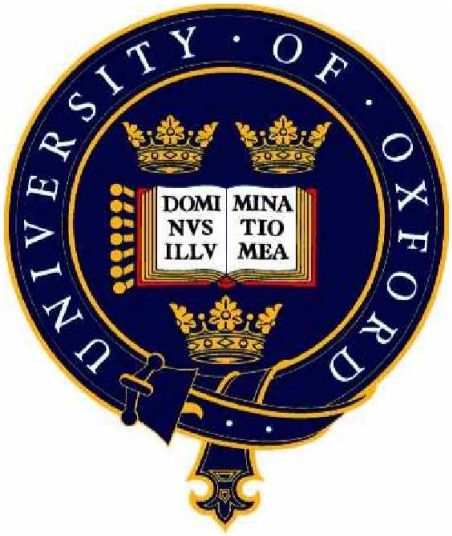
A Comparative Study of Algorithms for Solving Büchi Games

Rachel Bailey
Merton College
Supervisor: Professor Luke Ong



Submitted as part of
Master of Computer Science
Computing Laboratory
University of Oxford

September 2010

Abstract

Recently, as the complexity of software has increased, model checking has become an increasingly important topic in the field of computer science. In the last three decades model checking has emerged as a powerful approach towards formally verifying programs. Given an abstract model of a program, a model checker verifies whether the model meets a given specification. A particularly expressive logic used for specifying programs is a type of temporal logic known as the modal μ-calculus. Determining whether a given model satisfies a modal μ-calculus formula is known as the modal μ-calculus model checking problem. A popular method for solving this problem is by solving parity games. These are a type of two player infinite game played on a finite graph. Determining a winner for such a game has been shown to be polynomial time equivalent to the modal μ-calculus model checking problem.

This dissertation concentrates on comparing algorithms which solve a specialised type of parity game known as a Büchi game. This type of game can be used to solve the model checking problem for a useful fragment of the modal μ-calculus. Implementations of three Büchi game solving algorithms are given and are compared. These are also compared to existing parity game solvers. In addition to this, a description of their integration into the model checker Thors is given.

Acknowledgements

I would like to thank Luke Ong for supervising this project. I would also like to express my deepest appreciation to Steven and Robin for their invaluable help during the past four months. For proofreading my dissertation at a moments notice I would also like to thank my brother Andrew.

Contents

1 Introduction

As the complexity of software increases, so does the need for automatic verification. In the case of safety critical software such as that for nuclear reactors and air traffic control systems this is particularly apparent. In order to verify such software formally, a technique known as model checking can be used. Given an abstract model of a system, a model checker verifies automatically whether the model meets a given specification. This abstract model should therefore describe the relevant behaviours of the corresponding system. In the case of systems where there is an ongoing interaction with their environment (known as reactive systems), transition systems are usually used as models. The specification for such a system is usually given in the form of a logical formula.

The modal μ-calculus is a logic which extends modal logic with fixpoint operators. The fixpoint operators are used in order to provide a form of recursion for specifying temporal properties. It was first introduced in 1983 by Dexter Kozen [10] and has since been the focus of much attention due to its expressive power. In fact, the modal μ-calculus subsumes most other temporal logics such as LTL, CTL, and CTL*. It is extensively used for specifying properties of transition systems and hence much time has been invested into finding efficient solutions to the corresponding model checking problem.

Given a transition system T, an initial state s_0 and a modal μ-calculus formula φ, the model checking problem asks whether the property described by φ holds for the transition system T at state s_0. The problem was shown to be in NP [5] and thus, since it is closed under negation, it is in NP$\cap$co-NP. It is not yet known whether this class of problems is polynomial and currently the best known algorithms for solving the model checking problem are exponential in the alternation depth.

The alternation depth of a formula, informally, is the maximum number of alternations between maximum and minimum fixpoint operators. Formulas with a high alternation depth are well known to be difficult to interpret and in practice, formulas of alternation depth greater than 2 are not frequently used.

One popular method of solving the model checking problem for the modal μ-calculus is by using parity games since it can be shown that determining the winner of a parity game is polynomial time equivalent to this model checking problem.

A parity game is a two-player infinite game played on a finite directed graph. Each vertex in the graph is owned by one of the two players called Eloïse and Abelard. Play begins at a given initial state and proceeds as follows: If a vertex is owned by Eloïse then Eloïse picks a successor vertex and play continues from there, whereas if a vertex is owned by Abelard then Abelard picks a successor

vertex. A play either terminates at a vertex with no successors or it does not terminate, resulting in an infinite play. If a play terminates then the winner of the game is the player who does not own the final vertex. If a play does not terminate then the winner is determined using a given priority function which maps the set of vertices to the natural numbers. The set of vertices which occur infinitely often is calculated and the minimum priority of those vertices is found. If this minimum priority is even then Eloïse wins the game whereas if the minimum priority is odd then Abelard wins the game.

When considering the model checking problem, the number of priorities in the corresponding parity game is equal to the alternation depth of the modal μ-calculus formula. Thus, in terms of complexity, the best algorithms for solving parity games are exponential in the number of priorities used. For example, a classical algorithm introduced by McNaughton in [12] has a time complexity of $O(mn^{p-1})$ where m is the number of edges, n is the number of vertices and p is the number of priorities. This was later improved by Jurdziński [7] to $O(pm(n/p)^{p/2})$. More recently, an algorithm was presented by Schewe [15] which has a time complexity of approximately $O(mn^{p/3})$ for games with at least 3 priorities. Currently, for games with 2 priorities, however, the time complexity of $O(mn)$ using McNaughton's algorithm has not been improved upon.

Parity games with no more than 2 priorities are divided into two types of games: Büchi games and co-Büchi games. Büchi games correspond to games where the priorities 0 and 1 are used and co-Büchi games correspond to games where the priorities 1 and 2 are used. These types of games are vital in relation to the model checking problem since they correspond to a fragment of the modal μ-calculus in which formulas have alternation depth 2 or less. As previously remarked, it is rare to use a formula with greater alternation depth and notably, this fragment of the modal μ-calculus subsumes the popular logic CTL*. In addition to this, co-Büchi games can easily be solved using Büchi games, thus finding efficient algorithms for solving Büchi games is useful.

In [2], Chatterjee, Henzinger and Piterman investigated this topic, presenting new algorithms specifically for solving Büchi games. Although these algorithms do not improve on the time complexity of $O(mn)$, they do provide an alternative approach for solving these games, with a marked improvement in complexity for specific families of Büchi games. In one algorithm presented, for a specific family of Büchi games with a fixed number of outgoing edges, the time complexity was shown to be $O(n)$ in comparison to the time complexity of $O(n^2)$ for the classical algorithm. Also, it was shown that this algorithm performs at worst $O(m)$ more work than the classical algorithm.

For the interest of comparison of these different approaches for solving Büchi games, this dissertation implements two of the algorithms from [2] and also

implements the more classical approach to solving Büchi games in OCaml. The advantages and disadvantages of each is investigated, and all are compared to an implementation [6] of three parity game solvers [18, 7, 16]. These have worst case time complexities $O(mn^2)$, $O(mn)$ and $O(2^m mn)$ when restricted to Büchi games. Although one of these solvers has the same worst case time complexity as the algorithms for solving Büchi games, the specialisation of the latter should result in a faster performance. In addition to comparing the algorithms against parity game solvers, they are integrated into the model checker Thors [14]. Thors is used for model checking higher-order recursion schemes (HORS). Currently, a construction of this problem to a subset of Büchi games called weak Büchi games is used, allowing for model checking against the alternation-free modal μ-calculus. By integrating a Büchi game solver, HORS can be checked against modal μ-calculus formulas of alternation depth 2.

Section 2 introduces the concepts of parity games and the modal μ-calculus and shows how the model checking problem can be represented by a parity game. It then relates this to Büchi games. Section 3 describes the algorithms presented in [2] and explains their correctness and complexity. Section 4 describes the implementation of the Büchi game solving algorithms. Section 5 describes the integration of a solver into Thors and Section 6 tests the efficiency of each of the solvers comparing them against each other and against existing parity game solvers. Finally, Section 7 draws conclusions about the findings of this dissertation.

2 Preliminaries

This section begins by defining the modal μ-calculus, parity games and the concepts associated with them. It then shows how the model checking problem for the modal μ-calculus can be represented as a parity game and for which fragment of the modal μ-calculus the model checking problem can be represented by a Büchi game.

2.1 The Modal μ-Calculus

The modal μ-calculus [10] is a general and expressive logic that subsumes logics such as PDL, CTL, and CTL*. It is an extension of modal logic with least and greatest fixpoint operators and is useful for specifying the properties of transition systems which are defined below as given in [1].

Definition 2.1 *A* **Labelled transition system** *over a set of propositions P and a set of labels $\mathcal{L}$ is a directed graph whose edges are labelled with elements of $\mathcal{L}$ and whose vertices are labelled with elements of 2^P. More formally, a labelled transition system is a triple $T = \langle S, \rightarrow, \rho \rangle$ where:*

- *S is the set of states.*
- *$\rightarrow \subseteq S \times \mathcal{L} \times S$ is the transition relation (This is usually written as $s \xrightarrow{a} t$ for each $(s, a, t) \in \rightarrow$).*
- *$\rho : P \longrightarrow 2^S$ is an interpretation of the propositional variables where each propositional variable is mapped to the set of states where it holds true.*

Definition 2.2 *Given a set Var of variables, the set of* **modal μ-calculus formulas** *(denoted by L_μ) is defined inductively as follows (as given in [1]) :*

- *All $p \in P$ are in L_μ*
- *All $Z \in Var$ are in L_μ*
- *If φ_1 and φ_2 are in L_μ then $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are in L_μ.*
- *If $\varphi \in L_\mu$ then for all $a \in \mathcal{L}$, $[a]\varphi$ and $\langle a \rangle \varphi$ are in L_μ.*
- *If $\varphi \in L_\mu$ then $\neg\varphi$ is in L_μ*
- *If $Z \in Var$ and $\varphi(Z)$ is a formula in L_μ then $\mu Z.\varphi(Z)$ and $\nu Z.\varphi(Z)$ are formulas in L_μ. (A variable is defined to be free or bound in the usual sense where μ and ν are the binding operators.)*

The symbols μ and ν are known as least and greatest fixpoint operators respectively.

The semantics of this logic, given a transition system T, maps formulas to sets of states in $\mathcal{P}(S)$. Thus, given a variable Z, a formula $\varphi(Z)$ can be viewed as a function from $\mathcal{P}(S)$ to itself.

We require that $\varphi(Z)$ is a monotonic function with respect to Z, so as to ensure the existence of unique minimal and maximal fixpoints for φ. In order to guarantee monotonicity, we require that all formulas are in positive normal form.

A formula is in positive form if negation is only applied to atomic propositions. It is in positive normal form if it is in positive form and additionally all bound variables are distinct (i.e. for fixpoint operators σ_1 and σ_2, if $\sigma_1 X.\psi_1$ and $\sigma_2 Y.\psi_2$ both occur in a positive normal formula φ, then $X \neq Y$). By using De Morgan laws and renaming bound variables if required, we can assume that all modal μ-calculus formulas are in positive normal form. This assumption is made during the construction of an equivalent parity game in Section 2.3.

Definition 2.3 *Given a labelled transition system T and a valuation of variables $Val : Var \longrightarrow 2^S$, we can define the* **semantics of formulas in $\mathbf{L}_\mu$** *inductively as follows (as given in [1]):*

$$
\begin{aligned}
||p||^T_{Val} &:= \rho(p) \\
||Z||^T_{Val} &:= Val(Z) \\
||\varphi_1 \vee \varphi_2||^T_{Val} &:= ||\varphi_1||^T_{Val} \cup ||\varphi_2||^T_{Val} \\
||\varphi_1 \wedge \varphi_2||^T_{Val} &:= ||\varphi_1||^T_{Val} \cap ||\varphi_2||^T_{Val} \\
||[a]\varphi||^T_{Val} &:= \{s \mid \forall t \in S,\ s \xrightarrow{a} t \implies t \in ||\varphi||^T_{Val}\} \\
||\langle a \rangle \varphi||^T_{Val} &:= \{s \mid \exists t \in S \text{ s.t. } s \xrightarrow{a} t \wedge t \in ||\varphi||^T_{Val}\} \\
||\neg\varphi||^T_{Val} &:= S \backslash ||\varphi||^T_{Val} \\
||\mu Z.\varphi||^T_{Val} &:= \bigcap \{U \subseteq S \mid ||\varphi||^T_{Val[Z \mapsto U]} \subseteq U\} \\
||\nu Z.\varphi||^T_{Val} &:= \bigcup \{U \subseteq S \mid ||\varphi||^T_{Val[Z \mapsto U]} \supseteq U\}
\end{aligned}
$$

The valuation $Val[Z \mapsto U]$ is given by:

$$
Val[Z \mapsto U](X) := \begin{cases} U & \text{If } X = Z \\ Val(X) & \text{If } X \neq Z \end{cases}
$$

Given a state $s \in S$ and a modal μ-calculus formula φ, we say that $s \vDash^T_{Val} \varphi$ if $s \in ||\varphi||^T_{Val}$.

The semantics of all these formulas are fairly intuitive except for those of the least and greatest fixpoint operators which are notoriously difficult to interpret. The expressive power of the modal μ-calculus lies in the use of these fixpoint operators which are used to express temporal properties recursively.

Informally, the μ fixpoint operator is used for finite looping, whereas ν is used for looping. For example, consider the following two properties we may want to express:

1. "On all a-labelled paths, ϕ holds until φ holds" (and φ does eventually hold).
2. "On all a-labelled paths, ϕ holds while φ does not hold" (φ does not have to eventually hold).

Thus, property 1 holds for states where a finite number of a-transitions are made to states where ϕ holds, after which an a-transition is made where φ holds. This requirement of a finite number of a-transitions indicates the use of the μ fixpoint operator.

Conversely, property 2 holds for states where either an infinite number of a-transitions are made to states where ϕ holds, or property 1 holds. This indicates the use of the ν fixpoint operator.

Suppose, given a transition system T, that Z_1 is a variable with a valuation corresponding to the set of states on which property 1 holds and Z_2 is a variable that corresponds to the set of states on which property 2 holds.

The set $||Z_1||^T_{Val}$ can be expressed recursively as follows. For a state z, if φ holds then $z \in ||Z_1||^T_{Val}$. Also, if ϕ holds and all a-labelled transitions go to a state contained in the valuation of Z_1 then $z \in ||Z_1||^T_{Val}$. Thus:

$$z \in ||Z_1||^T_{Val} \Longleftarrow z \in ||\varphi||^T_{Val} \text{ or } (z \in ||\phi||^T_{Val} \text{ and } z \in ||[a]Z_1||^T_{Val})$$

or:

$$||Z_1||^T_{Val} \supseteq ||\varphi \vee (\phi \wedge [a]Z_1)||^T_{Val}$$

Sets like $||Z_1||^T_{Val}$ for which this condition holds are pre-fixed points for the function $\psi(U) := ||\varphi \vee (\phi \wedge [a]Z)||^T_{Val[Z \mapsto U]}$. More precisely, $||Z_1||^T_{Val}$ corresponds to the least such pre-fixed point or rather

$$\begin{aligned} ||Z_1||^T_{Val} &= \bigcap \{U \subseteq S \mid U \supseteq ||\varphi \vee (\phi \wedge [a]Z)||^T_{Val[Z \mapsto U]}\} \\ &:= ||\mu Z.\varphi \vee (\phi \wedge [a]Z)||^T_{Val} \end{aligned}$$

In the case of property 2, the complement of the set $||Z_2||^T_{Val}$ can be expressed recursively as follows. For a state z, if φ does not hold and ϕ does not hold then $z \notin ||Z_2||^T_{Val}$. Also, if φ does not hold and there exists an a-transition to a state in the complement of $||Z_2||^T_{Val}$ then $z \notin ||Z_2||^T_{Val}$. Thus:

$$z \notin ||Z_2||^T_{Val} \Longleftarrow (z \notin ||\varphi||^T_{Val} \text{ and } z \notin ||\phi||^T_{Val}) \text{ or } (z \notin ||\varphi||^T_{Val} \text{ and } z \notin ||[a]Z_2||^T_{Val})$$

Taking the contrapositive we get:

$$z \in ||Z_2||^T_{Val} \Longrightarrow z \in ||\varphi||^T_{Val} \text{ or } (z \in ||\phi||^T_{Val} \text{ and } z \in ||[a]Z_2||^T_{Val})$$

or:

$$||Z_2||^T_{Val} \subseteq ||\varphi \vee (\phi \wedge [a]Z_2)||^T_{Val}$$

Sets for which this condition holds are post-fixed points for the function $\psi(U) := ||\varphi \vee (\phi \wedge [a]Z)||^T_{Val[Z \mapsto U]}$. In particular, $||Z_2||^T_{Val}$ is the greatest such post-fixed point and hence:

$$\begin{aligned} ||Z_2||^T_{Val} &= \bigcup \{U \subseteq S \mid ||\varphi||^T_{Val[Z \mapsto U]} \supseteq U\} \\ &:= ||\nu Z.\varphi \vee (\phi \wedge [a]Z)||^T_{Val} \end{aligned}$$

Thus, property 1, an instance of finite looping, is expressed using the μ fixpoint operator, whereas property 2 is expressed using the ν fixpoint operator.

The fact that $\psi(U) := ||\varphi \vee (\phi \wedge [a]Z)||^T_{Val[Z \mapsto U]}$ is monotonic, guarantees the existence of a least pre-fixed point and a greatest post-fixed point. It can be proved that these points coincide with the least and greatest fixed points, hence μ and ν are referred to as least and greatest fixpoint operators respectively.

Modal μ-calculus formulas become more difficult to interpret (and more powerful) when fixpoint operators are nested inside other fixpoint operators. For example, consider the following formula as given in [1]:

$$||\mu Y.\nu Z.(P \wedge [a]Y) \vee (\neg P \wedge [a]Z)||^T$$

After some thought, this can be seen to correspond to the set of states in T from which "on any a-path, P is true only finitely often".

As described in [17], an important property of a modal μ-calculus formula is its fixpoint alternation depth defined later in this section. In particular, it is relevant to the complexity of algorithms for solving the model checking problem.

The model checking problem for the modal μ-calculus is defined as follows:

Given a transition system T, an initial state $s_0 \in S$ and a modal μ-calculus formula φ, is it true that $s_0 \models^T_{Val} \varphi$?

The best known algorithms for this model checking problem are exponential in the alternation depth.

In its simplest form, the alternation depth of a formula is defined as the maximum number of alternations between nested least and greatest fixpoint operators. The definition used here (as in [17]) was introduced by Niwinski [13] and is a stronger and more useful definition then simply counting syntactic alternations.

We begin with the following definition using the notation $\varphi_1 \geq \varphi_2$ to denote that φ_2 is a subformula of φ_1 and $\varphi_1 > \varphi_2$ to denote that φ_2 is a proper subformula of φ_1:

Definition 2.4 *An* **alternating chain** *is a sequence $\langle \sigma_1 Z_1.\psi_1, \sigma_2 Z_2.\psi_2, ..., \sigma_l Z_l.\psi_l \rangle$ such that:*

- $\sigma_1 Z_1.\psi_1 > \sigma_2 Z_2.\psi_2 > ... > \sigma_l Z_l.\psi_l$
- *For all $1 \leq i < l$, if $\sigma_i = \mu$ then $\sigma_{i+1} = \nu$ and similarly if $\sigma_i = \nu$ then $\sigma_{i+1} = \mu$.*
- *For all $1 \leq i < l$, the variable Z_i occurs free in every formula ψ with $\psi_i \geq \psi \geq \psi_{i+1}$*

Using this definition, the alternation depth of a formula is defined as follows:

Definition 2.5 *The* **alternation depth** *of a formula φ is the length of the longest alternating chain contained in φ.*

Using this, we can classify modal μ-calculus formulas as follows:

Definition 2.6 (Niwinski [13]) *The classes Σ^μ_n and Π^μ_n for each $n \in \mathbb{N}$ are defined as follows:*

- *A formula φ is said to be in the classes Σ^μ_0 and Π^μ_0 if φ contains no fixpoint operators.*
- *For each $n > 0$, φ is said to be in Σ^μ_n (respectively Π^μ_n) if φ has alternation depth n or less and all alternating chains of length n contained in φ begin with a μ (respectively ν) fixpoint formula.*

For example, consider the following formulas:

$$\begin{aligned}\varphi_1 &:= \nu X.\mu Y.[a]Y \vee (p \wedge \langle a \rangle X) \\ \varphi_2 &:= \nu X.(\mu Y.[a]Y) \vee (p \wedge \langle a \rangle X)\end{aligned}$$

Using the definition given, φ_1 has alternation depth 2 and is in Π_2^μ but is not in Σ_2^μ since there exists an alternating chain of length 2, contained in φ_1, which begins with a ν fixpoint formula. The formula φ_2, however, has alternation depth 1 and is thus in both Π_2^μ and Σ_2^μ. Notice that using the simple definition of alternation depth, both formulas would have alternation depth 2 since the μ fixpoint formula is nested within the ν fixpoint formula.

