Symbolic Parity Game Solvers that Yield Winning Strategies

Oebele Lijzenga
Formal Methods and Tools
University of Twente, Enschede
o.r.lijzenga@student.utwente.nl

Tom van Dijk
Formal Methods and Tools
University of Twente, Enschede
t.vandijk@utwente.nl

Parity games play an important role for LTL synthesis as evidenced by recent breakthroughs on LTL synthesis, which rely in part on parity game solving. Yet state space explosion remains a major issue if we want to scale to larger systems or specifications. In order to combat this problem, we need to investigate symbolic methods such as BDDs, which have been successful in the past to tackle exponentially large systems. It is therefore essential to have symbolic parity game solving algorithms, operating using BDDs, that are fast and that can produce the winning strategies used to synthesize the controller in LTL synthesis.

Current symbolic parity game solving algorithms do not yield winning strategies. We now propose two symbolic algorithms that yield winning strategies, based on two recently proposed fixpoint algorithms. We implement the algorithms and empirically evaluate them using benchmarks obtained from SYNTCOMP 2020. Our conclusion is that the algorithms are competitive with or faster than an earlier symbolic implementation of Zielonka's recursive algorithm, while also providing the winning strategies.

1 Introduction

Parity games are turn-based games played by the players *Even* and *Odd* on a finite directional graph. All vertices are labelled by an integer priority. A play in a parity game is an infinite sequence of vertices consistent with the edge relation, where the owner of the current vertex (*Even* or *Odd*) determines the next vertex. The play eventually results in an infinitely repeating sequence of vertices and their priorities. If the highest priority in this sequence is even, then player *Even* wins, otherwise player *Odd* wins. Solving a parity game can either be to determine the winning area of the graph for a player, or determining a winning strategy for both players, or both. A strategy is winning for some player if it contains one move for each vertex controlled by, and in the winning area of that player, and all moves consistent with that strategy always cause that player to win.

It is widely believed that a polynomial time solution exists for determining the winner of a parity game, as they lie in the intersection of UP and co-UP [20], which is contained in the intersection of NP and co-NP. In the current paper, however, we are not interested in theoretical complexity but in practical performance. Parity games are closely related to many problems in formal verification and synthesis that can be reduced to the problem of solving parity games, as parity games capture the expressive power of nested least and greatest fixpoint operators. Parity games are especially relevant for LTL synthesis. In recent years, the fastest practical solver for LTL synthesis has been STRIX [26], winning SYNTCOMP [19] editions 2018 and 2019. STRIX converts an LTL formula to a deterministic parity automaton and then splits the automaton into a parity game. The solution of the parity game determines if the LTL formula is realizable and the winning strategy is used to construct a controller that is guaranteed to implement the LTL specification.

There are various classes of parity game solving algorithms. Broadly speaking, we identify four classes of algorithms. First, the algorithms that use repeated attractor computation to partition a game

J.-F. Raskin and D. Bresolin (Eds.): 11th International Symposium
on Games, Automata, Logics, and Formal Verification (GandALF'20).
EPTCS 326, 2020, pp. 18–32, doi:10.4204/EPTCS.326.2

into regions, which are subsequently refined until the game is solved. A particular algorithm in this category is Zielonka's recursive algorithm [35], where the partitioning is done recursively and subgames are fully solved before refining the top region. Recent work in this category includes novel algorithms like priority promotion [2] and tangle learning [10], but also a quasi-polynomial variation of the recursive algorithm [27]. Second, the class of progress measures algorithms [21], where each vertex in the game has a value which increases monotonically, based on the values of the direct successors of the vertex. These values are typically tuples of integers, and in recent work various authors have proposed sets of values that are quasi-polynomially bounded in size yet sufficient to solve parity games [7, 9]. Third, many different strategy improvement algorithms [33] have been studied, where one player iteratively improves its strategies by playing against the best response of the opponent. Fourth, several algorithms that do not fit in the above categories are formulated as a nested fixpoint in the modal μ-calculus. We call these simply the fixpoint iteration algorithms. These are the APT algorithm [32], a fixpoint iteration algorithm we refer to as BFL [4], as well as the recent DFI [13] and FPJ [25] algorithms.

