the two states were bisimilar or not, as that influences how many states the local algorithm has to consider. We also noticed that when checking non bisimilar states the time needed varied a lot, which we suspect was due to some states having lot of transitions with the same label and thus causing lot of pairs of states to be checked before becoming distinguishable.

It should be noted that strong bisimilarity admits an algorithm that runs in $O(M \log N)$ [20] time, where $N$ is the number of states and $M$ the number of transitions. In comparison, for states that are bisimilar the solver needs to visit at least as many vertexes as states, similarly to the deadlock case for $\mu$-calculus, leading to a complexity of $O(N \cdot M)$ for each the improvement step, let alone the whole algorithm.

Ultimately the goal was to show that the powerset game was flexible enough, and being able to solve bisimilarity too, although a bit inefficiently, does confirm it.

# 5 Conclusions

We have seen how common systems of fixpoint equations are, especially in model checking, and how we can characterize them using a particular parity game called the powerset game. We have also seen how the moves of this game can be reduced and efficiently expressed using a logic for upward closed sets, which also fits a local algorithm for solving the game. We have then considered a pair of algorithms for locally solving parity games based on strategies.

Our contribution has then been to adapt such algorithms to be used with the powerset game, in particular bridging the conflicting requirements of the two by converting the powerset game to a total parity game, generalizing subgames to consider a subset of edges rather than vertices. This also resulted in the need for a lazy generation of symbolic moves through formula iterators, which also required an adaptation of the simplification process of the corresponding formulas to work on such iterators. In this process we have also introduced a series of small optimizations, most notably the ability to compute play profiles while expanding, which avoids potentially expensive valuation steps. We have then implemented a tool based on our theoretical work, which we have compared against LCSFE, an existing implementation with similar ideas, showing that we have improved over it, in some cases even by orders of magnitude.

## Future work

Although the focus of the game characterization is to be as general as possible, which we have also shown by providing a formulation of bisimilarity using logic formulas, the performance is still quite questionable. A possible improvement in this area could be obtained by integrating different parity game algorithms while keeping the local approach, for example the recent quasi-polynomial algorithms [21, 22] seems to be very good candidates for this.

There also seems to be a lot of room for smarter expansion schemes by combining the informations given by play profiles with symbolic formulas. In particular we believe it could be possible to simplify some formulas such that moves that lead to better improvements are preferred over those that do not. Ultimately the goal would be to include the most critical edges in the expansion as soon as possible, so that the optimal strategy becomes known earlier.

The use of a logic to express symbolic moves also suggests that it might be possible to use symbolic data structures like *Binary Decision Diagrams* (BDDs) to represent them, possibly improving their efficiency. This might in turn open new possibilities to the usage of symbolic moves thanks to making them easier and faster to manipulate, especially in combination with the previous point.

Another challenge involves integrating up-to techniques [23], possibly in a generic way, which could result in speed ups by orders of magnitude. However the problem of efficiently determining when to apply them is still open. We suspect that the previously mentioned use of BDDs might help with this though.

Finally, further work could be done on the adaptation of the different domains. Our $\mu$-calculus and bisimilarity adaptations currently expect a full labelled transition system to be given ahead of time, which sometimes may be too prohibitive and more in general reduce the benefits of a local algorithm. However in principle this could be generated on the fly, avoiding the need to generate parts of the model that are not necessary. The implementation of more and varied domain, like for example the previously mentioned Łukasiewicz $\mu$-calculus and abstract interpretation techniques, is also a possibility for further work to show the generality of the game characterization for solving systems of fixpoint equations.

# Bibliography

[1] D. Kozen, “Results on the Propositional mu-Calculus,” *Theor. Comput. Sci.*, vol. 27, pp. 333–354, 1983, doi: 10.1016/0304-3975(82)90125-6.

[2] D. Sangiorgi, *Introduction to Bisimulation and Coinduction.* USA: Cambridge University Press, 2011.