2.2 Parity Games

In this section, the notion of parity games is introduced and we demonstrate how the model checking problem for the modal μ-calculus can be reduced to the problem of solving a parity game. The definitions given in this section are similar to those given in [2].

Definition 2.7 *A* **game graph** $G = (V, E, \lambda)$ *is a directed graph where* V *is the set of vertices and* E *is the set of edges.* $\lambda : V \to \{\exists, \forall\}$ *is a mapping that identifies an owner for each vertex from one of two players. If* $\lambda(v) = \exists$ *then* v *is owned by the player Eloïse and if* $\lambda(v) = \forall$ *then* v *is owned by the player Abelard.*

We use $E(v)$ to denote the set of all successor vertices for v i.e. $E(v) = \{u \in V | (v, u) \in E\}$

Definition 2.8 *A* **parity game** *is a tuple* (G, v_0, Ω) *where* $G = (V, E, \lambda)$ *is a game graph,* $v_0 \in V$ *is the initial vertex and* $\Omega : V \to \{0, ..., P\}$ *for some* $P \in \mathbb{N}$ *is the priority function.*

Given a parity game $\mathcal{G} = (G, v_0, \Omega)$, a **play** is a path in the game graph beginning at the initial vertex v_0. For each vertex $v \in V$, the path is extended as follows: if v is an Eloïse vertex then Eloïse chooses which $u \in E(v)$ to traverse next, whereas if v is an Abelard vertex then Abelard chooses (provided that $E(v) \neq \emptyset$). If a vertex has no outgoing edges then play terminates and the path is finite. If this does not occur, then the corresponding path is infinite.

More formally, each play is represented by either a finite or infinite sequence of vertices $\omega = \langle v_0, v_1, v_2, ... \rangle$ such that for all v_i in the sequence, $v_i \in E(v_{i-1})$. If the sequence is finite with final element v_n for some $n \in \mathbb{N}$ then $E(v_n) = \emptyset$. If $E(v_i) \neq \emptyset$ then if $\lambda(v_i) = \exists$, Eloïse chooses v_{i+1} else Abelard chooses.

Definition 2.9 *A play* ω *is defined to be* **winning** *for Eloïse in the following cases:*

- *If ω is finite and the play ends in an Abelard vertex.*
- *If $\omega = \langle v_0, v_1, v_2, ... \rangle$ is infinite and the minimum number in the set $\{i \mid i$ occurs in the sequence $\Omega(v_0), \Omega(v_1), \Omega(v_2)...$ infinitely many times$\}$ is even (This value is denoted by* $\min(\inf\{\Omega(v_0), \Omega(v_1), \Omega(v_2), ...\})$.

A play ω is defined to be winning for Abelard if it is not winning for Eloïse.

Definition 2.10 *A* **strategy** *for Eloïse is a mapping $\sigma : V^* \cdot \{v \mid \lambda(v) = \exists\} \rightarrow V$ that defines how Eloïse should extend the current play. In particular $\sigma(\langle v_0, v_1, ...v_n \rangle) \in E(v_n)$. A strategy π for Abelard is similarly defined.*

Thus, given a strategy σ for Eloïse and a strategy π for Abelard, there exists a unique play $\omega(\sigma, \pi) = \langle v_0, v_1, v_2, ... \rangle$ such that for each v_i in the play, if $\lambda(v_{i-1}) = \exists$ then $\sigma(\langle v_0, v_1, v_2, ...v_{i-1} \rangle) = v_i$. Similarly, if $\lambda(v_{i-1}) = \forall$ then $\pi(\langle v_0, v_1, v_2, ...v_{i-1} \rangle) = v_i$.

Definition 2.11 *A strategy is said to be* **memoryless** *if each move only depends on the current vertex. More specifically, in the case of Eloïse, a strategy is memoryless if it can be defined as a function $\sigma : \{v \mid \lambda(v) = \exists\} \rightarrow V$ where $\sigma(v) \in E(v)$ (A memoryless strategy for Abelard is analogously defined).*

Definition 2.12 *Given a parity game $\mathcal{G} = (G, v_0, \Omega)$, Eloïse is said to have a* **winning strategy** *if there exists a strategy σ such that for all possible Abelard strategies π, the corresponding play $\omega(\sigma, \pi)$ is winning for Eloïse. A winning strategy for Abelard can be similarly defined.*

In particular, it was shown in [4] that for a parity game, a *memoryless* winning strategy always exists for one of the players.

The purpose of solving a parity game is to determine which player has a winning strategy given the initial vertex v_0. In some definitions, the solution of a parity game is defined as determining which player has a winning strategy for all possible initial vertices in the game graph. For such a global solution, the sets W_E and W_A are given which partition the set of vertices V into vertices from which Eloïse and Abelard have winning strategies respectively. In addition to this, the corresponding winning strategy may be given in some solutions. For the purposes of implementing the game solver into the model checker THORS, we will concentrate on solving games where an initial vertex is specified. In Section 6.2, however, it is necessary to provide a global solution and return the corresponding winning strategy so that a fair comparison to existing parity game solvers can be made.

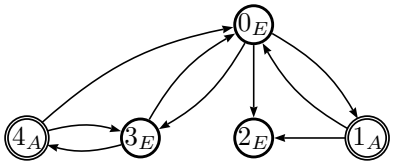
Definition 2.13 *A* **Büchi game** *is a game $\mathcal{G}_\mathcal{B} = (G, v_0, F)$ where G is a game graph, v_0 is an initial vertex and $F \subseteq V$ is the accepting set. A play is defined analogously to that of a parity game.*
A play is defined to be winning for Eloïse in the following cases:

- *ω is finite and the play ends in an Abelard vertex.*
- *$\omega = \langle v_0, v_1, v_2, ... \rangle$ is infinite and* $\inf(v_0, v_1, v_2, ...) \cap F \neq \emptyset$

If neither of these cases holds then the play is defined to be winning for Abelard.

Note that a Büchi game $\mathcal{G}_{\mathcal{B}} = (G, v_0, F)$ can easily be shown to be equivalent to a restricted type of parity game $\mathcal{G} = (G, v_0, \Omega)$ where Ω maps each vertex to a priority of 1 or 0 (i.e. $\Omega : V \to \{0, 1\}$). More specifically, if $v \in F$ then $\Omega(v) = 0$ and if $v \notin F$ then $\Omega(v) = 1$.

In order to demonstrate a Büchi game consider the following example:



The game $\mathcal{G}_{\mathcal{B}} = (G, v_0, F)$, with $G = (V, E, \lambda)$ where V and E are as shown in the diagram. For $v \in \{1_A, 4_A\}$, $\lambda(v) = \forall$ else $\lambda(v) = \exists$. The initial vertex $v_0 = 0_E$ and the accepting set $F = \{1_A, 4_A\}$. A play, therefore, begins at the vertex 0_E and so Eloïse makes the first move. She has three successor vertices to choose from. If Eloïse selects 2_E, then Abelard wins the game since 2_E is owned by Eloïse and has no successor vertices. If Eloïse selects 1_A, then Abelard owns this vertex and hence makes the next move. The vertex 2_E is a successor of 1_A, so Abelard could select 2_E resulting in a win for him. Thus at the beginning of the game, two of the three vertices Eloïse can pick from result in a win for Abelard provided he employs the correct strategy. It remains to consider the final successor vertex of 0_E. If Eloïse moves to 3_E, then since this vertex is owned by her, she makes the next move. From 3_E, Eloise has two choices. Either move back to 0_E, or move to 4_A. In the first case, if she continues with the same strategy then play will loop between 0_E and 3_E. Since neither of these vertices are accepting this would result in a win for Abelard. If Eloïse moves to 4_A from 3_E then the next move is determined by Abelard. If Abelard chooses to move to 0_E, then by repeating the same strategy the play ends up in an infinite loop of vertices including the vertex 4_A which is accepting. This is similarly the case if Abelard chooses to move to 3_E. Thus if the initial vertex is 0_E then Eloïse has a winning strategy. Provided that at 0_E, Eloïse moves to 3_E and at 3_E, Eloïse moves to 4_A, Eloïse will always win regardless of what strategy Abelard employs.

Definition 2.14 *A* **co-Büchi game** *is a game $\mathcal{G}_{\mathcal{CB}} = (G, v_0, F)$ where G is a game graph, v_0 is an initial vertex and $F \subseteq V$. A play is defined analogously to that of a parity game.*

In contrast to that of a Büchi game, however, a play is defined to be winning for Eloïse in the following cases:

- *If ω is finite and the play ends in an Abelard vertex.*
- *If $\omega = \langle v_0, v_1, v_2, ... \rangle$ is infinite and $inf(v_0, v_1, v_2, ...) \cap F = \emptyset$*

As with Büchi games, if neither of these cases holds then the play is defined to be winning for Abelard.

Note that, similarly to the case of Büchi games, a co-Büchi game $\mathcal{G}_{\mathcal{CB}} = (G, v_0, F)$ can be shown to be equivalent to a restricted type of parity game $\mathcal{G} = (G, v_0, \Omega)$ where Ω maps each vertex to a priority of 1 or 2. In particular $\Omega(v) = 2$ for each $v \in V \backslash F$ and $\Omega(v) = 1$ for each $v \in F$.

2.3 Parity Games in Relation to the Modal μ-Calculus

We now demonstrate how the model checking problem for the modal-μ calculus can be reduced to the problem of solving a parity game. The explanation given is similar to that in [17].

Given a closed modal μ-calculus formula φ in positive normal form, with alternation depth D, a labelled transition system $T = (S, \rightarrow, \rho)$ and an initial state s_0, we define the parity game $\mathcal{G} = (G, v_0, \Omega)$ as follows, using $Sub(\varphi)$ to denote the set of all subformulas for φ:

- $G = (V, E, \lambda)$ where $V = Sub(\varphi) \times S$.
- For each $v \in V$, $E(v)$ is defined as follows:
  - If $v = (p,s)$, $(\neg p,s)$, then $E(v) = \emptyset$
  - If $v = (\psi_1 \vee \psi_2, s)$ or $(\psi_1 \wedge \psi_2, s)$ then $E(v) = \{(\psi_1, s), (\psi_2, s)\}$
  - If $v = ([a]\psi, s)$ or $(\langle a \rangle \psi, s)$ then $E(v) = \{(\psi, t) | s \xrightarrow{a} t\}$
  - If $v = (\sigma Z.\psi, s)$ for a fixpoint operator σ then $E(v) = \{(Z, s)\}$
  - If $v = (Z, s)$ for a bound variable Z, where $\sigma Z.\psi$ is the corresponding fixpoint subformula of φ, then $E(v) = \{(\psi, s)\}$.
- For each $v \in V$, $\lambda(v)$ is defined as follows:
  - If $v = (p, s)$, where $s \notin \rho(p)$ or $v = (\neg p, s)$ where $s \in \rho(p)$ then $\lambda(v) = \exists$
  - If $v = (\psi_1 \vee \psi_2, s)$ then $\lambda(v) = \exists$
  - If $v = (\langle a \rangle \psi, s)$ then $\lambda(v) = \exists$

  - For any other $v \in V$, $\lambda(v) = \forall$. (Note that since the vertices of type (Z, s) and $(\sigma Z.\psi, s)$ have exactly one outgoing edge these vertices can be owned by either player)

- $v_0 = (\varphi, s_0)$
- For each $v \in V$, $\Omega(v)$ is defined as follows:
  - If $\sigma Z.\psi$ is a subformula of φ then $\Omega((Z, s))$ depends on l, where l is the position of $\sigma Z.\psi$ in the longest alternating chain it appears in within φ. If the initial element of this longest alternating chain is a μ fixpoint formula then $\Omega((Z, s)) = l$. If, however, the initial element of this alternating chain is a ν fixpoint formula, then $\Omega((Z, s)) = l - 1$.
  - For all other subformulas ψ of φ, if there exists an alternating chain in φ of length D that begins with a μ fixpoint formula then $\Omega(\psi, s) = D$ else $\Omega(\psi, s) = D - 1$, where D is the alternation depth of φ.

This parity game corresponds to the model checking problem in the sense that if Eloïse has a winning strategy then $s_0 \vDash^T \varphi$ and if Abelard has a winning strategy then $s_0 \nvDash^T \varphi$. Thus, the task of the player Eloïse is to try and prove the property φ holds for the transition system T and the task of the player Abelard is to disprove the property holds.

Essentially, for each vertex (ψ, s), Eloïse tries to prove the property ψ holds from state s and Abelard tries to disprove this. Beginning at the initial vertex (φ, s_0), by traversing the graph, the problem is reduced to proving the property ψ for state s for the current vertex (ψ, s).

This reduction of the problem is guided by Eloïse and Abelard in such a way that if an existential property needs to be proved (i.e the formula is of the form $(\langle a \rangle \psi, s)$ or $(\psi_1 \vee \psi_2, s)$) then Eloïse picks the next vertex. Conversely if a universal property needs to be proved (i.e. the formula is of the form $([a]\psi, s)$ or $(\psi_1 \wedge \psi_2, s)$) then Abelard picks the next vertex. This ensures that, provided the correct strategy is employed, the truth of whether the original property holds is maintained along each vertex travelled.

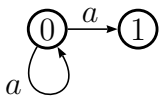
The game can only enter an infinite play for vertices corresponding to subformulas of a fixpoint formula. In particular, if such an play occurs then the winner is determined by the vertices of the type (Z, s) that occur infinitely often (for $Z \in Var$). Each of the corresponding variables for each of these vertices is bound to a fixpoint operator. Note that by construction of the game, the fixpoint operators binding these variables will be nested within one

another. The variable bound to the outermost fixpoint operator, of those that occur infinitely often, determines the winner of the game.

The priority function Ω of the game is defined so that if the variable bound to the outermost fixpoint operator is a μ fixpoint operator then Abelard wins and if it is a ν fixpoint operator then Eloïse wins. This corresponds to the fact that μ is for finite looping.

For example, consider the formula $\mu Z.p \vee (q \wedge \langle a \rangle Z)$ which is similar to a previously discussed example and is used to describe the property "There exists an a-labelled path where q holds until p holds".

Consider the following labelled transition system:



Let $\rho(q) = \{0\}$ and $\rho(p) = \{1\}$. Clearly the property holds for the state 0 since q holds at 0 and there is an a-transition to the state 1 where p holds. The corresponding game graph is shown in figure 1.

For this graph $\Omega(v) = 1$ for all $v \in V$ since the longest alternating chain is of length 1 and this uses a μ fixpoint formula.

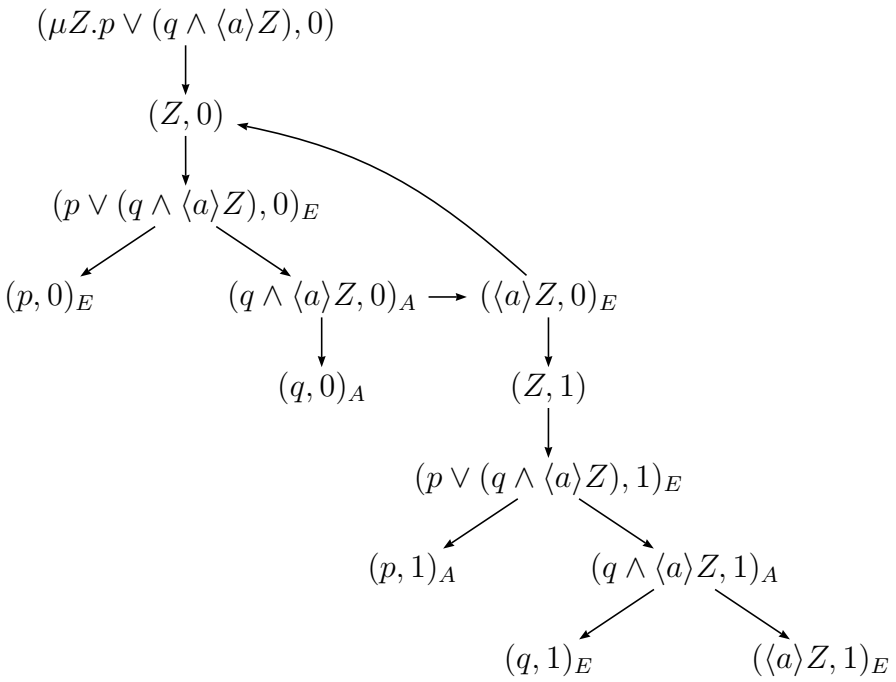
As would be expected Eloïse can win the game by employing a strategy where $\sigma((p \vee (q \wedge \langle a \rangle Z), 0)) = (q \wedge \langle a \rangle Z, 0)$, $\sigma((\langle a \rangle Z, 0)) = (Z, 1)$ and $\sigma((p \vee (q \wedge \langle a \rangle Z), 1)) = (p, 1)$.

For the formula $\mu Z.p \vee (q \wedge [a]Z)$, the graph would have the same structure as the previous example but Abelard would get to pick the next vertex at $([a]Z, 0)$ which would replace $(\langle a \rangle Z, 0)$. Abelard could therefore employ a strategy where $\pi(([a]Z, 0)) = (Z, 0)$, resulting in a infinite play that Abelard would win. This corresponds to the fact that the property $\mu Z.p \vee (q \wedge [a]Z)$ does not hold on the given transition system since by traversing the a-transition from 0 to itself, the property p never holds.

Note that using this construction, Büchi games can be used to solve the model checking problem for any modal μ-calculus formula contained in the class Π_2^μ and similarly, co-Büchi games can be used to solve the model checking problem for any formula contained in the class Σ_2^μ.

Also, given that co-Büchi games can be easily solved using Büchi games, Büchi games can be used to solve all modal μ-calculus formulas contained in $\Pi_2^\mu \cup \Sigma_2^\mu$. This is notable since the logic CTL* embeds into $\Pi_2^\mu \cup \Sigma_2^\mu$ [3].

Figure 1: Example parity Game



3 Algorithms for Solving Büchi Games

In this section we present the algorithms from [2] that are to be implemented along with an explanation of their correctness. It is important to note that the algorithms assume that all vertices in the game graph have at least one outgoing edge so that play never terminates. Since it is trivial to convert all Büchi games to this form, these algorithms can be used to solve all Büchi games.

We start by defining closed sets and attractors as given in [2], since these notions play an integral role in the definition and analysis of the algorithms.

Definition 3.1 *Given a game graph $G = (V, E, \lambda)$, an Eloïse* **closed set** *is a set $U \subseteq V$ of vertices such that the following two properties hold:*
1) For all $u \in U$, if $\lambda(u) = \exists$ then $E(u) \subseteq U$.
2) For all $u \in U$, if $\lambda(u) = \forall$ then $E(u) \cap U \neq \emptyset$.
Thus a set $U \subseteq V$ is closed for Eloïse if there exists a strategy for Abelard such that whenever a game starts from some $u \in U$, the play remains within U. (The definition for an Abelard closed set is analogous to this).

Definition 3.2 *Given a game graph $G = (V, E, \lambda)$ and a set $U \subseteq V$, the Eloïse* **attractor set** *for U (denoted $Attr_E(U, G)$) is the set of all states from which Eloïse can employ a strategy such that, regardless of Abelard's strategy, the play will always eventually reach an element of U. This can be defined formally using the following inductive definition:*

$Attr_E(U, G) := \bigcup_{i \geq 0} S_i$ *where:*

- $S_0 := U$
- $S_i := S_{i-1} \cup \{v \in V \mid \lambda(v) = \exists \text{ and } E(v) \cap S_{i-1} \neq \emptyset\}$
$\cup \{v \in V \mid \lambda(v) = \forall \text{ and } E(v) \subseteq S_{i-1}\}$

($Attr_A(U, G)$ can be defined analogously.)

Thus, an attractor set for some set $U \subseteq V$ can be computed by performing a backward search from all the vertices in U and thus can be computed in $O(m)$ time where $m = |E|$.

Notice that $V \backslash Attr_E(U, G)$ is an Eloïse closed set (and similarly $V \backslash Attr_A(U, G)$ is Abelard closed).

3.1 Algorithm 1

We begin by presenting the first algorithm which is usually regarded as the classical method for solving Büchi games:

Informal description of algorithm 1. The algorithm proceeds as follows: Given the Büchi game $\mathcal{G} = (G, v_0, F)$, consider the underlying game graph $G = (V, E, \lambda)$. Begin by computing $R_0 := Attr_E(F, G)$. This corresponds to the set of vertices from which Eloïse has a strategy to reach an accepting vertex at least once.