Often formal verification and also synthesis suffers from the *state space explosion* problem. Indeed, [29] reports that parity game solvers can require over 8 GB memory for large specifications. One method to alleviate this problem is to use binary decision diagrams [6]. In the past, symbolic parity game solvers using binary decision diagrams have been explored [1, 23, 29, 32]. Symbolic algorithms replace explicit data structures with implicit, symbolic representations such as binary decision diagrams, relying on optimised BDD implementations such as CUDD [31] and Sylvan [12]. Using binary decision diagrams can result in a massive difference in memory usage [29].

For applications such as LTL synthesis, but also for model checking, obtaining the winning strategies is essential. Controller synthesis requires the winning strategy. For model checking, the winning strategy is used to derive counterexamples. In the current literature, several symbolic implementations of parity game solving algorithms have been proposed. In [23], Kant et al. implement Zielonka's recursive algorithm symbolically. Their focus lies on generating the games and little attention is given to strategy derivation. Their online implementation in LTSmin [22] appears to be capable of generating the winning strategy, but this is not reported in the paper and the solver is embedded within the LTSmin model checker. A symbolic implementation of the quasi-polynomial "ordered progress measures" algorithm [17] has been proposed [8] which requires only quasi-polynomially many symbolic operations. In [32], Di Stasio et al. implement Zielonka's recursive algorithm, the fixpoint algorithm APT [24], and a symbolic small progress measures [21] algorithm. Winning strategies are not derived. Finally, Sanchez et al. [29] implement Zielonka's recursive algorithm, two fixpoint iteration algorithms, and priority promotion [2] symbolically, but without strategy derivation.

If we want to use symbolic methods in the entire LTL synthesis toolchain, from specification to controller, then we require fast symbolic parity game algorithms that produce the winning strategies.

In recent work, Van Dijk and Rubbens [13] and Lapauw et al. [25] propose fixpoint algorithms that yield strategies in a straightforward way. Previous fixpoint algorithms either do not produce a winning strategy [24] or require a secondary, complicated algorithm [4]. Fixpoint iteration algorithms have the advantage that the data structures that they maintain are very simple. For various algorithms like strategy improvement and small progress measures, relatively complex and refined labelings are used, whereas fixpoint iteration algorithms maintain very little information per vertex: membership in the distraction set is sufficient, and in addition the algorithms we study also store the strategy (a subset of the edge relation) and either the so-called justification (a subset of the edge relation), or membership in one of the d so-called frozen sets, where d is the highest priority in the parity game.

The contributions of this paper are as follows. We **propose** and **implement** two novel symbolic parity game algorithms based on [13] and [25]. We **evaluate** these algorithms empirically using benchmarks

from the 2020 edition of SYNTCOMP [19]. We find that the algorithms are competitive compared to state-of-the-art algorithms, while also producing winning strategies.

2 Preliminaries

2.1 Parity games

We define a parity game PG as a tuple $(V, V_\Diamond, V_\Box, E, \mathsf{pr})$, where V is a set of vertices partitioned into $V_\Diamond$ and $V_\Box$; vertices in $V_\Diamond$ are controlled by player Even and vertices in $V_\Box$ are controlled by player Odd. The mapping $E \subseteq V \times V$ describes which moves can be made from each vertex in V; each vertex must have at least one successor. The function $\mathsf{pr}\colon V \to \{0, 1, ..., d\}$ assigns a priority to each vertex, where d is the highest priority in the game.