[3] M. Mio and A. Simpson, “Łukasiewicz µ-Calculus,” in *Proceedings Workshop on Fixed Points in Computer Science, FICS 2013, Turino, Italy, September 1st, 2013*, D. Baelde and A. Carayol, Eds., in EPTCS, vol. 126. 2013, pp. 87–104. doi: 10.4204/EPTCS.126.7.

[4] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, 1977, pp. 238–252. doi: 10.1145/512950.512973.

[5] A. Tarski, “A lattice-theoretical fixpoint theorem and its applications,” *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.

[6] P. Cousot and R. Cousot, “Constructive versions of Tarski's fixed point theorems,” *Pacific Journal of Mathematics*, vol. 82, no. 1, pp. 43–57, 1979.

[7] P. Baldan, B. König, C. Mika-Michalski, and T. Padoan, “Fixpoint games on continuous lattices,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–29, 2019, doi: 10.1145/3290339.

[8] A. Flori, “A Local Algorithm for Systems of Fixpoint Equations: Study and Implementation,” Master's thesis, Università di Padova, 2022. [Online]. Available: https://hdl.handle.net/20.500.12608/61406

[9] G. Mazzocchin, “Solving fixpoint equations using Progress Measures,” Master's thesis, Università di Padova, 2019.

[10] J. Vöge and M. Jurdziński, “A Discrete Strategy Improvement Algorithm for Solving Parity Games,” in *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, E. A. Emerson and A. P. Sistla, Eds., in Lecture Notes in Computer Science, vol. 1855. Springer, 2000, pp. 202–215. doi: 10.1007/10722167_18.

[11] O. Friedmann and M. Lange, “Two Local Strategy Iteration Schemes for Parity Game Solving,” *Int. J. Found. Comput. Sci.*, vol. 23, no. 3, pp. 669–685, 2012, doi: 10.1142/S0129054112400333.

[12] P. Baldan, B. König, T. Padoan, and C. Mika-Michalski, “Fixpoint Games on Continuous Lattices (Full version),” *CoRR*, 2018, [Online]. Available: http://arxiv.org/abs/1810.11404

[13] E. A. Emerson and C. S. Jutla, “Tree Automata, Mu-Calculus and Determinacy (Extended Abstract),” in *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, IEEE Computer Society, 1991, pp. 368–377. doi: 10.1109/SFCS.1991.185392.

[14] W. Zielonka, “Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees,” *Theor. Comput. Sci.*, vol. 200, no. 1–2, pp. 135–183, 1998, doi: 10.1016/S0304-3975(98)00009-7.

[15] I. Walukiewicz, “Monadic second-order logic on tree-like structures,” *Theor. Comput. Sci.*, vol. 275, no. 1–2, pp. 311–346, 2002, doi: 10.1016/S0304-3975(01)00185-2.

[16] “Rust programming language.” [Online]. Available: https://www.rust-lang.org/

[17] “PGSOLVER.” [Online]. Available: https://github.com/tcsprojects/pgsolver

[18] “Oink.” [Online]. Available: https://github.com/trolando/oink

[19] “AUT format specification.” [Online]. Available: https://cadp.inria.fr/man/aut.html

[20] D. N. Jansen, J. F. Groote, J. J. A. Keiren, and A. Wijs, “A simpler O(m log n) algorithm for branching bisimilarity on labelled transition systems,” *CoRR*, 2019, [Online]. Available: http://arxiv.org/abs/1909.10824

[21] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan, “Deciding parity games in quasipolynomial time,” in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, H. Hatami, P. McKenzie, and V. King, Eds., ACM, 2017, pp. 252–263. doi: 10.1145/3055399.3055409.

[22] P. Parys, “Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time,” in *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany*, P. Rossmanith, P. Heggernes, and J.-P. Katoen, Eds., in LIPIcs, vol. 138. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 1–13. doi: 10.4230/LIPICS.MFCS.2019.10.

[23] P. Baldan, B. König, and T. Padoan, “Abstraction, Up-to Techniques and Games for Systems of Fixpoint Equations,” *CoRR*, 2020, doi: 10.48550/arXiv.2003.08877.