Let $T_0 := V \backslash R_0$. Clearly this set of vertices is winning for Abelard. We then compute the set W_0 of all vertices from which Abelard has a strategy to reach T_0, namely $W_0 := Attr_A(T_0, G)$. Clearly W_0 is also winning for Abelard and so the winning set of vertices for Eloïse must be contained in the reduced game graph $G_1 = (V_1, E_1, \lambda) := G \backslash W_0$.

Note that $V_1 := V \backslash Attr_A(T_0, G)$ and so is Abelard closed in G. Also, a winning play for Eloïse must remain in this set and thus when considering the set of winning vertices for Eloïse, we need only consider the game on the reduced graph G_1.

We continue by repeating the process for the reduced game graph G_1, computing $R_1 := Attr_E(F_1, G_1)$ (where $F_1 := F \cap V_1$), $T_1 := V_1 \backslash R_1$ and $W_1 := Attr_A(T_1, G_1)$ and then we reduce the game graph again to $G_2 := G_1 \backslash W_1$. We keep repeating this process until $T_i = \emptyset$ for some $i \in \mathbb{N}$ in which case the set of vertices in the remaining game is the set of winning vertices for Eloïse.

Thus, by checking if v_0 is in this set we can ascertain whether Eloïse has a winning strategy in the original game or not. Since each attractor computation takes $O(m)$ time, each iteration of the algorithm takes $O(m)$ time and thus since the algorithm does not run for more than $O(n)$ iterations, where $n = |V|$, the worst case running time for this algorithm is $O(mn)$.

The algorithm can be described more formally as follows:

Figure 2: Algorithm 1

Input: $\mathcal{G} := (G, v_0, F)$ where $G = (V, E, \lambda)$
Output: True or False
1. $G_0 = (V_0, E_0, \lambda) := (V, E, \lambda) = G$, $F_0 := F$, $i := 0$
2. $R_i := Attr_E(F_i, G_i)$
3. $T_i := V_i \backslash R_i$
4. If $T_i = \emptyset$ go to step 8 else continue to step 5.
5. $W_i := Attr_A(T_i, G_i)$
6. $G_{i+1} := G_i \backslash W_i$, $F_{i+1} := V_{i+1} \cap F_i$, $i := i + 1$
7. Go to step 2
8. **return** True if $v_0 \in V_i$ else False

$Attr_E(U, G)$ can be computed using the following algorithm:

Figure 3: Eloïse attractor algorithm

Input: $G = (V, E, \lambda)$, $U \subseteq V$
Output: $K \subseteq V$
1. $P(v) := |E(v)|$ for each $v \in V$
2. $I(v) := \{z \mid v \in E(z)\}$
3. $K_0 := U$, $i := 0$
4. $K_{i+1} := \emptyset$
5. **For each** $k \in K_i$:
   5.1 **For each** $v \in I(k)$:
      5.1.1 If $\lambda(v) = \exists$ or $P(v) = 1$, $K_{i+1} := K_{i+1} \cup \{v\}$ else $P(v) := P(v) - 1$
6. i:= i+1
7. If $(K_i \cup \bigcup_{j=0}^{i-1} K_j) = \bigcup_{j=0}^{i-1} K_j$ continue to step 8 else go to step 4
8. **return** $K = \bigcup_{j=0}^{i} K_j$

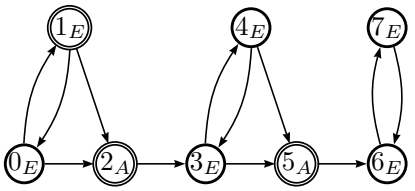
($Attr_A(U, G)$ can be similarly computed)

Note that both step 1 and 2 of the attractor algorithm can be computed by considering each edge in the graph in turn and hence both steps have a running time of $O(m)$. Similarly, step 5 iterates over elements of $\{I(z) \mid z \in V\}$ and at each iteration, all elements of $I(z)$ are considered. Since $\sum_{z \in V} |I(z)| = m$, the total running time for this step is $O(m)$. The worst case running time for the computation of an attractor set is therefore $O(m)$.

Thus in algorithm 1, at each iteration, step 2 has a worst case running time of $O(m)$. Also, since R_i can be at worst of size $O(n)$, and similarly for V_i, step 3 has a worst case running time of $O(n)$. Over all iterations of algorithm 1, step 5 has a total running time of $O(m)$, since any edges considered in the attractor calculation are removed from the game graph in step 6, and thus these edges

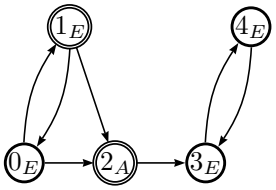
are not considered again in step 5 in a later iteration. Due to step 2 and since $n \leq m$, each iteration of the algorithm does at most $O(m)$ work. Since the algorithm iterates at most $O(n)$ times, the worst case time complexity for this algorithm is therefore $O(mn)$.

The algorithm is demonstrated using the following example:



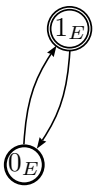
1. $G_0 = (V_0, E_0, \lambda)$ where $V_0 = \{0_E, 1_E, 2_A, 3_E, 4_E, 5_A, 6_E, 7_E\}$,
   $E_0 = \{(0_E, 1_E), (0_E, 2_A), (1_E, 0_E), (1_E, 2_A), (2_A, 3_E), (3_E, 4_E), (3_E, 5_A), (4_E, 3_E),$
   $(4_E, 5_A), (5_A, 6_E), (6_E, 7_E), (7_E, 6_E)\}$,
   $\lambda(v) = \exists$ if $v \in \{0_E, 1_E, 3_E, 4_E, 6_E, 7_E\}$ else $\lambda(v) = \forall$,
   $F_0 = \{1_E, 2_A, 5_A\}$ and i=0
2. $R_0 = Attr_E(F_0, G_0) = F_0 \cup \{0_E, 3_E, 4_E\}$
3. $T_0 = V_0 \backslash R_0 = \{6_E, 7_E\}$
4. $T_0 \neq \emptyset$ go to step 5
5. $W_0 = Attr_A(T_0, G_0) = T_0 \cup \{5_A\}$
6. $G_1 = G_0 \backslash W_0$, $F_1 = \{1_E, 2_A\}$, i=1.

Hence G_1 is the following game graph:



7. Go to step 2
2. $R_1 = F_1 \cup \{0_E\}$
3. $T_1 = \{3_E, 4_E\}$
4. $T_1 \neq \emptyset$. Go to step 5
5. $W_1 = T_1 \cup \{2_A\}$
6. $G_2 = G_1 \backslash W_1$, $F_2 = \{1_E\}$, i=2

Hence G_2 is the following game graph:



7. Go to step 2
2. $R_2 = F_2 \cup \{0_E\}$

3. $T_2 = \emptyset$
4. Go to step 8
8. If $v_0 \in \{0_E, 1_E\}$ return True else False

The correctness of the algorithm can be verified using the following results:

Proposition 3.1 *For $W_i \neq \emptyset$, the set W_i is winning for Abelard where $i \in \mathbb{N}$.*

Proof (By induction) As explained in the description of the algorithm and by the definition of attractors, W_0 is winning for Abelard.

Now assume that the result holds for W_i for all $0 \leq i < k$ for some $k \in \mathbb{N}$. Thus consider T_k. This corresponds to the set of vertices from which Eloïse does not have a strategy to reach an accepting vertex in the remaining game graph. Thus, in the original game the only accepting states that Eloïse can possibly have a strategy to reach, beginning from an element of T_k, are those that have already been removed (i.e. are contained in W_i for some $0 \leq i < k$) and thus are winning for Abelard by the inductive hypothesis. Hence all the elements of T_k are winning for Abelard so $W_k := Attr_A(T_k, G_k)$ is also winning for Abelard.

Proposition 3.2 *If $T_i = \emptyset$ then V_i is the set of winning vertices for Eloïse.*

Proof $T_i = \emptyset \implies R_i = V_i$. Hence, since $R_i := Attr_E(F_i, G_i)$, from any $v \in V_i$, Eloïse has a strategy to reach an accepting vertex, and the corresponding play is contained in the reduced game graph G_i. Thus any infinite play employing this strategy must reach an accepting vertex in F_i infinitely often and hence the strategy is winning for Eloïse since the game graph is such that all plays are infinite. By Proposition 3.1, any vertex that has previously been removed from the game graph is winning for Abelard and hence V_i is the set of winning vertices for Eloïse in $\mathcal{G}$.

The following result can be used to confirm the complexity of the algorithm:

Proposition 3.3 *If $T_{i+1} \neq \emptyset$ then $F_i \cap W_i \neq \emptyset$.*

Proof Since R_i is the Eloïse attractor for F_i and W_i is the Abelard attractor for $V \backslash R_i$, if $F_i \cap W_i = \emptyset$ then $R_i \cap W_i = \emptyset$. If $R_i \cap W_i = \emptyset$ then $V_{i+1} = R_i$ and hence $R_{i+1} = R_i$ which implies that $T_{i+1} = \emptyset$.

From this proposition we can conclude that the algorithm iterates at most $O(b)$ times where $b = |F|$. Hence if $b = O(n)$, since each iteration has $O(m)$ running time, the worst case time complexity for this algorithm is $O(mn)$.

3.2 Algorithm 2

We now present an algorithm given in [2]. Although this algorithm has the same worst case time complexity as the first algorithm, it improves on algorithm 1 in certain examples and is at worst $O(m)$ slower. A notable improvement is demonstrated for a particular family of graphs. In this case the second algorithm has a linear $O(n)$ time complexity whereas the first has a quadratic $O(n^2)$ time complexity.

Informal description of algorithm 2. The algorithm proceeds as follows: Given the Büchi game $\mathcal{G} = (G, v_0, F)$, consider the underlying game graph $G = (V, E, \lambda)$. The algorithm begins by considering the set C_1^0 of non-accepting vertices that are owned by Eloïse and have all outgoing edges going to non-accepting vertices and the set C_2^0 of non-accepting vertices that are owned by Abelard and have an outgoing edge to a non-accepting vertex.

The union of these two sets is the set of candidate vertices to be included in the set T_0 from algorithm 1. More precisely $T_0 \subseteq (C_1^0 \cup C_2^0)$. The Abelard attractor set X_0 for the union of these two sets is found ($X_0 := Attr_A(C_1^0 \cup C_2^0, G)$). This corresponds to the set of all vertices from which Abelard can force the play to reach one of the candidates for the set T_0.

The algorithm then proceeds by finding the intersection of X_0 with the set of non-accepting vertices (call this set Z_0). The set D_0 is then calculated which is the union of the set $X_0 \backslash Z_0$ and the set of all vertices in Z_0 from which Eloïse can force the game to leave Z_0 in the next move. Clearly, the set D_0 is disjoint from T_0.

The Eloïse attractor set of D_0 restricted to the vertices in X_0 is then computed (call it L_0). Note that although X_0 is not a closed set, since all vertices in $X_0 \backslash D_0$ have at least one outgoing edge contained in $G \backslash (V \backslash X_0)$, it is valid to compute the Eloïse attractor of D_0 on $G \backslash (V \backslash X_0)$. L_0 therefore corresponds to the set of all vertices in X_0 from which Eloïse has a strategy to force the play to either reach an accepting vertex in X_0 or to leave X_0 altogether.

By definition, from the set of vertices in $V \backslash X_0$, Eloïse has a strategy to reach an accepting vertex. Thus $X_0 \backslash L_0$ (and analogously $Z_0 \backslash L_0$) corresponds exactly to those vertices in V from which Eloïse cannot force the play to reach an accepting vertex in G, i.e. $T_0 = X_0 \backslash L_0$ (or $Z_0 \backslash L_0$) and thus the algorithm can proceed as in algorithm 1. $W_0 := Attr_A(T_0, G)$ and $G_1 := G \backslash W_0$ and then the algorithm is repeated again on the reduced game graph G_1 as before and the algorithm continues until $T_i = \emptyset$ for some $i \in \mathbb{N}$. As is given in [2], algorithm 2 can be described more formally as shown in figure 4.

Figure 4: Algorithm 2

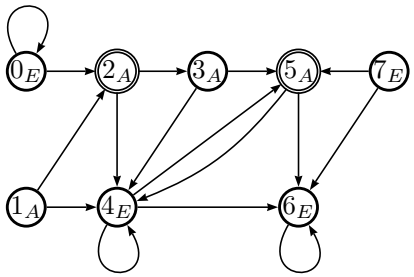
Input: $\mathcal{G} := (G, v_0, F)$ where $G = (V, E, \lambda)$
Output: True or False
1. $G_0 = (V_0, E_0, \lambda) := (V, E, \lambda) = G$, $i := 0$
2. $C^0 := V_0 \backslash F$
3. $C_1^i := \{v \in V_i \mid \lambda(v) = \exists \text{ and } E(s) \cap V_i \subseteq C^i\}$
4. $C_2^i := \{v \in V_i \mid \lambda(v) = \forall \text{ and } E(s) \cap C^i \neq \emptyset\}$
5. $X_i := Attr_A(C_1^i \cup C_2^i, G_i)$
6. $Z_i := X_i \cap C^i$
7. $D_i := \{v \in Z_i \mid \lambda(v) = \exists \text{ and } E(s) \cap (V_i \backslash Z_i) \neq \emptyset\} \cup \{v \in Z_i \mid \lambda(v) = \forall \text{ and } E(s) \cap V_i \subseteq (V_i \backslash Z_i)\} \cup (X_i \backslash Z_i)$
8. $L_i := Attr_E(D_i, G_i \backslash (V_i \backslash X_i))$
9. $T_i := Z_i \backslash L_i$
10. If $T_i = \emptyset$ go to step 14 else continue to step 11.
11. $W_i := Attr_A(T_i, G_i)$
12. $G_{i+1} := G_i \backslash W_i$, $C^{i+1} := V_{i+1} \cap C^i$, $i := i + 1$
13. Go to step 3
14. **return** True if $v_0 \in V_i$ else False

Since the attractor computations have a worst case time complexity of $O(m)$, in each iteration steps 5 and 8 have at worst a time complexity of $O(m)$.

In step 7, the algorithm works on the edges of the vertices in Z_i. The work done at each iteration in step 7 is therefore also at worst $O(m)$. Similarly, the work done for steps 3 and 4 at each iteration is at worst $O(m)$.

Also, it is clear that the work done at each iteration for steps 6 and 9 are no worse than $O(n)$ and hence since there could be at worst $O(n)$ iterations, similarly to algorithm 1, algorithm 2 has a worst case time complexity of $O(mn)$.

The following example demonstrates algorithm 2:



1. $G_0 = (V_0, E_0, \lambda)$ where $V_0 = \{0_E, 1_A, 2_A, 3_A, 4_E, 5_A, 6_E, 7_E\}$,
   $E_0 = \{(0_E, 0_E), (0_E, 2_A), (1_A, 2_A), (1_A, 4_E), (2_A, 3_A), (2_A, 4_E), (3_A, 4_E), (3_A, 5_A), (4_E, 4_E), (4_E, 5_A), (4_E, 6_E), (5_A, 4_E), (5_A, 6_E), (6_E, 6_E), (7_E, 5_E), (7_E, 6_E)\}$,
   $\lambda(v) = \exists$ if $v \in \{0_E, 4_E, 6_E, 7_E\}$ else $\lambda(v) = \forall$,

$F_0 = \{2_A, 5_A\}$ and i=0
2. $C^0 = \{0_E, 1_A, 3_A, 4_E, 6_E, 7_E\}$
3. $C_1^0 = \{6_E\}$
4. $C_2^0 = \{1_A, 3_A\}$
5. $X_0 := Attr_A(C_1^0 \cup C_2^0, G_0) = (C_1^0 \cup C_2^0) \cup \{2_A, 5_A, 7_E\}$
6. $Z_0 := X_0 \cap C^0 = \{1_A, 3_A, 6_E, 7_E\}$
7. $D_0 = \{7_E\} \cup \{1_A\} \cup \{2_A, 5_A\}$
8. $L_0 := Attr_E(D_0, G_0 \backslash \{0_E, 4_E\}) = \{1_A, 2_A, 3_A, 5_A, 7_E\}$
9. $T_0 := Z_0 \backslash L_0 = \{6_E\}$
10. $T_0 \neq \emptyset$. Continue to step 11.
11. $W_0 := Attr_A(T_i, G_i) = \{1_A, 2_A, 3_A, 5_A, 6_E, 7_E\}$
12. $G_1 = G_0 \backslash W_0, C^1 = \{0_E, 4_E\}$, $i = 1$

Hence G_1 is the following game graph:

13. Go to step 3
3. $C_1^0 = \{0_E, 4_E\}$
4. $C_2^0 = \emptyset$
5. $X_1 = \{0_E, 4_E\}$
6. $Z_1 = \{0_E, 4_E\}$
7. $D_1 = \emptyset$
8. $L_1 = \emptyset$
9. $T_1 = \{0_E, 4_E\}$
10. $T_1 \neq \emptyset$. Continue to step 11.
11. $W_1 = \{0_E, 4_E\}$
12. $G_2 = G_1 \backslash W_1 = (\emptyset, \emptyset, \lambda), C^2 = \emptyset$, $i = 2$

Clearly on the next iteration the algorithm will terminate and since $V_2 = \emptyset$, the algorithm will return false for any vertex in the graph.

A proof of correctness of algorithm 2 is included in [2], though a informal explanation for the correctness of the algorithm is given as follows:

Whereas algorithm 1 finds all the vertices from which Eloïse can force the play to reach an accepting vertex and then takes the complement of this to obtain T_i, algorithm 2 focuses on finding T_i in a more direct manner by examining a well chosen superset of T_i, namely $C_1^i \cup C_2^i$. In order to decide which vertices in $C_1^i \cup C_2^i$ are not in T_i, it is necessary to expand this superset to the Abelard

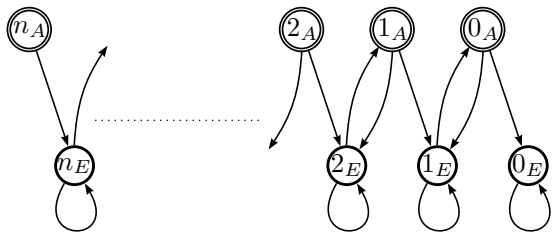
attractor set X_i of $C_1^i \cup C_2^i$. The algorithm then continues by computing the vertices in X_i which are not included in T_i.

Z_i is calculated as the intersection between X_i and C^i since if a vertex is not in C^i then it is an accepting vertex and thus cannot be a member of T_i. Using Z_i, the set D_i is calculated which corresponds to the elements of X_i which are either not in Z_i or are vertices from which Eloïse can force the play to leave Z_i in one move, preventing such a vertex from being in T_i. The Eloïse attractor (L_i) of D_i on the graph restricted to the vertices in X_i is then calculated so as to determine from which vertices in X_i Eloïse can force the play to leave the subset Z_i thus identifying such vertices as not being in T_i.

By removing L_i from X_i, the remaining vertices are Eloïse closed, and thus since $X_i \backslash L_i$ contains no accepting vertices, this set must be included in T_i. Since L_i and T_i are disjoint and $T_i \subseteq X_i$, we therefore have that $T_i = X_i \backslash L_i$. From the proof of correctness of algorithm 1, the correctness of algorithm 2 now follows.

Algorithm 2 is preferable to algorithm 1 when the superset X_i is relatively small, thus implying that the set R_i in algorithm 1 is relatively large. If R_i is relatively small, however, then T_i is large, implying that X_i is large thus making algorithm 1 preferable to algorithm 2. It is worth noting, however, that as proved in [2], algorithm 2 performs at most $O(m)$ more work than algorithm 1 whereas there exist examples of Büchi games where algorithm 1 runs in quadratic time whereas algorithm 2 runs in linear time.

As given in [2], consider the following example:



Clearly the graph has $O(n)$ vertices and $O(n)$ edges, and at each iteration of both algorithms $T_i = \{i_E\}$ and $W_i = \{i_E, i_A\}$ hence at each iteration $V_i = \{i_E, i_A, (i+1)_E, (i+1)_A, (i+2)_E, (i+2)_A, ..., n_E, n_A\}$.

Using algorithm 2, T_i is obtained as follows:

- $C_1^i = \{i_E\}$ and $C_2^i = \emptyset$
- $X_i = \{i_E, i_A\}$
- $Z_i = \{i_E\}$
- $D_i = \{i_A\}$
- $L_i = \{i_A\}$

- $T_i = Z_i \backslash L_i = \{i_E\}$

In order to calculate X_i, algorithm 2 works on the incoming edges of i_E and i_A thus this step is completed in constant time. The computation of D_i and L_i is restricted to the edges of the vertices in X_i thus these sets are also obtained in constant time. Clearly Z_i can be decided in constant time and hence at each iteration, T_i can be decided in constant time.