We introduce some additional notation. We write $E(u)$ for all successors of u, and $u \to v$ if $v \in E(u)$. We write player $\alpha \in \{\Diamond, \Box\}$ when referring to one of the players and player $\overline{\alpha}$ to refer to the other player. We use V_0 for vertices with an even priority and V_1 for vertices with an odd priority. We extend this notation for any set $X \subseteq V$ such that $X_0 := X \cap V_0$, $X_\alpha := X \cap V_\alpha$, etc. Finally, we use the following notations from μ-calculus [34] to denote predecessors with *some* or *all* successors in the set X:

$$\begin{array}{lcl} \Diamond X & := & \{v \in V \mid \exists u\colon v \to u \wedge u \in X\} \\ \Box X & := & \{v \in V \mid \forall u\colon v \to u \Rightarrow u \in X\} \end{array}$$

With $\Diamond X$ we mean all vertices with a successor in X. This is also called the *preimage* of X. With $\Box X$ we mean the vertices where *all* successors are in X, i.e., vertices with no edges to vertices outside X.

A *play* $\pi = v_0 v_1 \ldots$ is an infinite sequence of vertices consistent with E, i.e., $v_i \to v_{i+1}$ for all successive vertices. We denote with $\inf(\pi)$ the vertices that occur infinitely often in π. Player Even wins a play π if the highest priority in $\inf(\pi)$ is even; player Odd if the highest priority in $\inf(\pi)$ is odd.

A (positional) *strategy* $\sigma\colon V \to V$ assigns to each vertex in its domain a single successor in E, i.e., $\sigma \subseteq E$. We refer to a strategy of a player α to restrict the domain of σ to V_α. In the remainder, all strategies σ are of a player α. We write $\mathrm{Plays}(v)$ for the set of plays starting at vertex v and $\mathrm{Plays}(v, \sigma)$ for all plays from v consistent with σ.

A basic result for parity games is that they are memoryless determined [16], i.e., each vertex is either winning for player Even or for player Odd, and both players have a positional strategy for their winning vertices. Player α wins a vertex v if there exists a strategy σ of player α such that every $\pi \in \mathrm{Plays}(v, \sigma)$ is winning for player α.

Example 1. *Figure 1 is an example of a simple parity game, consisting of 9 vertices and 15 edges. The diamond-shaped vertices are in $V_\Diamond$ and the square-shaped vertices are in $V_\Box$. Player Odd wins all vertices of the parity game with the strategy $\{\boldsymbol{b} \to \boldsymbol{f}, \boldsymbol{d} \to \boldsymbol{e}\}$.*

2.2 Fixpoint iteration algorithms for parity games

Many different types of algorithms have been proposed to solve parity games. In this paper, we focus on fixpoint iteration algorithms. Fixpoint iteration algorithms iteratively refine an estimation of which vertices are won by which player. A typical initial estimation is that player Even wins all even-priority vertices and player Odd wins all odd-priority vertices. This estimation is then iteratively refined by considering the direct successors of each vertex. By updating the estimation in a strict order, starting with the lowest-priority vertices, and resetting the estimation of lower-priority vertices whenever a higher-priority vertex is updated, the solution of parity games can be computed [13].

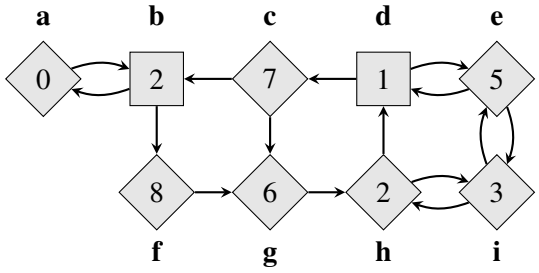

Figure 1: A simple parity game, consisting of 9 vertices and 15 edges.

We consider two fixpoint iteration algorithms proposed in recent literature. The `DFI` algorithm proposed by Van Dijk and Rubbens [13] is a fixpoint iteration algorithm based on the concept of *distractions*. The freezing extension of `DFI` enables straightforward strategy computation. When updating vertices of some priority, all lower-priority vertices that are won by the winner are frozen, i.e., they will no longer be reevaluated. Only vertices that are not frozen are reevaluated. The effect is that freezing preserves the winning strategy. The `FPJ` algorithm proposed by Lapauw et al. [25] modifies the standard fixpoint iteration by maintaining a *justification graph*, which essentially records which vertices are currently "justified" and the strategy that witnesses the justification. Justified vertices are not reevaluated, but each time a vertex is updated by the algorithm, the justification graph is pruned by removing vertices that are no longer justified. It is straightforward to extract the winning strategy from the justification graph.