Since $W_i = \{i_E, i_A\}$, it is decided by working on the edges in $\{i_E, i_A\}$ and hence each iteration of algorithm 2 of this graph is completed in constant time.

Since the algorithm iterates $O(n)$ times before terminating, the total work for algorithm 2 is $O(n)$. When using algorithm 1, however, T_i is obtained as follows:

- $R_i = \{i_A, (i+1)_E, (i+1)_A, (i+2)_E, (i+2)_A, ..., n_E, n_A\}$
- $T_i = V_i \backslash R_i = \{i_E\}$

In order to calculate R_i, algorithm 1 must work on the incoming edges of the vertices in R_i, thus the total work for each iteration is at least $O(n-i)$. Since the algorithm continues for $O(n)$ iterations, the total work for algorithm 1 on this graph is at least $\Sigma_{i=1}^{n-1}(n-i) = O(n^2)$.

3.3 Algorithm 3

As has been established in the previous section, when R_i is small it is faster to use algorithm 1, whereas when X_i is small it is faster to use algorithm 2. It is not necessarily true, however, that given a Büchi game, over all the iterations of the algorithms, either R_i or X_i is consistently small. It may be that for some iterations R_i is relatively small whereas for others X_i is. Hence, for some graphs, it would be an preferable to use algorithm 1 for some iterations and algorithm 2 for others.

In order to address this, since it is too expensive to determine at the start of each iteration which algorithm would be faster, the following is suggested in [2]: Use an algorithm that dovetails algorithm 1 and 2 at the start of each iteration until T_i is obtained by one of the algorithms and then proceed by calculating W_i as before.

In the cases where it varies between iterations whether algorithm 1 or 2 is preferable, this third algorithm may outperform both of the previous algorithms. An added advantage is that even with a Büchi game where one algorithm is consistently better than the other, although algorithm 3 will perform worse than the preferred algorithm, it will likely perform better (possibly significantly so) than the other algorithm.

Algorithm 3 will perform less well however, in the case of Büchi games where at each iteration both algorithms 1 and 2 are comparable in speed, thus the advantage of dovetailing would be lost.

The performance of Algorithm 3 along with Algorithm 2 and 1 will be tested in Section 6.

3.4 Generating Winning Strategies

In Section 6, the performance of the algorithms will be compared to several parity game solvers from [6]. In [6], however, a solution is given in the form of Abelard and Eloïse winning regions and a corresponding winning strategy for each player. In order to make a fair comparison to these parity game solvers, therefore, it is necessary modify the algorithms so that they also produce this type of solution. All three of the algorithms solve Büchi games in a global manner and thus Abelard and Eloïse winning regions can easily be produced. More precisely the Abelard winning region for a Büchi game will be $\bigcup_{i\geq 0} W_i$ and the Eloïse winning region is the complement of such a set. Hence, in order to allow for a fair comparison against the parity game solvers in [6], it remains to consider a method of generating a winning strategy.

Given a game graph $G = (V, E, \lambda)$ and an accepting set $F \in V$, we wish to compute the strategies $\sigma : \{v \in V \mid \lambda(v) = \exists\} \to V$ and $\pi : \{v \in V \mid \lambda(v) = \forall\} \to V$ such that for all $v \in V$, if v is winning for Eloïse then the play $\omega_v(\sigma, \pi)$, starting from v corresponding to those strategies results in a win for Eloïse. Similarly, if v is winning for Abelard then $\omega_v(\sigma, \pi)$ results in a win for Abelard.

We begin with the following definition:

Definition 3.3 *Given an attractor set $Attr_\theta(U, G)$, an **attractor strategy** is a strategy for player $\theta \in \{\exists, \forall\}$, for vertices $v \in \{u \in V \mid \lambda(u) = \theta\} \cap Attr_\theta(U, G)$, such that from any $v \in Attr_\theta(U, G)$, a play corresponding to that strategy always reaches a vertex in the set U.*

This can be computed as follows:

Given $v \in \{u \in V \mid \lambda(u) = \theta\} \cap Attr_\theta(U, G)$, let $i \in \mathbb{N}$ be the smallest i such that $v \in S_i$ where S_i is as given in the definition of attractor sets. Then for an attractor strategy σ_θ, $\sigma_\theta(v) \in \{w \in E(v) \mid w \in S_{i-1}\}$.

An attractor strategy can thus be computed at the same time as the corresponding attractor set is decided. While searching backwards through the graph, when a vertex v is added to the attractor set, $\sigma(v)$ is assigned to the successor vertex whose inspection caused v to be added.

At each iteration i of algorithms 1 to 3, the set T_i is calculated which corresponds to an Eloïse closed set on which, from all vertices in T_i, Abelard has a winning strategy. Thus, there are no accepting vertices in T_i and hence an Abelard winning strategy for $v \in T_i \cap \{u \in V \mid \lambda(u) = \forall\}$ can be calculated simply by setting $\pi(v)$ equal to any successor vertex that is a member of T_i. Note that since T_i is Eloïse closed and is winning for Abelard, it is irrelevant what $\sigma(v)$ is set to for $v \in T_i$ such that $\lambda(v) = \exists$. Thus for such v, $\sigma(v)$ can be set to any successor vertex.

From T_i, the attractor set $W_i := Attr_A(T_i, G_i)$ is then calculated, which is also a set of vertices that are winning for Abelard. The correct strategy for Abelard on the vertices in $W_i \backslash T_i$ can be computed simply by computing the attractor strategy during the computation of W_i. This is because employing this strategy results in Abelard forcing the play to reach an element of T_i from which a winning strategy for Abelard has already been decided. Again, for $v \in W_i \backslash T_i$ such that $\lambda(v) = \exists$, $\sigma(v)$ can be set equal to any successor vertex.

Since all Abelard winning vertices are contained in one of W_i during algorithms 1-3, it remains to calculate the correct strategy for the Eloïse winning vertices. In the final iteration (say $l \in \mathbb{N}$) of each of the algorithms, exactly the Eloïse winning vertices remain. Thus from each $v \in V_l$, Eloïse can force the play to reach one of $f \in F \cap V_l$. The set is Abelard closed and hence for any $v \in V_l$ such that $\lambda(v) = \forall$, $\pi(v)$ can be set to any successor vertex. Also, since only Eloïse winning vertices remain, for each $f \in F \cap V_l$ such that $\lambda(f) = \exists$, $\sigma(v)$ can be set to any successor vertex in V_l. It remains to consider the strategy for vertices in $v \in V_l \backslash F$ such that $\sigma(v) = \exists$. These have to be calculated by computing the attractor strategy for the attractor set $Attr_E(F \cap V_l, G_l)$ which is equivalent to the set V_l.

Note that in algorithm 1, the set $Attr_E(F_i, G_i)$ is calculated at each iteration and so the attractor strategy for this Eloïse attractor set can be calculated simultaneously. However, since algorithm 2 only works on the edges of vertices in X_i for each iteration, an extra attractor computation will have to be made at the end of the algorithm in order to identify the correct strategy for Eloïse. This could significantly affect its performance in comparison to algorithm 1. For the same reasons, an extra attractor computation will also have to be made at the end of algorithm 3.

Although computing winning strategies affects the performance of algorithm 2 to a larger degree than that of algorithm 1, it could still be preferable to use algorithm 2 (or 3). In particular, when there are a large number of iterations, the effect of the extra attractor computation in the final iteration would be reduced. Note also that except for Section 6.2 when a comparison to parity game solvers is made, winning strategies will not be computed.

4 Implementation

This section describes the implementation in OCaml of the Büchi game solving algorithms from Section 3.

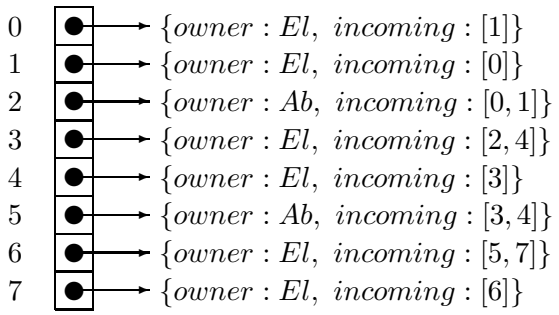
4.1 Büchi Game Structure

In order to implement these algorithms while maintaining a time complexity of $O(mn)$, it is important to choose appropriate data structures.

To aid the implementation of these algorithms, it is assumed that the graph is given in the form of an array where each element of the array is a record that corresponds to a vertex. Each vertex is represented by an integer and each record is used to store a list of the predecessor vertices for that vertex and a data type that determines whether the vertex is owned by Eloïse or Abelard. It would be sensible to receive the graph in this form since the attractor calculation in the algorithm requires that we search backwards through the graph and thus we will need to iterate through the set of incoming edges of vertices as they are added to the attractor set.

Since the predecessor vertices for each vertex are stored in a list, when a vertex is removed from the graph during the algorithm, it will not be removed from any list of predecessor vertices that it appears in since this would be too expensive. Instead, when computing each attractor, while checking through the incoming edges of a vertex, a check will be made for each corresponding predecessor vertex to see if it has been removed from the game graph before considering adding it to the current attractor set. This will not affect the overall complexity of the algorithms since each attractor computation will still take at worst $O(m)$ time provided that the check whether a vertex has been removed can be made in constant time.

In the example given in Section 3 to demonstrate algorithm 1 therefore, the graph would be received in the following form:



A game will therefore be given as a record that stores a graph in the form given above and a list of accepting vertices.

4.2 Data Structures for Algorithm 1

As mentioned above, as the algorithm progresses, vertices are removed from the graph, so we will need a data structure that stores which vertices are included in the current iteration. This would initially correspond to the set V_0 and then be modified to correspond to V_i at each iteration i by removing appropriate vertices. For each iteration, in step 6 of algorithm 1 in figure 2, we could be removing $O(n)$ vertices, thus the vertex set will need to be stored in a data structure such that each removal can be performed in constant time.

Also, when computing an attractor set, we will need to check whether vertices considered for addition to this set are contained in the current vertex set by referring to this data structure. Again, since we will need to examine at worst $O(n)$ vertices for each attractor calculation, we will need to be able to ascertain whether a vertex is in the current vertex set in constant time. In addition to this, V_i is used to compute $T_i \backslash R_i$ in algorithm 1. This computation of T_i should be completed in no worse than $O(n)$ time. The structure of V_i therefore depends on the structure required for T_i and vice-versa.

For V_i, an array of length n will be used, that stores Boolean values corresponding to whether a vertex is in the current vertex set. Thus, the membership of a vertex in the current vertex set can be ascertained in constant time, and can also be altered in constant time using this data structure.

Initially, this will be an array of length n where each entry is set to true and when a vertex is removed the corresponding boolean value in the array can be set to false.

Since T_i is used as the input of an attractor calculation, we will need to iterate through all the elements of T_i in order to check their predecessor vertices. In order to achieve this efficiently, it would be sensible to store T_i in a list. This means that in order to compute T_i we will need to iterate through the elements of V_i, checking their membership in the set R_i and adding them to T_i accordingly. Iterating through elements of V_i using the Boolean array data structure would be inefficient since it would involve iterating over all elements of the length n array. Thus computing T_i would always take $O(n)$ time regardless of how many vertices were left in the graph. For this reason, a second data structure will be used for V_i that allows the iteration of elements in V_i to take time proportional to the cardinality of the set V_i. In order to update the vertex set at each iteration, it will also be necessary to remove vertices from the data structure in constant time.

Thus the following data structure will be used. An integer array of length $n+1$ will be used to store the vertices (the vertex array) and an integer array of length n will be used to store the location of each vertex in the vertex array (the location array). The last element in the vertex array will correspond to the size of the current vertex set (some $0 \leq k \leq n$) and the first k elements in the vertex array will correspond to the vertices in the vertex set (not necessarily in

numerical order). To facilitate the removal of elements from the vertex array, the position of each vertex will be stored in the location array. Thus, a vertex can be removed by checking the position of the vertex and replacing it with the last vertex of the set stored in the vertex array. The cardinality of the set would then be reduced by one (by reducing the value of the last element in the vertex array). The following example demonstrates this:

The vertex set $\{2, 1, 5, 4\}$ out of the possible set $\{0, 1, 2, 3, 4, 5, 6\}$ of 7 vertices could be represented by the following arrays:

V

0	1	2	3	4	5	6	7
2	1	5	4	8	8	8	4

$Vloc$

0	1	2	3	4	5	6
7	1	0	7	3	2	7

The first four elements of the array V correspond to the desired set. The last element of V corresponds the cardinality of the set and the remaining elements of V, which are not being used, are by default set equal to the length of the array. In $Vloc$, the entries corresponding to vertices not included in the set are by default set to the length of $Vloc$. The other entries correspond to the position of each vertex in V.

If we wanted to remove vertex 1 from this set, V and $Vloc$ would be altered to the following:

V

0	1	2	3	4	5	6	7
2	4	5	4	8	8	8	3

$Vloc$

0	1	2	3	4	5	6
7	1	0	7	1	2	7

Vertex 1 in V is replaced with the last element in the set, namely vertex 4, and the location for 4 in $Vloc$ is altered accordingly. Also the cardinality of the set is reduced by one.

If instead, we wished to remove vertex 4 from the original set then V and $Vloc$ would be altered to the following form:

V

0	1	2	3	4	5	6	7
2	1	5	4	8	8	8	3

$Vloc$

0	1	2	3	4	5	6
7	1	0	7	1	2	7

Vertex 4 occurs last in the set and so cannot be replaced in the same way as before. Instead the only alteration required is to reduce the cardinality by one.

Notice that the vertex location of those vertices removed is not altered. This is because no vertex is removed from any set more than once in the algorithm and the vertex location of a removed vertex is never referred to again.

Using this data structure, the elements of V_i can now be iterated through in time proportional to their size. This is achieved by iterating through the vertex array up to the index that is one less than the cardinality of the vertex set.

Similarly to T_i, F_i is also used as the input of an attractor calculation and thus a data structure allowing efficient iteration through its elements is required. Unlike T_i though, at the end of each iteration, the set F_{i+1} needs to be calculated by removing elements from F_i. Elements in F_i therefore need to be removed in constant time as was the case with V_i. In order to achieve both of these requirements, the same data structure as the second structure for V_i will be used.

Additionally, when considering algorithm 1, a data structure that stores R_i for each iteration is required. R_i is an attractor set and thus is computed by adding vertices inductively to a set. In the worst case, $O(n)$ vertices are added. For this reason, R_i needs to be stored in a data structure such that a vertex can be added to R_i in constant time. Also, we will need to check membership of vertices in R_i in order to compute $T_i := V_i \backslash R_i$. To execute this in the most efficient manner, it is therefore appropriate to store each R_i as an array of boolean values like the first structure used for V_i.

The most appropriate data structure for W_i is a list since it is necessary to iterate through the elements of W_i in order to remove them from the game graph.

For algorithm 1, it now remains to consider the data structures required for the attractor computations:

For each iteration of algorithm 1, two attractor sets are computed. Throughout an attractor computation, over all $v \in V$, the value of $P(v)$ (initially equal to $|E(v)|$) is considered at worst $O(m)$ times. Hence for all v, the value of $P(v)$ must be accessed in constant time in order to maintain the correct complexity. Thus this information will be stored in an array of length n where each entry in the array contains $P(v)$ for the vertex corresponding to that entry in the array. Because $P(v)$ is modified during each attractor computation, in order to prevent having to re-calculate $|E(v)|$ each time an attractor set is computed, this data will be managed in four arrays instead of one. One array will be used to store $|E(v)|$ for each v and will be initialised at the start of the main algorithm. It will then be fixed and not modified until the end of each iteration of the algorithm, when the value of $|E(v)|$ changes. The second and third arrays will be used to store which iteration of the main algorithm each vertex was last considered during the first and second attractor set computations respectively. The fourth array will store the current 'working value' of $P(v)$ for each v. That is, at each iteration, when a vertex is considered in the first attractor calculation, the corresponding entry for that vertex in the second array will be checked. For the second attractor calculation the third array will be checked. If the value in this array is not equal to the current iteration then the vertex

has not yet been considered in the current attractor set computation. In this case the value of $P(v)$ from the first array will be copied into the fourth and then this entry in the fourth array will be modified during the attractor calculation. If the value is equal to the current iteration number then the vertex has previously been considered in the current attractor computation and thus the algorithm will inspect and modify the value stored in the fourth array.

$I(v)$ can already be accessed in the inputted graph as a list, so no data structure needs to be considered for this.

In the algorithm, when computing the attractor set for a set U, the set K_0 (as defined in the attractor set algorithm) is assigned as equivalent to U. In the main algorithm, F_i and T_i are the inputs for the attractor computations at each iteration, where one is a list and the other is stored in an array as described above. For continuity and transparency of the attractor implementation however, the input of the attractor computation will correspond to a list. F_i will therefore be converted into a list before being inputted into the attractor calculation. Since all the elements of F_i will be considered during the attractor calculation anyway, this should not noticeably affect the efficiency of the implementation.

Also, since the output of the attractor computation is the union of all K_i for each iteration i, the output of the attractor computation will be a list. This matches the data structure required for W_i, however for R_i this will have to be translated into a boolean array. Again, this does not affect the efficiency of the implementation too much since it can be achieved in time proportional to the size of the outputted list.

4.3 Algorithm 1 Implementation

Using the data structures given, algorithm 1 is given in figure 5.

The function $Initialise$ calculates the number of successor vertices for each vertex and stores this count in the array P. It iterates through each list of predecessor vertices in the array I, and increments the value of $P[i]$ by one for each vertex i that appears in a list.

The function $InitialiseF$ iterates through the list of accepting vertices $Flist$, and adds them to the F, $Floc$ data structure as described in the previous section. For each $i \in Flist$, the current number of vertices in F is determined by referring to $F[n]$. The vertex i is then added to F at position $F[n]$ and the value of $F[n]$ is incremented by one. The position of i in F is then recorded in $Floc[i]$ to allow for removal of the vertex at a later stage. Note that in contrast to F and $Floc$, V and $Vloc$ are initialised to the correct value in step 2 since all vertices are contained in the initial vertex set.

Figure 5: Algorithm 1 Implementation

Input: Graph $G := (I[n], Owner[n])$, Integer list $Flist$, Integer v_0

Output: True or False

1. **Create arrays** $isvert[n]$, $V[n+1]$, $Vloc[n]$, $F[n+1]$, $Floc[n]$, $R[n]$, $P[n]$, $LookF[n]$, $LookT[n]$, $Work[n]$
2. **For** $0 \leq j \leq n-1$:
   $isvert[j] := true$, $V[j] := j$, $Vloc[j] := j$, $F[j] := n+1$,
   $Floc[j] := n$, $R[j] := false$, $LookF[j] := -1$, $LookT[j] := -1$,
   $P[j] := 0$, $Work[j] := 0$
4. $V[n] := n$, $F[n] := 0$
5. $Initialise(P, G)$
6. $InitialiseF(Flist, F, Floc)$
7. $MainAlgorithm$(0,G,v_0,$isvert$,V,$Vloc$,F,$Floc$,R,P,$LookF$,$LookT$,$Work$)

Once all the data structures have been initialised, the function $MainAlgorithm$ given in figure 6 implements the rest of the algorithm.

Figure 6: MainAlgorithm

Input: Integers: i, v_0, Graph: $G := (I[n], Owner[n])$,
Boolean Arrays: $isvert, R$,
Integer Arrays: $V, Vloc, F, Floc, P, LookF, LookT, Work$

Output: True or False

1. Create list $Flist$ of elements in array F
2. $Rlist := Attractorcomp(i, isvert, P, Work, Eloise, LookB, Flist, G)$
3. **For each** $j \in Rlist$: $R[j] := true$
4. $T := ComputeT(R, V)$
5. $W := Attractorcomp(i, isvert, P, Work, Abelard, LookT, T, G)$
6. If $W :=$ [ ] then **return** $true$ else continue to step 7
7. If $v_0 \in W$ then **return** $false$ else continue to step 8
8. **For each** $j \in W$:
   8.1. $isvert[j] := false$
   8.2. $Remove(j, V, Vloc)$
   8.3. $Remove(j.F, Floc)$
   8.4. **For each** $v \in I[j]$:
      8.4.1 $P[v] := P[v] - 1$
9. **For each** $j \in Rlist$: $R[j] := false$
10. MainAlgorithm(i+1,G,v_0,$isvert$,V,$Vloc$,F,$Floc$,R,P,$LookF$,$LookT$,$Work$)

The list $Flist$ of accepting vertices is computed by iterating through the array F up to the index $F[n]-1$. The $ComputeT(R,V)$ function finds the list T by iterating through the array V up to the index $V[n]-1$. This corresponds to the current vertex set. Each of these vertices is checked for membership in R and added to T accordingly.

Once the list W is computed, step 8 updates the data structures by removing vertices which are in W. The function *Remove* removes a vertex from the V, $Vloc$ data structure as described in the previous section. Also, if the vertex location of a vertex is set to n then the function *Remove* does not attempt to remove that vertex. Since, all values of $Floc$ are initialised to n in step 2 of figure 5, if a vertex j is not contained in the set F, the function will not attempt to remove it. In step 8.4, for each predecessor vertex of a vertex in W, the outgoing edge count stored in P is decremented by one. In step 9, the value of $R[j]$ is reset to false for the next iteration.

The function Attractorcomp computes attractor sets and is described in figure 7.

Figure 7: Attractorcomp(i,$isvert$,P,$Work$,$Owns$,$Look$,U,G)

Input: Integer: i, Boolean array: $isvert$, Integer arrays: P, $Work$, $Look$, $Owns = Eloise$ or $Abelard$, Integer list: U,
Graph: $G = (I[n], Accept[n], Owner[n])$

Output: Integer List

1. Process(i,$Work$,$Look$,U)
2. Attractor(i,$isvert$,P,$Work$,$Owns$,$Look$,G,[ ],U,[ ])

The function $Process$ iterates through the vertices in U and alters the values of $Work$ and $Look$ accordingly. For a vertex $j \in U$, $Work[j]$ is set equal to 0 and $Look[j]$ is set equal to the current iteration i. This prevents these vertices being added more than once to the attractor set during the function *Attractor*.

Using the updated value of $Work$ and $Look$, the function *Attractor* then computes the attractor set as described in figure 8.

Figure 8: Attractor(i,$isvert$,P,$Work$,$Owns$,$Look$,G,Z,K,Y)

Input: Integer: i, Boolean array: $isvert$, Integer arrays: $P, Work, Look$, $Owns = Eloise$ or $Abelard$, Graph $G = (I[n], Accept[n], Owner[n])$, Integer lists: Z, K, Y

Output: Integer list

1. If list $K = [\,]$ then:
   1.1 If list $Z = [\,]$ then **return** Y
   1.2 If list $Z = z :: zs$ then:
      1.2.1 Attractor(i,$isvert$,P,$Work$,$Owns$,$Look$,G,[ ],Z,Y)
2. If list $K = k :: ks$ then:
   2.1. Z' := Expand(i,$isvert$,P,$Work$,$Owns$,$Look$,$Owner$,$I[k]$,Z)
   2.2 Attractor(i,$isvert$,P,$Work$,$Owns$,$Look$,G,Z',ks,$k :: Y$)

The list K is used to store those vertices found to be in the attractor set on the previous iteration of the attractor computation and Z is used to store those vertices found so far in the current iteration. For each $j \in K$, the predecessor vertices are checked using the function *Expand* and are added to the list Z if they need to be added to the attractor set. Once the predecessor vertices of j have been considered, j is added to the list Y, which is eventually the output of the function. Once all the vertices in K have been considered, if Z is not empty then the attractor function is called with the list K equal to Z and Z equal to the empty list. If Z is also empty then the function terminates, returning Y.

The function *Expand*, given a list of predecessor vertices, determines which need to be added to the attractor set and thus added to the list Z. This is achieved by considering the values of $isvert$, P, $Work$, $Owns$, $Look$ and $Owner$. First, for each vertex j in the list of predecessor vertices, the value of $isvert[j]$ is checked. If this is false then j is not added to Z. Next, the owner of j is checked using the array $Owner$. If the owner of the vertex matches $Owns$ then the value of $Look[j]$ is checked. If this matches the current iteration i then the vertex has already been added to the attractor set and thus is not added this time. If it does not match i then it is added to Z and $Look[j]$ is set equal to i to prevent the vertex being added again. If the owner of the vertex j does not match $Owns$ then the outgoing edges of j need to be considered. First, the value of $Look$ is examined. If this equals i then the corresponding outgoing edge count $P[j]$ has already been copied to the array $Work$. If it does not equal i then $Work[j]$ is set equal to $P[j]$. The value of $Work[j]$ is then considered. If $Work[j] = 1$ then all successor vertices have been added to the attractor set and thus j is added to Z. If $Work[j] \neq 1$ then there remain some successor vertices that are not included in the attractor set. In this case, the value of $Work[j]$ is decremented by one and j is not added to Z.

4.4 Data Structures for Algorithm 2

We now consider the additional data structures required for the implementation of algorithm 2. The structures required for T_i and W_i for algorithm 2 is analogous to that required in algorithm 1, thus both will be stored as a list.

Since V_i is not used in the computation of T_i in this algorithm, it suffices to just use the boolean array structure from the first algorithm

Note that X_i is the output of an attractor computation and thus will initially be in the form of a list. It is used as the set of vertices the graph is restricted to in the attractor computation in step 8 of algorithm 2. Thus, a data structure for X_i is required such that membership of vertices in X_i can be checked in constant time during the attractor computation. For this purpose, X_i will be stored as a boolean array, which can be computed in time proportional to the size of X_i from the list structure of X_i.

Note also that, during the attractor computation in step 8, a separate count of successor vertices will have to be used that counts only those successor vertices contained in X_i. This can be done by storing such a count in an array initialised to the same value as the original count of successor vertices. When the list X_i is given, this count is calculated by iterating through the successor vertices of vertices in X_i and decrementing the count accordingly by checking the membership of each successor vertex in X_i using the boolean array. For this purpose, since we could be considering $O(m)$ edges, it will be necessary to iterate through the sets of successor vertices in time linearly proportional to their size. Hence, when the count of successor vertices is calculated at the beginning of the algorithm, an array storing lists of successor vertices will also need to be calculated.

In order to calculate T_i in list form efficiently, Z_i will need to be stored in the form of a list in order to iterate through its elements in time linearly proportional to its size. Additionally, it is used in the calculation of D_i, where, in order to make sure the work required for this step is at worst $O(m)$, membership of vertices in Z_i need to be checked in constant time. This is because, in the computation of D_i, successor vertices of elements of Z_i need to be checked for membership in Z_i and at worst $O(m)$ checks could be made. Thus, Z_i, as with X_i, will additionally be stored in the form of a boolean array.

Similarly, L_i will also initially be in the form of the list due to it being the output of an attractor computation. L_i is used to obtain the set T_i by computing $Z_i \backslash L_i$, thus, as with R_i in the computation $V_i \backslash R_i$, L_i will also be stored as a boolean array.

C_1^{i+1} and C_2^{i+1} can be computed at each iteration using W_i as follows:

Clearly $C_2^{i+1} = C_2^i \backslash W_i$ hence C_2^{i+1} can be computed by removing the appropriate vertices from the set C_2^i.

C_1^{i+1} can be calculated by removing elements from C_1^i that are in W_i as with

C_2^{i+1} and keeping a count of accepting successor vertices for each vertex in C_1^i. At the end of each iteration, this count can be decremented by iterating through the predecessor vertices of vertices in W_i and decrementing the count for each vertex accordingly. (Note the count is only decremented if the vertex in W_i is accepting.) When the count reaches 0, the corresponding vertex is added to C_1^{i+1}. This count can be initialised in the first iteration at the same time that the count of successor vertices is calculated by checking whether successor vertices are accepting or not, and is similarly stored in an array. For this purpose, C^i will be stored as a boolean array. This data structure will also facilitate the computation of $Z_i = X_i \cap C_i$.

Thus, it remains to consider the data structures required for C_1^i and C_2^i. Since both can be computed by adding and removing elements at the end of each iteration and since only their union is used in the algorithm, only $C_1^i \cup C_2^i$ will be stored. Since we could be adding or removing $O(n)$ elements at each iteration, both of these operations should take a constant amount of time. Also, since $C_1^i \cup C_2^i$ is used as the input of an attractor computation, in order to be most efficient it would be preferable to iterate though its elements in time linearly proportional to its size. For this purpose, the vertex array, vertex location array structure used for V_i and F_i in the previous implementation will be used.

Finally, the data structures required for the computation of D_i need to be considered. Since D_i will be used as input to an attractor computation, it will be sensible to store it as a list. Note that $X_i \backslash Z_i$ can be obtained efficiently in list form using the list of X_i and the boolean array of Z_i, hence no additional data structures are required for this part of D_i. The remaining part of D_i is calculated by iterating through the successor vertices of vertices in Z_i, hence the list structure of Z_i will be required. Also, since we could be considering $O(m)$ edges it will be necessary to iterate through the successor vertices in time linearly proportional to their size. For this purpose, the array storing the lists of successor vertices will be used.

4.5 Algorithm 2 Implementation

Using the data structures given, figure 9 outlines algorithm 2.

The function $Initialise'$ calculates the number of successor vertices for each vertex and stores this count in the array P, similarly to the function $Initialise$. It also initialises the array Out which stores lists of successor vertices. This is achieved by iterating through each list of predecessor vertices in the array I.

The function $Initialise2$ iterates through the vertices in $Flist$ and sets the corresponding value in the array $coBuchi$ to false. In addition to this, the arrays $CountC$ and PX are set equal to the array P.

Figure 9: Algorithm 2 Implementation

Input: Graph $G := (I[n], Owner[n])$, Integer list $Flist$, Integer v_0

Output: $True$ or $False$

1. **Create arrays** $isvert[n]$, $P[n]$, $PX[n]$, $coBuchi[n]$, $X[n]$, $Z[n]$, $L[n]$, $CountC[n]$, $C[n+1]$, $Cloc[n]$, $LookC[n]$, $LookD[n]$, $LookT[n]$, $work[n]$, $Out[n]$
2. **For** $0 \leq j \leq n-1$: $isvert[j] := true$, $P[j] := 0$, $PX[j] := 0$, $coBuchi[j] := true$, $X[j] := false$, $Z[j] := false$, $L[j] := false$, $CountC[j] := 0$, $C[j] := n+1$, $Cloc[j] := n$, $LookC[j] := -1$, $LookD[j] := -1$, $LookT[j] := -1$, $Work[j] := 0$, $Out[j] := [\,]$
3. $C[n] := 0$
4. $Initialise'(P, Out, G)$
6. $Initialise2(P, coBuchi, PX, CountC, Flist)$
7. $ComputeC(Outgoing, coBuchi, CountC, C, Cloc, G)$
8. $MainAlgorithm2(0, G, v_0, isvert, C, Cloc, P, PX, coBuchi, X, Z$
   $L, CountC, LookC, LookD, LookT, Work, Out)$

The function $ComputeC$ is used to initialise the C, $Cloc$ structure. It iterates through all lists of predecessor vertices stored in the array I. If for a vertex j and a predecessor vertex k, $coBuchi[j]$ and $coBuchi[k]$ are both true (i.e. neither are accepting) then the owner of vertex k is checked. If k is owned by Eloïse then $CountC[k]$ is decremented by one whereas if it is owned by Abelard then $CountC[k]$ is set equal to 0. Once all lists of predecessor vertices have been considered, the value of $CountC$ for each vertex is checked. If for a vertex j, $CountC[j] = 0$, then j is added to the data structure C, $Cloc$ and $CountC[j]$ is set equal to -1. This prevents j being added to C a second time later in the implementation. If $CountC[j] \neq 0$ then the vertex is not added to C.

The function $MainAlgorithm2$ is described in figure 10.

Given the set X, the function $CountOutX$ sets the count PX of successor vertices restricted to the set X to the correct value. It achieves this by iterating through $Xlist$. For each $j \in Xlist$, the list of successor vertices $Out[j]$ is considered. For every k in this list of successor vertices for which $X[k]$ is false, the value of $PX[j]$ is decremented by one.

The function $ComputeZ$ returns the list $Zlist$ by iterating through $Xlist$ and adding a vertex j to $Zlist$ if $coBuchi[j]$ is true.

Figure 10: MainAlgorithm2

Input: Integer: i, v_0, Graph: $G := (I[n], Owner[n])$
Boolean Arrays: $isvert$, $coBuchi$, X, Z, L
Integer Arrays: P, PX, $CountC$, C, $Cloc$,
$LookC$, $LookD$, $LookT$, $Work$
Integer List Array: Out

Output: True or False

1. Create list $Clist$ of elements in array C
2. $Xlist := Attractorcomp(i, isvert, P, Work, Abelard, LookC, Clist, G)$
3. **For each** $j \in Xlist$: $X[j] := true$
4. $CountOutX(PX, Out, Xlist, X)$
5. $Zlist := ComputeZ(coBuchi, Xlist)$
6. **For each** $j \in Zlist$: $Z[j] := true$
7. $D := InitD(Xlist, Z)$
8. $D := ComputeD(G, isvert, Out, Zlist, Z, D)$
9. $Llist := Attractorcomp(i, X, PX, Work, Eloise, LookD, D, G)$
10. **For each** $j \in Llist$: $L[j] := true$
11. $T := ComputeT2(Zlist, L)$
12. $W := Attractorcomp(i, isvert, P, Work, Abelard, LookT, T, G)$
13. If $W = [\ ]$ then **return** $true$ else continue to step 14
14. If $v_0 \in W$ then **return** $false$ else continue to step 15
15. **For each** $j \in W$:
    15.1 $isvert[j] := false$
    15.2 $Remove(j, C, Cloc)$
    15.3 **For each** $v \in I[j]$:
        15.3.1. $P[v] := P[v] - 1$
        15.3.2. $AddToC(isvert, coBuchi, CountC, C, Cloc, v, j)$
16. $ReinitXZL(Xlist, X, Zlist, Z, Llist, L, PX, P)$
17. $MainAlgorithm2(i + 1, G, v_0, isvert, C, Cloc, P, PX, coBuchi, X,$
$Z, L, CountC, LookC, LookD, LookT, Work, Out)$

The function $InitD$ finds the complement of X and Z using $Xlist$. This is then returned as a list which corresponds to the first vertices to be added to D. The function $ComputeD$ is then called to compute the remaining vertices in the list D. This process is outlined in figure 11. In step 11, $ComputeT2$ returns the list T using $Zlist$ and L.

Once the list W has been computed, step 15 updates the data structures by removing vertices that are in W. In addition to this, in step 15.3.2, vertices

are considered for addition into the set C using the function $AddToC$. Given a vertex j to be removed from the graph and its predecessor k, if $isvert[k]$ is true, the value of $coBuchi[j]$ is considered. If this is false then $CountC[k]$ is decremented by 1. If $CountC[k]$ is now 0 then k is added to C and $CountC[k]$ is set to -1 to prevent k being added more than once.

In step 16, $ReinitXZL$ uses $Xlist$, $Zlist$ and $Llist$, to reinitialise X, Z, L, and PX for the next iteration. In order to set PX to the correct value for the next iteration, for each vertex $j \in Xlist$, $PX[j]$ is set to the value of $P[j]$.

Figure 11: ComputeD

Input: Graph: $G = (I[n], Owner[n])$, Boolean Array: $isvert$,
Integer List Array: Out, Integer Lists: $Zlist$, D, Boolean Array: Z

Output: Integer List

1. If list $Zlist = [\,]$ then **return** D

2. If list $Zlist = z :: zs$ then:

2.1 If $Check(isvert, Z, Out, Owner, z) = true$ then
$ComputeD(G, isvert, Out, zs, Z, z :: D)$

2.2 If $Check(isvert, Z, Out, Owner, z) = false$ then
$ComputeD(G, isvert, Out, zs, Z, D)$

ComputeD iterates through the vertices $j \in Zlist$ adding them to the list D if the function $Check$ returns true. This function begins by considering the owner of the vertex in question. If $Owner[j] = Eloise$ then $Check$ iterates through the successor vertices $Out[j]$ of j. If there exists a successor vertex k such that $Z[k]$ is false and $isvert[k]$ is true then $Check$ returns true. If no such vertex exists then false is returned. In the case where $Owner[j] = Abelard$, $Check$ also iterates through $Out[j]$. This time, however, if there exists a successor vertex k such that $Z[k]$ is $true$ and $isvert[k]$ is true then $Check$ returns $false$. If no such vertex exists then true is returned.

4.6 Algorithm 3 Implementation

Clearly the data structures used for algorithm 3 are inherited from the implementations of algorithms 1 and 2. Due to the nature of algorithm 3, all the data structures used in both algorithm 1 and 2 have to be initialised and updated at each iteration. Thus it remains to consider how to dovetail the two algorithms.

In order to simplify the process, only the more expensive computations are dovetailed. More precisely only those processes with $O(m)$ time complexity are computed simultaneously at each iteration. Those with $O(n)$ time complexity

are computed individually when required. As well as simplifying the implementation, this allows the algorithm that is progressing faster to be affected less by the simultaneous computation of the other algorithm. Thus only the attractor set and the D_i set computations are dovetailed. We therefore need to implement the following two functions: A function which dovetails two attractor computations and a function which dovetails an attractor computation with a D_i set computation. The computation of attractor sets and D_i both progress by inspecting the edges of the game graph. Thus the function providing simultaneous computation operates by alternating between algorithms each time an edge is inspected.

During two simultaneous attractor/D_i set computations, when one completes, it is noted which algorithm this corresponds to and any subsequent $O(n)$ computations for that algorithm are carried out. If T_i has still not been returned then algorithm 3 resumes computing both algorithms simultaneously. For the algorithm corresponding to the computation that completed previously, the next attractor/D_i computation is begun. For the algorithm corresponding to the computation that did not complete previously, the partialy completed attractor/D_i set computation is continued.

4.7 Winning Strategy Implementation

As described in Section 3, winning strategies for all three algorithms are computed using attractor strategies. These attractor strategies are decided during attractor computations. Every time a vertex is added to an attractor set, the strategy for that vertex needs to be altered to the successor that caused the vertex to be added. Thus in order not to interfere with the time complexity of the algorithms, the strategy for each vertex needs to be updated in constant time. For this purpose, a strategy will be represented by an integer array. An integer stored in index i of the array corresponds to the move the owner of vertex i should make at that vertex.

5 Integration Into THORS Model Checker

In this section, the integration of a Büchi game solver into the model checker THORS [14] is outlined.

For the verification of higher-order functional programs, the purpose of THORS is to model check higher-order recursion schemes (HORS) against the alternation-free modal μ-calculus. The alternation-free modal μ-calculus corresponds to formulas of the modal μ-calculus which have an alternation depth of 1 or less. By integrating a Büchi game solver, THORS will be able to solve the model checking problem for HORS against the $\Pi_2^\mu \cup \Sigma_2^\mu$ fragment of the modal μ-calculus.

5.1 The Model Checking Problem for HORS and the Modal μ-Calculus

In order to define HORS, we begin by introducing the set of kinds which is defined by the grammar $A ::= o \mid A_1 \rightarrow A_2$. For a given term t, we write $t : A$ to denote that t has kind A. Note that it is more usual to use the word type for this definition, however, to avoid a clash of terminology later in this section, the word kind is used instead. The *order* and *arity* of a kind are defined inductively as follows:

- $ord(o) := 0$

  $ord(A_1 \rightarrow A_2) := max(ord(A_1) + 1, ord(A_2))$

- $arity(o) := 0$

  $arity(A_1 \rightarrow A_2) := arity(A_2) + 1$

We give the following definition from [9]:

A higher-order recursion scheme $R := (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where:

- Σ is a finite, ranked alphabet of symbols known as terminals (i.e. each terminal $a \in \Sigma$ has a kind of order 1 or 2).
- $\mathcal{N}$ is a finite set of kinded symbols known as non-terminals.
- S is a special non-terminal called the start symbol and has kind o.
- $\mathcal{R}$ is a map from $\mathcal{N}$ to λ terms of the form $\lambda x_1 \cdots x_n.e$ where e is a kinded term constructed from non-terminals, terminals and variables as defined below. Each $x_i : A_i \in Var$ where Var is a set of kinded variables.

We define the set of kinded terms inductively as follows:

- Terminals, non-terminals and variables of kind A are terms of kind A
- Given a term $t_1 : A_1 \to A_2$ and a term $t_2 : A_1$, their application $t_1\ t_2$ is a term of kind A_2.
- Given $F \in \mathcal{N}$, where $\mathcal{R}(F) = \lambda x_1 \cdots x_n.e$, the terms $F\ x_1 \cdots x_n$ and e must be terms of kind o and the variables that occur in e must be contained in $\{x_1 \cdots x_n\}$.