A distraction is some even-priority vertex v that can be won by player Odd if player Even always tries to reach v, and vice versa. We say that a distraction is *fatal* if player Odd wins vertex v regardless of the strategy of player Even, and that a distraction is *devious* if player Even wins vertex v using a strategy that at least sometimes avoids vertex v. They are called distractions because (many) parity game solvers will initially assume that these vertices are safe for player α to play to, but after some steps, the parity game solver will find that these vertices are actually unsafe for player α and have to recompute all vertices that relied on the unsafe vertex. For example in Figure 1, vertex **c** is a devious distraction. If player Odd plays from **d** to **c**, then player Even wins. By avoiding vertex **c**, player Odd wins the game.

Fixpoint algorithms offer a distinct advantage compared to other parity game algorithms, as they maintain uncomplicated evaluations of vertices. Earlier work, e.g. [29, 32], also considers fixpoint algorithms in their symbolic implementations. As demonstrated in [32], the symbolic implementation of the small progress measures algorithm often results in timeouts.

2.3 Binary decision diagrams

Binary decision diagrams [5, 14, 15] (BDDs) are a well known data structure for representing and manipulating Boolean functions. Figure 2 shows four examples of BDDs representing different Boolean functions. A binary decision diagram is a rooted directed acyclic graph, with internal nodes representing decisions over Boolean variables. Each internal node has two successors, one via the so-called "true" edge (depicted as a solid arrow) and one via the "false" edge (depicted as a dashed arrow). Given some valuation to Boolean variables, we either follow the "true" edge if the variable has the value true, or the "false" edge. If we arrive at the leaf node "1", then the represented Boolean function evaluates to true for the given valuation. Otherwise, the represented Boolean function evaluates to false. It is well known that BDDs are a canonical representation of Boolean functions if they are ordered, i.e., Boolean variables are

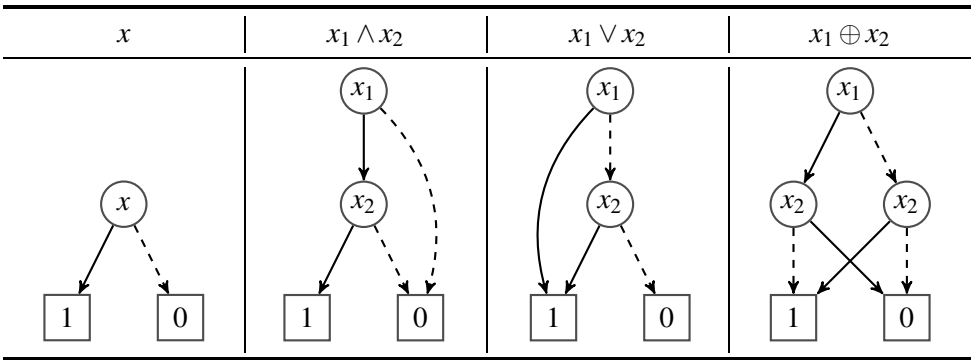

Figure 2: Four simple binary decision diagrams.

encountered according to a fixed variable ordering, and reduced, i.e., redundant decision nodes with two identical successors are removed [5].

Given some vector of N Boolean variables $\vec{x}$ and a decision diagram representing some function f, binary decision diagrams can very concisely represent sets $S \subseteq 2^N$, where $S = \{x \mid f(x) = 1\}$ is the set of valuations to $\vec{x}$ for which the represented Boolean function f evaluates to true. One can obtain a BDD for any set by means of a binary encoding of the elements of the set and the application of the Boole-Shannon decomposition $S = x \cdot S_x + \bar{x} \cdot S_{\bar{x}}$, where S_x is the subset of S where x equals 1, and $S_{\bar{x}}$ is the subset of S where x equals 0 [3, 30]. It is well known that BDDs can be incredibly efficient if a suitable variable ordering is found and the represented set is encoded in a way that results in small decision diagrams. Further study of variable ordering and encoding lie outside of the scope of the current paper and we refer to [15] for a comprehensive overview of binary decision diagrams.