Given a HORS R, the rewriting relation $\to$ is defined inductively as follows:

Given terms $s_1 \cdots s_n \in \Sigma$ and $F \in \mathcal{F}$ we write

- $F\ s_1 \cdots s_n \to ([s_1/x_1], \cdots, [s_n/x_n])e$ if $\mathcal{R}(F) = \lambda x_1 \cdots x_n.e$
- If $t \to t'$ then $t\ s \to t'\ s$ and $s\ t \to s\ t'$

In order to demonstrate this, consider the following HORS:

$R := (\mathcal{N}, \Sigma, \mathcal{R}, S)$ where:

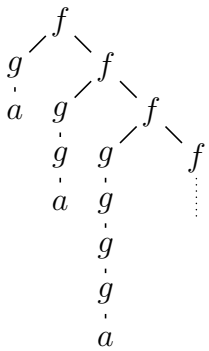
- $\Sigma = \{f : o \to o \to o,\ g : o \to o,\ a : o\}$ with respective arities 2, 1 and 0.
- $\mathcal{N} = \{S : o,\ B : (o \to o) \to (o \to o) \to o \to o,\ F : (o \to o) \to o\}$
- The set of rewrite rules R is defined as follows:

$$\begin{aligned} S &\to F\ g \\ F\ x_1 &\to f\ (x_1\ a)\ (F\ (B\ x_1\ x_1)) \\ B\ x_1\ x_2\ x_3 &\to x_1\ (x_2\ x_3) \end{aligned}$$

Thus by expanding S using the rewriting relation, we get

$$S \to F\ g \to f\ (g\ a)(F\ (B\ g\ g)) \to f\ (g\ a)(f\ (g\ (g\ a))(F\ (B\ (B\ g\ g)(B\ g\ g)))) \to \cdots$$

which gives rise to the following infinite tree:



Thus HORS can be used to define a family of finitely branching infinite trees.

We now define an alternating parity tree automata, beginning with the following definition as given in [14]:

The set $\mathsf{B}^+(X)$ of positive boolean formulas over a set X is defined inductively as follows:

- t, f and $x \in \mathsf{B}^+(X)$ for all $x \in X$
- For each $\theta_1, \theta_2 \in \mathsf{B}^+(X)$, $\theta_1 \vee \theta_2$ and $\theta_1 \wedge \theta_2 \in \mathsf{B}^+(X)$

A set $Y \subseteq X$ is said to satisfy a formula $\theta \in \mathsf{B}^+(X)$ if assigning all the elements of Y to true and all the elements of $X \backslash Y$ to false results in θ being true. Thus consider an alternating parity tree automaton $\mathcal{A}$:

$\mathcal{A} := \langle \Sigma, Q, \delta, q_0, \Omega \rangle$ where:

- Σ is a ranked alphabet. Let $m \in \mathbb{N}$ be the largest arity of the elements of this alphabet
- Q is a finite set of states
- $\delta : Q \times \Sigma \to \mathsf{B}^+(\{1, ..., m\} \times Q)$ is the transition function such that for each $a \in \Sigma$ and $q \in Q$, $\delta(q, a) \in \mathsf{B}^+(\{1, ..., k\} \times Q)$ where k is the arity of a.
- $q_0 \in Q$ is the initial state.
- $\Omega : Q \to \mathbb{N}$ is the priority function

A Σ labelled tree t can be thought of as a function $t : dom(t) \to \Sigma$ where $dom(t) \subseteq \{0, ..., m\}^*$. Thus a sequence $\alpha = \alpha_0\alpha_1, ... \in \{0, ..., m\}^*$ represents a path in the tree where α_i represents which branch of the tree the path travels down at depth i. The value of $t(\alpha)$ then represents the label at that point in the tree and $t(\epsilon)$ represents the root of the tree.

A run-tree of an alternating parity automaton on such a tree t, is a $(dom(t) \times Q)$-labelled tree r such that:

- $\epsilon \in dom(r)$ with $r(\epsilon) = (\epsilon, q_0)$
- If $\beta \in dom(r)$ with $r(\beta) = (\alpha, q)$ then there exists a possibly empty set S such that S satisfies $\delta(q, t(\alpha))$ and for each (i, q') in S there exists a j such that $\beta j \in dom(r)$ and $r(\beta j) = (\alpha i, q')$.

A run-tree is said to be accepting if for a labelled infinite path $(\alpha_0, q_0), (\alpha_1, q_1), ...$ in the tree, the minimum priority that occurs infinitely often in $\Omega(q_0), \Omega(q_1), ...$ is even. Thus a tree t is accepted by $\mathcal{A}$, if there exists an accepting run-tree.

As might be expected, this definition specialises to an alternating Büchi tree automaton when the priority function maps all states to 0 (an accepting state) or 1 (a non-accepting state). This can be specialised further to an alternating *weak* Büchi tree automaton if there exists a partial order $\leq$ over a partition $\{Q_1, ..., Q_n\}$ of Q such that for each $i \in \{1, ..., n\}$, states in Q_i are either all labelled as accepting or all labelled as non-accepting. Also, if a state $q' \in Q_j$ occurs in $\delta(q, a)$ for some $q \in Q_i$ and $a \in \Sigma$, then $Q_i \leq Q_j$. Thus, all infinite paths in a valid run-tree will eventually remain in some element Q_i of the partition.

To demonstrate an alternating Büchi tree automaton, consider the following example:

$\mathcal{A} = \langle \Sigma, Q, \delta, q_0, \Omega \rangle$ where:

- $\Sigma = \{f, g, a\}$ with respective arities 2, 1 and 0.
- $Q = \{q_0, q_1, q_2\}$
- $\delta(q_0, f) = (1, q_0) \wedge (2, q_0)$, $\delta(q_0, g) = (1, q_1)$, $\delta(q_0, a) = \mathsf{t}$

  $\delta(q_1, f) = \mathsf{f}$, $\delta(q_1, g) = (1, q_2)$, $\delta(q_1, a) = \mathsf{f}$

  $\delta(q_2, f) = (1, q_0) \wedge (2, q_0)$, $\delta(q_2, g) = (1, q_1)$, $\delta(q_2, a) = \mathsf{t}$
- $\Omega(q_0) = 0$, $\Omega(q_1) = 1$, $\Omega(q_2) = 1$

$\mathcal{A}$ accepts a Σ labelled tree t if in every path of t, whenever g occurs, it occurs a finite, even number of consecutive times. For a tree t, a run tree is in state q_0 when g is not occurring. It is in state q_1 when an odd number of g's have occurred consecutively and q_2 when an even number of g's have occurred. Note that the priorities of the states are such that in an infinite path, the state q_0 must occur infinitely often for the corresponding tree to be accepted. Thus a tree with an infinite path where eventually only the symbol g occurs is not accepted. The tree corresponding to the example HORS given above is not accepted since there is a branch of the tree where one g occurs.

Note that in the above alternating Büchi tree automaton $\mathcal{A}$, if all occurrences of $\wedge$ were replaced with $\vee$ then $\mathcal{A}$ would check for the property that there *exists* a path where whenever g occurs, it occurs a finite, even number of consecutive times rather than checking all paths. In this case, the example tree given above would be accepted.

The purpose of THORS is to check whether the infinite tree defined by a higher-order recursion scheme is accepted by an alternating weak Büchi tree automaton. This can be shown in [11] to be equivalent to checking whether a property specified by an alternation free modal μ-calculus formula holds for an infinite tree.

It has been shown in [17] that solving the model checking problem for the modal μ-calculus can be translated into solving the acceptance problem for alternating parity tree automata. The states of the alternating parity tree automata resulting from this translation are subformulas of modal μ-calculus formulas. In particular, the priority of these states is inherited from a modal μ-calculus formula according to the alternation depth in such a way that the translation of a formula in the $\Pi_2^\mu \cup \Sigma_2^\mu$ fragment of the modal μ-calculus results in an alternating Büchi tree automata.

In [14], by modifying a result from [9], given an alternating weak Büchi tree automaton $\mathcal{A}$, and a HORS R, the acceptance problem is solved by constructing an intersection type system $\lambda_\wedge^\mathcal{A}$ such that the infinite tree associated with R is accepted by $\mathcal{A}$ if and only if R is well-typed in $\lambda_\wedge^\mathcal{A}$. We define R to be well-typed in $\lambda_\wedge^\mathcal{A}$ if Eloïse has a winning strategy for a corresponding Büchi game $\mathcal{G}^R_{\lambda_\wedge^\mathcal{A}}$ defined later. In fact, due to the fact that the alternating Büchi tree automaton is weak, this game is a weak Büchi game. A Büchi game is defined to be weak if there exists a partition $\{V_1, ..., V_n\}$ of vertices such that: (i) The vertices in a given element of the partition are either all accepting or all not accepting, (ii) For a vertex $v \in V_i$, all successor vertices of v are contained in one of $V_i, V_{i+1}, ...V_n$ so that a play must eventually remain in one of the elements of the partition. In order to solve the acceptance problem for alternating weak Büchi tree automata, THORS uses a weak Büchi game solver which runs in linear time in the worst case.

For solving the acceptance problem for HORS on all alternating Büchi tree automata, the intersection type system used in [14] can be easily altered although the corresponding Büchi game will no longer be weak. Hence integrating a Büchi game solver into THORS will facilitate the solving of the acceptance problem for all alternating Büchi tree automata and thus solve the model checking problem for HORS against the $\Pi_2^\mu \cup \Sigma_2^\mu$ fragment of the model μ-calculus.

In order to define this corresponding Büchi game we introduce the following definitions from [14]:

Given $\mathcal{A}$ and R, for the type system $\lambda_\wedge^\mathcal{A}$, types are defined by the grammar:

$$\begin{aligned} \theta &::= \tau \to \theta \text{ where } q \in Q \\ \tau &::= \bigwedge\{\theta_1, ..., \theta_k\} \text{ where } k \geq 0 \end{aligned}$$

Thus types are formed from the states of $\mathcal{A}$ and are of the form $\theta = \tau_1 \to \cdots \to \tau_n \to q$. The state of θ is defined to be q and is written $state(\theta)$. The notion of well-kindedness for these types is defined using the following inductive rules:

$$\frac{}{q_0 :: o} \quad \frac{\tau :: A_1 \ \theta :: A_2}{\tau \to \theta :: A_1 \to A_2} \quad \frac{\theta_i :: A \text{ for each } i \in \{1, ...k\}}{\bigwedge_{i=1}^{k} \theta_i :: A}$$

A type environment Γ is a set of type bindings $F : \theta$ where F is a non-terminal of a given HORS and θ is a type. Thus we say that a type judgement $\Gamma \vdash t : \theta$, where t is a λ-term, is valid if it satisfies the rules of the given type system $\lambda^{\mathcal{A}}_{\wedge}$.

A type binding $F : \theta$ for $F \in \mathcal{N}$ is said to be R-consistent if for some kind A, $F : A$ and $\theta :: A$.

Thus given an alternating Büchi tree automata $\mathcal{A} := \langle \Sigma, Q, \delta, q_0, \Omega \rangle$, and a HORS $R := \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, a corresponding Büchi game is constructed using the type system $\lambda^{\mathcal{A}}_{\wedge}$ as follows:

$\mathcal{G}^{R}_{\lambda^{\mathcal{A}}_{\wedge}} := (G, v_0, F)$ with $G = (V, E, \lambda)$ where:

- $V := V_E \cup V_A$ where:
  - $V_E := \{(F : \theta) \mid F : \theta \text{ is } R\text{-consistent } \}$
  - $V_A := \{(\Gamma, q) \mid \exists F : \theta \text{ s.t. } \Gamma \vdash \mathcal{R}(F) : \theta \text{ and } q = state(\theta)\}$
- $E := \{((F : \theta), (\Gamma, state(\theta))) \mid \Gamma \vdash \mathcal{R}(F) : \theta\} \cup \{((\Gamma, q), (F : \theta)) \mid F : \theta \in \Gamma\}$
- $\lambda(v) = \exists$ for $v \in V_E$ else $\lambda(v) = \forall$
- $v_0 := (S : q_0)$
- For $v = (F : \theta) \in V_E$, if $\Omega(state(\theta)) = 0$ then $v \in F$ else $v \notin F$
- For $v = (\Gamma, q) \in V_A$, if $\Omega(q) = 0$ then $v \in F$ else $v \notin F$

If the alternating Büchi tree automaton is weak then the type system is such that the derived Büchi game is also weak.

As previously mentioned, a HORS is well-typed in $\lambda^{\mathcal{A}}_{\wedge}$ if Eloïse has a winning strategy in the corresponding game. Eloïse attempts to prove that it is well typed, and Abelard attempts to prove it is not. At each Eloïse vertex, Eloïse has to provide a type environment for which $\mathcal{R}(F)$ has type θ. Abelard then picks an element $F' : \theta'$ of the type environment for which Eloïse must similarly provide a type environment in which $\mathcal{R}(F')$ has type θ' and the game continues in this way.

It turns out that constructing the Büchi game directly as described above is unnecessarily expensive. This is due to the fact that it contains all possible type bindings $(F : \theta)$, many of which are not reachable from the initial vertex.

Though still in development, the idea of THORS is to reduce the size of this game, by restricting the possible vertices to those contained in a reachable part of the game which corresponds to a valid run-tree. THORS extends the approach in [8], in which a type environment is inferred which witnesses the existence of some valid run-tree. In THORS, once such a type environment Γ is found, in order to determine acceptance, a reduced Büchi game $\mathcal{G}^R_\Gamma$ is constructed and solved. This is achieved by computing a function $TEnv_\Gamma$ which maps the bindings $F : \theta$ to their successor vertices $\{\Delta \subseteq \Gamma \mid \Delta \vdash \mathcal{R}(F) : \theta\}$:

$\mathcal{G}^R_\Gamma := (G, v_0, F)$ with $G = (V, E, \lambda)$ where:

- V is defined inductively as follows:
  - $(S : q_0) \in V$
  - Given $(F : \theta) \in V$, if $\Delta \in TEnv_\gamma(F : \theta)$ then $(\Delta, state(\theta)) \in V$
  - Given $(\Delta, q) \in V$, all $(F : \theta) \in \Delta$ are in V
- $E := \{((F : \theta), (\Delta, state(\theta))) \mid \Delta \vdash \mathcal{R}(F) : \theta\} \cup \{((\Delta, q), (F : \theta)) \mid F : \theta \in \Delta\}$
- $\lambda(v) = \exists$ for $v \in V_E$ else $\lambda(v) = \forall$
- $v_0 := (S : q_0)$
- For $v = (F : \theta) \in V$, if $\Omega(state(\theta)) = 0$ then $v \in F$ else $v \notin F$
- For $v = (\Delta, q) \in V$, if $\Omega(q) = 0$ then $v \in F$ else $v \notin F$

If Eloïse does not have a winning strategy then the run-tree witnessed by the type environment Γ is not accepting and a type environment Γ' is found that contains Γ and witnesses additional run-trees. The corresponding game for Γ' is then calculated and solved.

Thus given an alternating Büchi tree automaton $\mathcal{A}$ and a HORS R, the approach of THORS to this acceptance problem is as follows:

1. Construct a type environment Γ from the intersection type system $\lambda^{\mathcal{A}}_\wedge$ which witnesses some run-tree of the infinite tree defined by R on $\mathcal{A}$. If no such Γ exists then no valid run-trees exist and thus R is not accepted on $\mathcal{A}$.

2. Using Γ, construct a Büchi game that is considerably smaller than the game derived directly from $\lambda^{\mathcal{A}}_{\wedge}$ and corresponds to some run-tree so that Eloïse has a winning strategy for this game if and only if the run-tree is accepting.

3. Repeat if Eloïse does not have a winning strategy, constructing a larger type environment Γ'.

If a recursion scheme R is accepted by $\mathcal{A}$ then this process will terminate. If, however, it is not accepted then it is possible this process will not terminate. In order to solve this problem we consider the automata $\mathcal{A}'$ which accepts exactly those trees which $\mathcal{A}$ does not accept. Such an automata can be constructed from $\mathcal{A}$. If given R, the process given above does not terminate for $\mathcal{A}$ then it will terminate for $\mathcal{A}'$. Thus, THORS runs the process given above on $\mathcal{A}$ and its complement $\mathcal{A}'$ in parallel. If Eloïse is found to have a winning strategy in the first case then R is accepted by $\mathcal{A}$. If Eloïse has a winning strategy in the second case then R is not accepted.

5.2 Integration Of A Büchi Game Solver

In order to integrate a Büchi game solver implemented in Section 4, using the function $TEnv_{\Gamma}$, it will be necessary to construct a game in the form used in Section 4. Thus we need to construct a game where each vertex is represented by an integer and the game is stored as an array of records. Each record will contain the owner for that vertex and a list of predecessor vertices.

In order to achieve this, the set of vertices reachable from the initial vertex $(S : q_0)$ need to be explored using $TEnv_{\Gamma}$. Initially, the successor vertices of $(S : q_0)$, which are type environment, state pairs, are found by evaluating $TEnv_{\Gamma}(S : q_0)$. The successor vertices of these are then known to be the set of type bindings within each type environment. The successor vertices for these type bindings are then found using $TEnv_{\Gamma}$ again and the graph continues to be expanded in this manner until all reachable vertices have been found. All reachable vertices have been found when $TEnv_{\Gamma}$ has been evaluated for all type bindings found so far and all successor vertices found from these have already been discovered.

Each time a new vertex is found, it needs to be assigned an individual vertex number so that it has a location in the array structure of the game that is to be constructed. For this purpose, when expanding the game graph with $TEnv_{\Gamma}$, we need to be able to identify which vertices have already been encountered. This prevents a vertex being assigned more than one integer and thus being added to the game more than once. Hence we need a data structure which stores each vertex and for which membership of a vertex in this data structure can be evaluated efficiently. For this purpose, initially, vertices which have

been found will be stored in a hash table. In the hash table, each vertex will be bound to the number it has been assigned. The advantages of using a hash table are that the membership of a vertex in a hash table can be checked efficiently and the exact number of vertices does not have to be known initially. Note, however, that in order for the hash table to operate efficiently, the *approximate* number of vertices has to be known. Since the size of the game is restricted by the size of Γ, the number of vertices in the game graph can be estimated in order to create a hash table with approximately the correct size. The initial vertex is assigned the number 0 and as new vertices are found, the number assigned to each vertex is incremented.

In addition to assigning each vertex a number, the set of predecessor vertices also needs to be found for each vertex and stored as a list. These lists will also be stored in the hash table bound to the corresponding vertex and are computed by storing vertices paired with their predecessor vertices in a list and processing that list as follows:

1. For a given vertex, find the successor vertices and add them, paired with the original vertex, to the list of vertex, predecessor vertex pairs.

2. Iterate through the pairs of vertices stored in the list, checking whether each successor vertex has already been added to the hash table.

   (a) If a vertex has already been added then add the predecessor vertex stored in the pair to the list of predecessor vertices stored in the hash table.

   (b) If a vertex has not been added, assign it a vertex number, add it to the hash table with a list containing the paired predecessor vertex and then go to step 1.

Thus it is possible to efficiently compute a unique vertex number and a list of predecessor vertices for each vertex. Hence, by iterating though and counting the number of elements in the hash table, we can create an array where each vertex is placed in the array according to its vertex number. The array then stores the list of predecessor vertices from the hash table and the owner of each vertex. The owner is determined by simply observing whether a vertex is a type binding or a non-terminal, state pair. For the former, the owner is Eloïse and for the latter, the owner is Abelard.

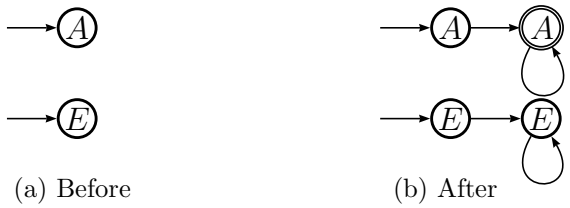
In Thors the acceptance condition for a given alternating weak Büchi tree automaton is stored as an array that corresponds to the partition of states. Given a partition $Q_1, ..., Q_n$, the i^{th} element of the corresponding array stores a record that contains the set of states in Q_i and whether that partition is accepting or not. In Thors the given automaton is checked to observe whether it has a valid partition and if this is the case, a weak Büchi game solver is used. In order to expand the range of possible inputs to all alternating Büchi tree

automata, if the automaton is not found to have a valid partition then the set of all accepting states are found and stored as a list of states and the Büchi game solver is used. The Büchi game solver requires the accepting vertices to be stored in a list, which can be generated at the same time as the game graph using the list of accepting states. When a vertex is added to the hash table as described above, the vertex is checked to see whether it is accepting or not. If it is accepting then its corresponding vertex number is added to the list of accepting vertices. In the case of a vertex corresponding to a type binding $F : \theta$, it is determined whether the vertex is accepting by checking the membership of $state(\theta)$ in the list of accepting states. If the vertex corresponds to a non-terminal state pair (Γ, q), then the vertex is accepting if q is in the list of accepting states.

The only remaining problem for generating the correct game is that not all vertices are guaranteed to have a successor vertex. It is possible for a vertex to have no successor vertices, such as when the type environment for an Abelard vertex is empty. Recall that the algorithms for solving a Büchi game require that all vertices have at least one successor vertex, thus the game graph is currently not in a suitable form for the solver. This problem is easily addressed by modifying the game graph as follows:

- When a terminating vertex is found add an extra vertex to the hash table which has the terminating vertex and itself as predecessor vertices.
- If the terminating vertex is owned by Abelard, and therefore a play reaching this point is winning for Eloïse, the extra vertex is added to the list of accepting vertices.

Figure 12: Eliminating Terminating Vertices



(a) Before (b) After

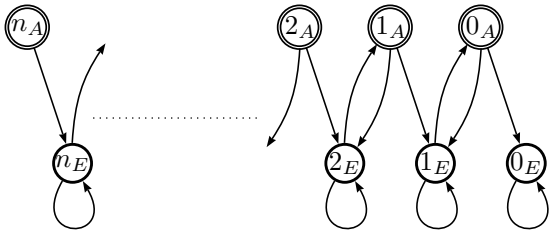
Clearly, the game graph is now in the correct form with no terminating vertices while maintaining the same winning sets. Thus it is now possible to integrate a Büchi solver from Section 4 into the model checker THORS.

6 Testing

In this section, the efficiency of all three algorithms are compared and contrasted using various different Büchi games. Initially, the three algorithms will be tested on Büchi games constructed specifically to demonstrate the advantages and disadvantages of each algorithm. Later, this will be extended to testing on random Büchi games and finally the Büchi game solvers will be compared with a selection of parity game solvers. For the purposes of testing, in this section, all the Büchi game solvers will return a partition of the vertices into Abelard and Eloïse winning sets as described in Section 3. This is so as to ensure that the solvers run for the maximum number of iterations on each example. All tests carried out in this section were run on a machine using the operating system Fedora with kernal version-2.6.29.6, with an AMD Athlon(tm) 64 X2 Dual Core Processor 4600+ and with 2GB of RAM.

6.1 Comparing Büchi Game Solvers

Recall the example given in Section 3.



It was noted that when algorithm 1 is used, this Büchi game is solved in $O(n^2)$ time, whereas with algorithm 2, it is solved in $O(n)$ time. Thus algorithm 3 should also solve this game in $O(n)$ time. The results of running the implementations of all three algorithms on this example are shown in table 1. The construct time is the time taken to initialise all data structures required at the start of each implementation and the solve time is the time taken to solve the game after such data structures have been initialised. All results are given in seconds. If the total execution time is over 4 minutes then the corresponding results are not recorded.

As expected, both algorithms 2 and 3 are considerably faster than algorithm 1. Although algorithm 3 is slower than algorithm 2, it is still running in linear time. Thus, given this example, it would be preferable to use algorithm 2 or 3 over algorithm 1.

Table 1: Test 1 Algorithms 1-3

	Construct Time (s)			Solve Time (s)			Total Time (s)		
n	1	2	3	1	2	3	1	2	3
1000	0.01	0.02	0.02	2.15	0.09	0.13	2.16	0.11	0.15
2000	0.02	0.03	0.05	8.70	0.20	0.22	8.72	0.23	0.27
3000	0.02	0.06	0.07	19.77	0.29	0.23	19.79	0.35	0.30
5000	0.04	0.07	0.09	57.31	0.37	0.57	57.35	0.44	0.66
7000	0.07	0.12	0.16	114.38	0.61	0.80	114.45	0.73	0.96
9000	0.09	0.16	0.17	190.92	0.67	0.87	191.01	0.83	1.04
10000	0.10	0.14	0.17	235.51	0.88	1.25	235.61	1.02	1.42
50000	-	0.50	0.50	-	4.51	6.63	-	5.01	7.13
100000	-	0.84	0.98	-	9.23	12.61	-	10.07	13.59
1000000	-	8.47	11.53	-	93.1	130.54	-	101.57	142.07

It has previously been discussed that algorithm 1 is preferable when R_i is relatively small. Note, however, that a small R_i implies that T_i is relatively large and thus, at the iteration in question, a large proportion of vertices are removed from the graph. Thus this limits the advantages of algorithm 1 since R_i being small restricts the number of iterations required to solve the game.

The advantages of algorithm 1 are explored using the following examples:

Consider the following inductive definition of a game given t, s and $l \in \mathbb{N}$:

- $\mathcal{G}_{t,s,l} := (G_{t,s,l}, v_0, F)$ where $G_{t,s,l} := (V_{t,s,l}, E_{t,s,l}\lambda)$
- $V_{t,s,l} := \{0, ..., (size_{t,s,l} - 1)\}$ where:
  - $size_{t,s,1} := t$
  - $size_{t,s,l} := s * size_{t,s,l-1} + size_{t,s,l-1}$ for $l > 1$
- $E_{t,s,l} := E^1_{t,s,l} \cup E^2_{t,s,l}$ where:
  - $E^1_{t,s,l} := \{(0,1), (1,2), ..., (n-2, n-1)\}$ where $n = size_{t,s,l}$
  - $E^2_{t,s,l} := \{(size_{t,s,1} - 1, size_{t,s,1} - 1), (size_{t,s,2} - 1, size_{t,s,2} - 1), ..., (n-1, n-1)\}$
- $\lambda(v) = \exists$ for all $v \in V_{t,s,l}$
- $F := \{0, size_{t,s,1}, size_{t,s,2}, ..., size_{t,s,l-1}\}$

Thus the graph consists of l sections, each s times the sum of the size of the previous sections. The sections are connected by an incoming edge from the last element of the previous section to the first element of the current section. Each section is in the following form:

When solving this Büchi game, at the i^{th} iteration, the i^{th} section is removed. Since the i^{th} section is s times larger than that of the sum of all the previous sections, T_i will be s times larger than R_i. Hence, for all l iterations, algorithm 1 should perform better than algorithm 2. Results for this type of game are displayed in tables 2 and 3.

Table 2: Test 2.1, Algorithms 1-3 with $s = 2$ and $t = 2$

	Construct Time (s)			Solve Time (s)			Total Time (s)		
l	1	2	3	1	2	3	1	2	3
10	0.14	0.21	0.23	0.15	0.38	0.21	0.29	0.58	0.43
11	0.43	0.45	0.49	0.73	1.05	0.58	1.17	1.50	1.07
13	2.19	5.08	5.72	3.94	10.02	5.52	6.13	15.10	11.24
14	6.28	14.40	17.80	11.94	30.21	16.32	18.23	44.62	34.12

Table 3: Test 2.2, Algorithms 1-3 with $s = 4$ and $t = 3$

	Construct Time (s)			Solve Time (s)			Total Time (s)		
l	1	2	3	1	2	3	1	2	3
7	0.01	0.17	0.26	0.15	0.38	0.19	0.24	0.55	0.46
8	0.45	0.79	1.31	0.76	2.30	1.00	1.20	3.09	2.31
9	2.08	4.90	6.16	3.90	9.42	5.00	5.98	14.32	11.16
10	9.53	28.50	30.72	19.84	91.26	43.57	29.37	119.76	74.28

Firstly, the algorithms are compared using the Büchi game where $s = 2$ and $t = 2$, thus the number of vertices in the earlier sections are relatively small and each section is double the size of the sum of the previous sections, thus R_i is always half the size of T_i.

As predicted, algorithm 1 is fastest with this example, and algorithm 2 is slowest. Algorithm 3, though not as fast as algorithm 1, performs better than algorithm 2.

Secondly, the algorithms are compared with the example where $s = 4$ and $t = 3$, thus there is a greater difference between each section and the sum of the size of the previous 3 sections. For this reason, there is a more significant difference between the performance of algorithm 1 and 2 when l is sufficiently large.

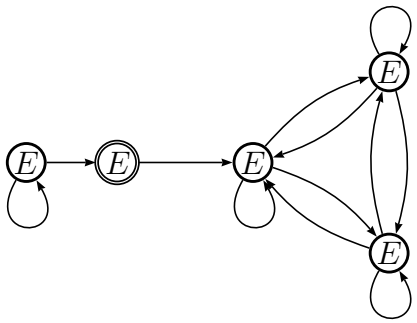
Notice, however, that the performance of algorithm 3 is significantly slower than algorithm 1. This is partly due to the construct time. For algorithm 3, the data structures for both algorithm 1 and 2 have to be initialised which can be expensive. Also, at the end of each iteration of the algorithm, the data structures from both algorithms have to be updated, regardless of which algorithm was used. This can significantly increase the amount of work required for each iteration.

In the case of the example above, due to the fact that the number of edges is approximately equal to the number of vertices and since a number of the data structures require at worst $O(n)$ work to update, the managing of data structures becomes a more significant factor in the solve time. This is in comparison to when the edge to vertex ratio is high, when the managing of data structures is much less significant in comparison to the attractor computations which work on the edges of the game graph.

In order to explore the advantages of algorithm 1 further, the following graph is considered:

- $\mathcal{G}_n := (G_n, v_0, F)$ where $G_n := (V_n, E_n, \lambda)$ and $n \in \mathbb{N}$
- $V_n := \{0, ..., n, n+1\}$
- $E_n := \{(0,0), (0,1), (1,2)\} \cup (\bigcup_{i=2}^{i=n+1} E_i)$ where:
  - $E_i := \{(i,2), (i,3), ..., (i,n), (i,n+1)\}$
- $\lambda(v) = \exists$ for all $v \in V_n$
- $F := \{1\}$

As an example, $\mathcal{G}_3$ is as follows:



Thus $\mathcal{G}_n$ is a graph of size $n+2$ where the successor vertices for each of the last n vertices is the set of the last n vertices. Since the only accepting vertex is vertex 1, $T_0 := \{2, 3, ..., n+1\}$ and thus $R_0 := \{0, 1\}$. It is clear that the algorithms terminate on the second iteration.

Also, since the number of incoming edges in T_0 equals $O(n^2)$ and the number of incoming edges in R_0 equals 2, algorithm 1 should perform better than algorithm 2. This is because in order to compute T_0, the attractor computations in algorithm 2 will work on at least the incoming edges in T_0, whereas the attractor computation of algorithm 1 will work on those in R_0.

Note, however, that in order to proceed to the final iteration of algorithm 1, the incoming edges in T_0 will still have to be inspected in order to calculate W_0 and in order to remove W_0 from the game graph. Hence, although the performance of algorithm 1 should be better, the first iteration will still take $O(m)$ time.

In contrast to test 2, this example has a fixed number of iterations (exactly 2) and thus relies on algorithm 1 performing significantly better than algorithm 2 in the first iteration. Instead of just exploring the relative size differences between the sets R_i and T_i, this test demonstrates how the size difference between the set of incoming edges for T_i and R_i affects the performance of the algorithms.

The results are displayed in table 4.

Table 4: Test 3, Algorithms 1-3

	Construct Time (s)			Solve Time (s)			Total Time (s)		
n	1	2	3	1	2	3	1	2	3
1000	0.28	0.94	0.87	0.53	1.66	0.53	0.81	2.60	1.40
3000	2.03	8.24	8.83	4.85	13.59	4.88	6.87	21.83	13.71
4000	3.58	16.04	16.37	8.74	26.75	8.64	12.33	42.79	25.01
5000	5.92	33.24	33.43	13.78	45.14	13.87	19.70	78.38	47.30
6000	8.51	45.22	47.09	20.04	65.11	19.76	28.55	110.33	66.86

As predicted, algorithm 1 performs significantly better than algorithm 2 and algorithm 3 performs worse than algorithm 1 but better than algorithm 2.

The construct time for the second algorithm is significantly worse than that for the first algorithm. This is because, when initialising data structures for algorithm 1, only one data structure (the count of outgoing edges) involves $O(m)$ work. All the others take $O(n)$ time. For algorithm 2, however, the initialisation of the array of successor vertices and the computation of $(C_1^0 \cup C_2^0)$ also require $O(m)$ time. Thus, since $m = O(n^2)$ and since there are only two iterations, the construct time for algorithm 2 forms a large proportion of the total time.

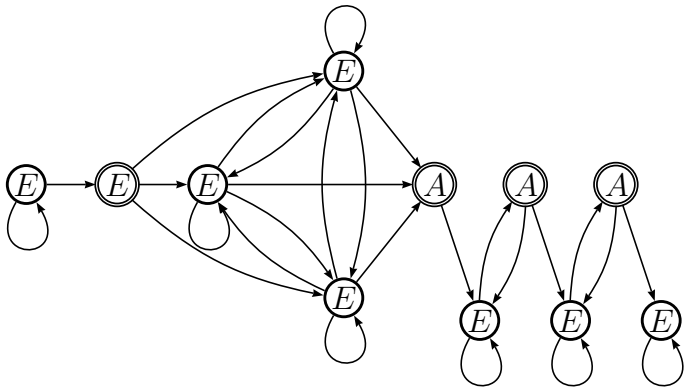
Regardless of the construct time, however, the solve time for algorithm 1 is also significantly better than algorithm 2.

The performance for algorithm 3 is severely restricted by the construct time required for algorithm 2. Notice, however, that the actual solve time of algorithm 3 is comparable to that of algorithm 1. This is in contrast to test 2 where the solve time was worse. As previously explained, this is due to the fact that the edge to vertex ratio is high, thus the attractor computations are the most significant factor of the solve time.

In order to explore instances where algorithm 3 performs better than either of the other two algorithms, consider the following set of Büchi games:

- $\mathcal{G}_{k,l} := (G_{k,l}, v_0, F)$ where $G_{k,l} := (V_{k,l}, E_{k,l}, \lambda)$ and $k, l \in \mathbb{N}$
- $V_{k,l} := \{k_0, k_1, ..., k_{k-1}, k_k, k_{k+1}\} \cup \{l_{0,A}, l_{0,E}, l_{1,A}, l_{1,E}, ..., l_{l-1,A}, l_{l-1,E}\}$
- $E_{k,l} := E_k \cup E_l$ where:
  - $E_k := \{(k_0, k_0), (k_0, k_1)\} \cup \{(k_1, k_2), (k_1, k_3), ..., (k_1, k_{k+1})\}$
    $\cup\{(k_2, l_{0,A}), (k_3, l_{0,A}), ..., (k_{n+1}, l_{0,A})\}$
    $\cup\ (\bigcup_{i=2}^{i=k+1} E_k^i)$ where:
    - $E_k^i := \{(k_i, k_2), (k_i, k_3), ...(k_i, k_k), (k_i, k_k + 1)\}$
  - $E_l := \bigcup_{i=0}^{l-1} E_l^i$ where:
    - $E_l^0 := \{(l_{0,A}, l_{0,E}), (l_{0,E}, l_{0,E}), (l_{0,E}, l_{1,A})\}$
    - $E_l^{l-1} := \{(l_{l-1,A}, l_{l-2,E}), (l_{l-1,A}, l_{l-1,E}), (l_{l-1,E}, l_{l-1,E})\}$
    - $E_l^i := \{(l_{i,A}, l_{i-1,E}), (l_{i,A}, l_{i,E}), (l_{i,E}, l_{i,E}), (l_{i,E}, l_{i+1,A})\}$
      for $0 < i < l - 1$
- $\lambda(v) := \forall$ if $v \in \{l_{0,A}, l_{1,A}, ...l_{l-1,A}\}$ else $\lambda(v) := \exists$
- $F := \{k_1\} \cup \{l_{0,A}, l_{1,A}, ..., l_{l-1,A}\}$

In other words, the Büchi game $\mathcal{G}_{k,l}$ is equivalent to the game used in test 1 with some extra vertices added. For example, $\mathcal{G}_{3,3}$ would be as follows:



Thus for the first l iterations, algorithm 2 is preferable since $X_i = \{l_{l-i+1,E}, l_{l-i+1,A}\}$ whereas $R_i = V_{k,l} \backslash \{l_{l-i+1,A}\}$. At iteration $l+1$, however, $X_{l+1} = \{k_1, k_2, ..., k_k, k_{k+1}\}$, whereas $R_{l+1} = \{k_0, k_1\}$. Thus for this iteration, algorithm 1 is preferable. Hence, for large enough k and l, algorithm 3 should be faster than both of the previous algorithms. The results are shown in table 5.

Table 5: Test 4, Algorithms 1-3

		Construct Time (s)			Solve Time (s)			Total Time (s)		
k	l	1	2	3	1	2	3	1	2	3
10	5000	0.04	0.09	0.10	57.03	0.41	0.56	57.07	0.50	0.66
1000	6	0.28	0.94	0.86	2.89	1.66	0.80	3.17	2.60	1.66
1000	10	0.27	0.88	0.89	4.35	1.66	0.80	4.63	2.54	1.69
1000	50	0.27	0.92	0.89	19.00	1.66	0.80	19.27	2.58	1.69
2000	6	0.96	3.46	3.42	11.61	6.77	3.16	12.57	10.23	6.58
2000	10	0.96	4.13	3.96	17.38	6.77	3.16	18.34	10.90	7.12
4000	10	3.72	16.32	18.20	69.90	31.00	12.73	73.61	47.32	30.93
5000	2	5.80	29.86	30.69	36.71	51.38	20.13	42.51	81.24	50.82
5000	6	5.70	29.58	31.97	73.15	51.41	19.93	78.85	81.99	51.90

As predicted, algorithm 3 performs best for large enough k and l. Notice that for large k, l need not be very large before algorithm 3 outperforms algorithm 1.

Algorithm 2 generally performs better than algorithm 1. This is because algorithm 1 only performs better on the last iteration and the size of k negatively affects the performance of this algorithm on all the other iterations. For this reason, the size of l (and therefore the number of iterations) need not be large before algorithm 2 is better than algorithm 1.

Finally, in order to investigate the general performance of the algorithms, they will be tested on random games implemented by [6]. In [6], a random parity game is generated given the number of vertices required, the maximum and minimum outdegree and the maximum priority. For each vertex, the priority is chosen with uniform probability from the priority range and the outdegree is chosen with uniform probability from the outdegree range. The successor vertices are then selected randomly from the set of all vertices. Since we wish to generate random Büchi games, the maximum priority will always be fixed at 1. The results are as follows:

Table 6: Test 5: Random games, algorithms 1-3

			Construct Time (s)			Solve Time (s)			Total Time (s)		
n	min out	max out	1	2	3	1	2	3	1	2	3
10000	2	100	0.27	0.65	0.65	0.69	0.77	1.14	0.96	1.42	1.83
10000	2	1000	2.30	7.35	8.18	7.49	7.79	11.80	9.79	15.14	19.98
50000	2	10	0.26	0.66	0.62	0.83	0.92	1.37	1.09	1.58	1.99
50000	2	100	1.53	4.77	4.60	4.73	5.12	7.41	6.26	9.89	12.01
50000	2	1000	13.82	52.08	67.38	45.70	50.85	72.74	59.52	102.93	140.12
100000	2	10	0.52	1.26	1.38	1.51	1.83	3.48	2.03	3.09	4.86
100000	2	100	3.27	9.86	11.01	9.81	10.86	15.66	13.08	20.72	26.67

Algorithm 1 consistently performs better than the other two with algorithm 3 performing worst. Note, however, that all of these games only required 1 iteration to solve, thus making the initialising of data structures at the start of each algorithm a more significant factor in the total solve time. This increases the likelihood of algorithm 1 performing best since it performs the least amount of work when initialising data structures. In fact, without taking the initialising of data structures into account, algorithm 1 and 2 performed similarly. This caused algorithm 3 to perform worst since algorithm 3 is only advantageous when at each iteration, one of the first two algorithms is noticeably faster than the other.

The fact that all of the random games generated only required 1 iteration to solve suggests that they possibly all have a similar simple structure, despite their random nature. Thus, they are not necessarily indicative of general Büchi games. In order to attempt to address this issue, the algorithms will now be compared using clustered random games as implemented in [6]. These are generated given the number of vertices n, the highest priority p, the out-degree range (l, h), the recursion depth r, the recursion breadth range (a, b) and the interconnection range (x, y). A game $\mathcal{G}^c_{(n,r)}$ where $c = (p, l, h, a, b, x, y)$ is then generated as follows:

- If $r = 0$ then a random game with n vertices, maximum priority p and out-degree range (l, h) is constructed.
- If $r > 0$ a number d is chosen randomly from the range $\{a, ..., min(b, n)\}$
- The numbers $k_0, ...k_{d-1}$ are randomly chosen from $\{0, ..., n-1\}$ such that $\sum_{i=0}^{d-1} k_i = n$
- Clustered random games $\mathcal{G}^c_{(k_i,r-1)}$ are constructed for $0 \leq i < d$.
- The graph $G = \bigcup_{i=0}^{d-1} \mathcal{G}^c_{(k_i,r-1)}$ is calculated.