3 Symbolic fixpoint algorithms that yield winning strategies

We now present our approach to converting the explicit parity game algorithms to their symbolic counterparts. In order to solve a parity game symbolically, the parity game itself has to be represented symbolically using BDD. Furthermore, data structures and procedures from `DFI` and `FPJ` have to be modified in order to fit the symbolic paradigms. Due to space constraints, we do not present the explicit algorithms `DFI` and `FPI` here and instead refer to [13] and [25].

3.1 Encoding of the parity game

To use decision diagrams as the data structure to store parity games, we replace explicit data structures by their symbolic counterparts. We use a naive binary encoding. Sets are modelled as a functions over a vector $\vec{x}$. Our input parity games have vertices numbered from $0 \ldots n$ and we simply encode vertices by encoding these numbers. For example we use 4 Boolean variables to represent a parity game with 9 vertices. Notice that in a full LTL synthesis toolchain, we would retain more information about each vertex and could do better than a naive binary encoding. This is a topic of future work.

We have a BDD for the set V of all vertices in the parity game. Furthermore, we have BDDs for $V_\Diamond$, $V_\Box$ and a BDD for V^p for every priority p. We thus choose to represent the priority function using one BDD for every priority. An alternative could be to use a multi-terminal BDD (or algebraic decision diagram) for the priority function, but practical parity games have only few priorities anyway.

Finally, to represent the successor relation E, we introduce additional Boolean variables $\vec{x}'$. We use the original variables $\vec{x}$ for the source vertex and the new variables $\vec{x}'$ for the target vertex. We order all $\vec{x}$ before all $\vec{x}'$ in the variable ordering.

3.2 Symbolic alternatives for explicit procedures

When manipulating binary decision diagrams, the internal nodes are never modified directly. Instead, BDD operations compute new decision diagrams, reusing existing nodes as much as possible, thus ensuring that BDDs are canonical representations. As a consequence, computing something like a union $S := S \vee T$ results in a new BDD, that may or may not share nodes with the original representations of S and T. When we perform more complicated operations, such as $S := S \wedge (A \vee B \vee C)$, the order of operations may have a major influence on the size of intermediate BDDs.

The original algorithms as presented in e.g. [13] use for-loops over individual vertices. In order to have efficient BDD implementations, this must be avoided whenever possible. The power of symbolic data structures lies in representing and manipulating sets of vertices rather than individual vertices.

When algorithms use data structures such as arrays that assign some value to each vertex, we need a symbolic representation of such arrays. If only few values are assigned, then we can represent these arrays using one BDD for each possible value. Alternatively we could encode the assigned values or use multi-terminal BDDs. For example we use sets V^p for every priority in the game, since there are only few priorities anyway. If the parity game had unique priorities for every vertex, such a representation would be very inefficient. The representation may also influence operations that might be wasteful when operating on a set encoding all values, and efficient when operating on sets representing the value that we are interested in. Hence, the choice of representation is not trivial, and may have significant impact on the performance of algorithms.

Two particular procedures are computing the sets $\Diamond X$ and $\Box X$, i.e., predecessors of X:

$$\begin{aligned} \Diamond X &:= \{v \in V \mid \exists u\colon v \to u \wedge u \in X\} \\ \Box X &:= \{v \in V \mid \forall u\colon v \to u \Rightarrow u \in X\} \end{aligned}$$

Computing the preimage $\Diamond X$ is a well known operation. We first substitute the Boolean variables of the target set X, which is represented using Boolean variables $\vec{x}$, to use Boolean variables $\vec{x}'$ instead. Then we intersect the successor relation E with this set X'. By performing existential quantification of the $\vec{x}'$ variables on the intersection, we obtain all vertices that have a successor in the original set X.