- e randomly chosen edges, where e is chosen randomly from $\{x, x+1, ..., y\}$, are added to G and G is returned.

The idea is to create clusters in the game graph and then connect them together using the randomly chosen interconnection edges. This should then provide more structure to the graph. Choosing the most effective parameters for this kind of graph in order to create interesting structures can be difficult. In particular, it is difficult to pick the most effective interconnection range. If the number of interconnection edges is too small then many of the clusters will not be connected, resulting in a graph composed of disconnected clusters. If the number of interconnection edges is too high then the clusters will be connected via a large number of edges, effectively resulting in one large cluster. This will thus result in a structure similar to the non-clustered random game graphs.

From experimenting with varying the parameters, due to the random nature of the interconnection edges, it is difficult to generate a clustered random graph that requires more than 3 iterations to solve. The most effective recursion depth and breadth range found was $r = 3$ and $(l, h) = (3, 4)$, with any greater recursion depth or breadth range making little difference on the number of iterations or the solve time. Similarly, it was found that an interconnection range of $(x, y) = (50, 100)$ was most effective, consistently producing a game that required 3 iterations to solve. Table 7 gives the results of experiments on clustered random game graphs using these parameters, where the number of vertices and the maximum outdegree are varied.

Table 7: Test 6: Clustered random games with recursion depth 3, algorithms 1-3

			Construct Time (s)			Solve Time (s)			Total Time (s)		
n	min out	max out	1	2	3	1	2	3	1	2	3
50000	2	10	0.26	0.56	0.56	0.72	0.82	1.29	0.98	1.38	1.85
50000	2	10	0.21	0.50	0.58	0.73	0.81	1.18	0.94	1.31	1.76
50000	2	10	0.23	0.50	0.51	0.66	0.71	1.06	0.89	1.21	1.57
100000	2	10	0.42	1.01	1.05	1.46	1.60	2.51	1.88	2.61	3.56
100000	2	10	0.46	1.05	1.41	1.48	1.59	2.54	1.94	2.64	3.95
100000	2	10	0.38	0.96	1.05	1.40	1.57	2.47	1.78	2.53	3.52
100000	2	50	1.03	3.37	3.24	4.41	3.89	6.16	5.44	7.26	9.40
100000	2	50	1.08	3.22	3.29	3.57	3.83	5.92	4.65	7.05	9.21
100000	2	50	0.98	3.09	3.17	4.14	3.84	5.87	5.12	6.93	9.04
500000	2	30	4.29	13.69	20.21	18.85	17.89	25.70	23.14	31.58	45.91
500000	2	30	4.51	13.94	20.56	18.24	17.25	25.97	22.75	31.19	46.53
500000	2	30	4.50	13.90	14.24	17.19	16.19	24.68	21.69	30.09	38.92

Similarly to the random games, algorithm 1 consistently performs best. In fact, comparing with random games that have the same number of vertices

and equal maximum outdegree, the performance times are very similar. It is also notable that producing several clustered random games with the same parameters results in similar performance times. The better performance of algorithm 1 is again mainly due to the relatively small number of iterations required to solve each game, resulting in a high proportion of the total solve time comprising of the initialising of data structures. The small number of iterations required to solve clustered random games suggests that, in general, Büchi games do not require many iterations to solve. It is possible, however, that clustered random games also do not produce enough variety of structure to indicate the general performance of each solver on Büchi games.

Using the examples given above, the following has been demonstrated:

- A small number of iterations required to solve a game restricts the performance of algorithms 2 and 3 due to a higher proportion of the total time comprising of construct time.
- A large edge to vertex ratio restricts the performance of algorithm 2 due to a large construct time. This, therefore, also affects algorithm 3.
- A small edge to vertex ratio can make the management of data structures in algorithm 3 a more significant time factor. This can cause the difference between algorithm 3 and the better performing algorithm at each iteration to be more significant.
- For each algorithm, there exist examples where that algorithm performs best.

6.2 Testing Against Parity Solvers

The performance of the algorithms implemented will now be compared to the performance of three implemented parity game solvers given in [6]. These parity game algorithms are The Recursive Algorithm [18], The Small Progress Measures Algorithm [7] and The Strategy Improvement Algorithm [16] which respectively have time complexities $O(mn^d)$, $O(dm(n/d)^{d/2})$ and $O(2^m mn)$ where n is the number of vertices, m is the number of edges and d is the number of priorities. Thus when restricted to Büchi games they have time complexities $O(mn^2)$, $O(mn)$ and $O(2^m mn)$.

As has been previously mentioned, the implementation of these algorithms, as well as returning the partition of Abelard and Eloïse winning vertices, also return a corresponding winning strategy for Abelard and Eloïse. Thus, in this section, when comparing to parity game solvers, an implementation of algorithms 1-3 will be used that also returns a winning strategy as described in Section 4.

In [6], some universal optimisations are also made while solving a game. These optimisations include strongly connected component (SCC) decomposition, detection of special cases and priority compression, the first two of which are relevant to Büchi games.

Given a graph G, an SCC is a set S of vertices such that every vertex in S is reachable from every other vertex in S. Note that this simply requires that from each vertex in S there is a sequence of edges to any other vertex in S and does not relate to the owner of each vertex in a game graph. Any graph G can be partitioned into SCCs, and in particular there exists at least one final SCC. An SCC is final if no vertex from outside the SCC is reachable from any vertex within the SCC. Thus, when solving a Büchi game, if the corresponding game graph is partitioned into SCCs, the vertices in a final SCC can be solved independently of the rest of the game graph. Once the vertices in a final SCC have been solved, the Abelard and Eloïse attractors of the resulting Abelard and Eloïse winning partitions can be computed. These attractors can be removed from the game graph as vertices that have already been solved. Those SCCs from which vertices were removed may no longer be SCCs and thus are themselves partitioned into SCCs. Again, the game is solved on the final SCCs and this continues until the entire game is solved.

The advantage of using SCC decomposition is that it allows the graph to be solved in small sections potentially reducing the number of times a vertex and its edges need be inspected. The best algorithms for performing this decomposition have time complexity $O(m)$, making the decomposition of a graph into SCCs a worthwhile procedure in the context of solving parity games.

In addition to using SCC decomposition, the following are types of special cases that the implementation in [6] searches for:

- Self-cycle games: This term refers to games where there exists a vertex that has an edge to itself. In this case, if the vertex is owned by Eloïse and is accepting then clearly this vertex is winning for Eloïse. Thus, taking the Eloïse attractor of this vertex solves a subset of the game graph. Similarly, the Abelard attractor is found if the self-cycle vertex is owned by Abelard and is not accepting. If a vertex is owned by Eloïse and is not accepting then the edge from the vertex to itself can be removed from the game graph providing that the vertex has at least one other outgoing edge. Similarly, an edge can be removed when the vertex is owned by Abelard and is accepting.

- One-parity games: If all the vertices are accepting or all the vertices are not accepting then the winning sets can be trivially determined.

- One-player games: If an SCC is composed of vertices that are all owned by Eloïse then, providing one of the vertices is accepting, all vertices

in the SCC are winning for Eloïse. Similarly if all vertices are owned by Abelard and there exists a cycle of non-accepting vertices then all vertices in the SCC are winning for Abelard.

Thus using these optimisations, given a game solver, a Büchi game can be solved as follows:

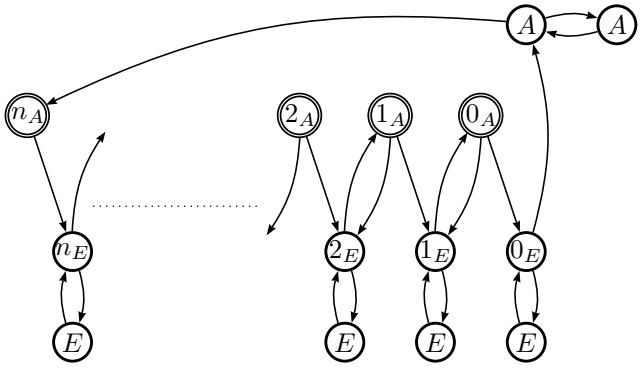
1. First check for self-cycles and deal with them accordingly.
2. Decompose the game graph into SCCs.
3. For each final SCC:
   - If it is a special case, solve it as described above.
   - If it is not then use the given solver.
4. Find the corresponding attractor for the computed winning sets found for each final SCC and add these to the winning sets.
5. Remove these winning sets from the game graph.
6. Any remaining SCC which has lost some vertices is decomposed further into SCCs.
7. Return to step 3

To demonstrate the use of these optimisations, recall the example given in test 1 of Section 6.1. This can be solved entirely using universal optimisations:

- In step 1, the self-cycles of vertices in the set $\{1_E, 2_E, ..., n_E\}$ are eliminated.
- The SCCs are computed to be the sets $\{0_E\}$, $\{n_A\}$, and $\{i_E, (i-1)_A\}$ for each $1 \leq i < n$.
- The final SCC $\{0_E\}$ is identified to be the special case of a one parity game and is identified to be winning for Abelard.
- With the earlier self cycles removed, the Abelard attractor set of $\{0_E\}$ is found to be the entire graph, thus any vertex in the graph is winning for Abelard.

Since this Büchi game can be solved entirely using universal optimisations, it could be argued that it is not of interest to compare the efficiency of solvers on this game. It is possible, however, to modify the game slightly so as to render these universal optimisations ineffective. This is achieved by adding an extra self-cycle Abelard vertex connecting 0_E and n_A to make the graph one

single SCC. Also, all self-cycle edges are replaced with a two vertex cycle. This would result in the following graph:



Although this game is different to the original one, since it is of a similar structure and preserves the same winning sets for the original vertices, the behaviour for most algorithms on this new game, provided it does not use universal optimisations, should be comparable to that on the original game. Thus the original game is still useful for comparing different algorithms. This is similarly the case for other examples used previously. Hence, for the purposes of comparing the Büchi game solvers with the parity game solvers, these universal optimisations will be switched off in order to give a fair comparison.

Table 8 shows the results of comparing the parity game solvers with algorithms 1 and 2 on the example used in test 1 of Section 6.1.

Table 8: Test1 with parity game solvers and algorithms 1 and 2

	Total Time (s)				
n	Recursive	Small Progress Measures	Strategy Improvement	Algorithm 1	Algorithm 2
1000	13.24	9.42	3.03	2.17	0.12
2000	94.68	66.60	14.16	8.95	0.23
3000	-	86.35	34.45	20.26	0.32
4000	-	159.80	66.96	36.79	0.34
5000	-	253.90	113.71	58.26	0.49
6000	-	-	172.53	84.79	0.69

Clearly both algorithm 1 and 2 perform significantly better than any of the parity game solvers with the recursive algorithm performing particularly badly. As was demonstrated in the previous section, algorithm 2 is the best performing of the three Büchi game algorithms on this example and algorithm 1 is the

worst performing. Thus the parity game solvers are slower than all three Büchi game algorithms.

Table 9 compares algorithms 1 and 3 with the parity game solvers using the example from test 4 of Section 6.1. Algorithms 1 and 3 are used for comparison since they are respectively the generally worst performing and best performing of the three algorithms on this example.

Table 9: Test 4 with parity game solvers and algorithms 1 and 3

		Total Time (s)				
k	l	Recursive	Small Progress Measures	Strategy Improvement	Algorithm 1	Algorithm 3
10	2000	101.12	38.39	14.56	9.82	0.21
10	5000	-	252.35	114.24	58.25	0.75
500	6	1.89	2.89	91.59	0.90	0.52
500	10	2.94	4.20	92.00	1.27	0.53
500	50	13.49	17.81	104.38	5.07	0.53
500	100	29.93	38.09	119.58	9.81	0.55
1000	6	8.02	14.52	-	3.43	1.93
1000	10	12.77	21.31	-	5.00	1.89
1000	50	-	90.14	-	20.09	1.85

Again, both algorithm 1 and 3 perform consistently better than any of the parity game solvers. In this case, the strategy improvement algorithm is particularly slow when k is large but performs better than both of the other two parity game solvers when l is large. The recursive algorithm performs especially badly when l is large.

Finally, algorithm 1 is compared with the parity game solvers using clustered random games with the same parameters as those used in test 6 of Section 6.1. The results are shown in table 10.

Again, excepting one game when $n = 50000$, algorithm 1 is consistently better performing than any of the parity game solvers. The strategy improvement algorithm is generally very slow for these Büchi games. Interestingly, however, the solve time for this algorithm can vary dramatically between different clustered random games with the same parameters. This is also true to less of an extent for the small progress measures algorithm. For the recursive algorithm and algorithm 1, however, the solve times are fairly consistent between games with the same parameters.

Table 10: Clustered random games with parity game solvers and algorithm 1

			Total Time (s)			
n	min out	max out	Recursive	Small Progress Measures	Strategy Improvement	Algorithm 1
50000	2	10	3.37	6.82	234.38	1.93
50000	2	10	1.04	1.45	12.67	2.01
50000	2	10	1.97	5.65	142.58	1.77
100000	2	10	5.52	12.53	-	2.27
100000	2	10	4.30	13.21	-	2.04
100000	2	10	2.81	3.18	7.63	2.05
100000	2	50	12.37	21.87	-	5.11
100000	2	50	12.21	23.95	-	5.43
100000	2	50	12.92	24.77	-	6.06
100000	2	100	19.89	38.52	-	9.26
100000	2	100	19.61	20.67	22.09	9.26
100000	2	100	21.00	37.77	-	10.80
500000	2	10	21.33	23.68	98.08	12.10
500000	2	10	28.78	82.03	-	13.24
500000	2	10	22.15	22.33	29.91	13.87
500000	2	30	53.60	105.89	-	21.33
500000	2	30	63.70	105.73	-	21.01
500000	2	30	64.10	66.44	-	23.30

A range of different examples from Section 6.1 have been used which allowed for comparison of the different types of solver on a range of different types of Büchi game. The tests performed to compare parity game solvers with Büchi game solvers implemented in this dissertation suggest that the latter are generally faster. Thus, for parity games which are also Büchi games, it would be preferable to use these Büchi game solvers.

7 Conclusion

In this dissertation three algorithms for solving Büchi games were implemented. The main idea of all three algorithms is to identify winning regions of the game graph for Abelard through a series of iterations. At each iteration, the winning region found for Abelard is removed. Where the algorithms differ is their approach to finding such winning regions. The first identifies all potential Eloïse winning vertices and takes the complement of this. The disadvantage of such an approach is that at each iteration, potentially a large number of edges are worked are which are not then removed from the game graph. In order to improve on this approach the second algorithm from [2], reduces the number of edges which are examined but not removed, by considering a carefully chosen subset of the game graph which is known to contain the desired Abelard winning region. The third algorithm then dovetails the previous two algorithms.

In Section 5, the Büchi game solvers were integrated into the model checker THORS. This is believed to be the first implementation of an algorithm for solving the acceptance problem for HORS against tree automata with a non-trivial acceptance condition (a trivial Büchi tree automaton only requires there to *exist* a valid run-tree for a HORS in order for it to be accepted). The integration of a Büchi game solver into THORS allows it to solve the acceptance problem for all Büchi tree automata as opposed to just weak ones. Thus the fragment of the modal μ calculus which can be used in the corresponding model checking problem is expanded and is therefore more expressive.

In Section 6, the implementation of all three algorithms were compared. It was confirmed in test 1 that in the example introduced in [2], algorithms 2 and 3 performed significantly better than that of algorithm 1. It was also demonstrated that there exist examples where each algorithm performed best. In the case of the random game graphs, however, algorithm 1 was found to be consistently faster than the other two algorithms. This suggests that algorithms 2 and 3 are only faster for very specific examples. In particular, it was found that the initialisation of data structures for algorithms 2 and 3 significantly impaired their performance. Generally, for examples requiring a small number of iterations to solve therefore, it is unlikely that algorithm 2 or 3 would perform better than algorithm 1. The fact that the random game graphs (clustered or not) never required many iterations to solve may suggest that generally, game graphs are solved in a small number of iterations. This indicates that in practice, algorithm 1 would be preferred over the other two examples.

When comparing the three Büchi game solvers to a selection of implemented parity game solvers, it was found that the Büchi game solvers performed consistently better. Thus it was demonstrated that it is worth restricting a solver

to solving only Büchi games. This is particularly notable since solving Büchi games (and therefore co-Büchi games), allows for model checking using the expressive logic CTL*.

7.1 Possible Extensions

As was commented on in Section 6, in the implementation of parity game solvers from [6], a number of universal optimisations were used. This divided a game graph into SCCs, allowing the game graph to be solved on small subsections. Also, special cases of parity game were detected and solved separately. This use of universal optimisations can significantly decrease the solve time for a game, and in particular it was demonstrated in Section 6 that examples like those used in test 1 can be solved entirely using these optimisations. In addition to this, their use increases the chance of only a small number of iterations being required for a Büchi game solver. This therefore further highlights the advantages of using algorithm 1 over the other two algorithms. An interesting extension to this dissertation, therefore, would be to implement these universal optimisations for the Büchi game solvers in order to investigate how the performance of all three algorithms would be affected.

In addition to this, it may also be interesting to investigate other approaches of generating random graphs. This would allow for a more complete comparison of the different algorithms. For example, in the case of the random graph generators used in Section 6, a vertex is always labelled as accepting or not accepting with equal probability. It may be worth determining how the performance of the different algorithms varies when this probability is varied. Also, in the case of the clustered random graphs, the interconnection edges are added randomly in such a way that they may not connect separate clusters. This sometimes results in a graph which is split into several disconnected components. In order to increase the chances of separate clusters being connected, a larger number of interconnection edges can be used, although this can often result in clusters being indistinct from one another. In order to address this, it may be worth investigating the result of adding interconnection edges in a more prescribed manner, for example, by always connecting interconnection edges to vertices in different clusters. A more involved approach could be to specify how many interconnection edges each cluster should have. This many vertices from each cluster could then be randomly chosen and connected to a randomly chosen vertex in a different cluster. This way, it is known that all the clusters are always connected to at least one other cluster, and thus the chances of the graph being disconnected are much lower.

It may also be worth investigating what structure, games constructed from modal μ-calculus formulas generally produce. This would allow for a more informed decision about which Büchi game solver is preferable when considering

the model checking problem. Given the experiments in Section 6, however, it seems likely that algorithm 1 would perform best.

References

[1] J. Bradfield and C. Stirling. Modal logics and mu-calculi: An introduction, 2001.

[2] K. Chatterjee, T. A. Henzinger, and N. Piterman. Algorithms for Büchi games. In *GDV*, 2006.

[3] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. In *Theoret. Comput. Sci. volume 126*, pages 77–96, 1994.

[4] E. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, FoCS '91, IEEE Computer Society Press*, pages 368–377, 1991.

[5] E. Emerson and A. P. Sistla. On model-checking for fragments of μ-calculus. In *CAV'93, volume 697 of LNCS*, pages 385–396. Springer-Verlag, 1995.

[6] O. Friedmann and M. Lange. Pgsolver. `http://www2.tcs.ifi.lmu.de/pgsolver`.

[7] M. Jurdziński. Small progress measures for solving parity games. In *volume 1770 of LNCS*, pages 290–301. Springer, 2000.

[8] N. Kobayashi. Model-checking higher-order functions. In *PPDP 2009: Proceedings of the 11th ACM SIGPLAN conference on Principles and practive of declarative programming, ACM*, pages 25–36, 2009.

[9] N. Kobayashi and C.-H. Ong. A type theory equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009, IEEE Computer Society*, 2009.

[10] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[11] O. Kupferman and M. Vardi. Freedom, weakness and determinism: From linear-time to branching-time. In *LICS*, pages 81–92, 1998.

[12] R. McNaughton. Infinite games played on finite graphs. In *volume 65, issue 2 of Ann. Pure Appl. Logic*, pages 149–184, 1993.

[13] D. Niwinski. On fixed-point clones. In *Automata, Languages and Programming: 13th International Colloquium, volume 226 of LNCS*, pages 464–473. Springer-Verlag, 1986.

[14] S. Ramsay, M. Lester, R. Neatherway, and C.-H. Ong. Model checking non-trivial properties of higher-order functional programs [pre-print]. 2010.

[15] S. Schewe. Solving parity games in big steps . In *volume 4855 of LNCS*, pages 449–460. Springer-Verlag, 2007.

[16] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV'00 volume 1855 of LNCS*, pages 202–215. Springer, 2000.

[17] T. Wilke. Alternating tree automata, parity games, and modal mu-calculus. In *volume 8, number 2 of Bull. Belg. Math. Soc*, 2001.

[18] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. In *volume 100(1-2) of TCS*, pages 135–183, 1998.