Computing $\Box X$ is slightly different. We obtain the set X' as for $\Diamond X$. Now we compute $v \to u \Rightarrow u \in X \equiv \neg(v \to u) \vee (u \in X)$ simply by adding X' to the complement of E: $\neg E \vee X'$: the result is the set of edges that either do not exist or whose successor is in X; any edge not in this set is an existing edge to a vertex outside X. Hence, if we now perform universal quantification of the $\vec{x}'$ variables on this intersection, we obtain all vertices without successors outside X, i.e., $\Box X$.

3.3 Symbolic `DFI`

Algorithm 1 outlines the symbolic implementation of `DFI`. The symbolic `DFI` algorithm uses $d+1$ BDDs plus 3 additional BDDs to store essential information plus 2 helper sets for subresults:

- the current estimation of distractions Z
- a F_p set for every priority p with $0 \leq p \leq d$; so in total these are $d+1$ BDDs
- a set S for the chosen strategies.
- sets X and W that store subresults.

```
1  def dfi(PG):
2      Z ← ∅                                                  initialize distractions set
3      F_0 ← ∅, ..., F_d ← ∅                                  initialize each freezing set
4      S ← ∅                                                  initialize the strategy to be empty
5      p ← 0                                                  we start updating with the lowest priority
6      while p ≤ d :                                          run until all fixpoints are stable
7          α ← p mod 2                                        get the current player
8          X ← ((((V^p ∧ ¬Z) ∧ ¬F_0) ∧ ¬F_1) ...) ∧ ¬F_d      compute unfrozen vertices of priority p
9          if α = 0 : Z' ← X ∧ ¬onestep_0(X, Z)               compute now-distractions (winning for 1)
10         else: Z' ← onestep_0(X, Z)                         compute now-distractions (winning for 0)
11         Z ← Z ∨ Z'                                         add new distactions to Z
12         S ← (S ∧ ¬X)                                       reset strategy for unfrozen vertices
13         S ← S ∨ (X ∧ V_◇ ∧ E ∧ subst(even(Z), x⃗ → x⃗'))     add strategy for Even to even
14         S ← S ∨ (X ∧ V_□ ∧ E ∧ subst(odd(Z), x⃗ → x⃗'))      add strategy for Odd to odd
15         if Z' ≠ ∅ :                                        did we get new distractions?
16             X ← ((V^0 ∨ V^1) ...) ∨ V^{p-1}                get all vertices < p
17             X ← (((X ∧ ¬F_0) ∧ ¬F_1) ...) ∧ ¬F_d           ... that are not frozen
18             if α = 0 : W ← X ∧ even(Z)                     ... and are won by even
19             else: W ← X ∧ odd(Z)                           ... and are won by odd
20             F_p ← F_p ∨ (X ∧ ¬W)                           freeze lower vertices won by ᾱ: X \ W
21             Z ← Z ∧ ¬W                                     reset lower vertices won by α: W
22             p ← 0                                          continue with lowest priority
23         else:                                              no new distractions?
24             F_p ← ∅                                        thaw vertices
25             p ← p + 1                                      continue with next priority
26     W_◇ ← even(Z)                                          winning region for Even
27     W_□ ← odd(Z)                                           winning region for Odd
28     S_◇ ← W_◇ ∧ V_◇ ∧ S                                    restrict strategy to winning region
29     S_□ ← W_□ ∧ V_□ ∧ S                                    restrict strategy to winning region
30     return W_◇, W_□, S_◇, S_□
31 def onestep_0(X, Z):
32     Z' ← subst(even(Z), x⃗ → x⃗')                           rename variables of even region to x⃗'
33     A ← V_◇ ∧ X ∧ exists(E ∧ Z', x⃗')                       even-to-even
34     B ← V_□ ∧ X ∧ forall(V_□ ∧ (¬E ∨ Z'), x⃗')              odd-to-even, encoding of u → v ⇒ v ∈ Z ≡ ¬E ∨ Z'
35     return A ∨ B
36 def even(Z):
37     return (V_0 ∧ ¬Z) ∨ (V_1 ∧ Z)        even-priority and not in Z plus odd-priority and in Z
38 def odd(Z):
39     return (V_0 ∧ Z) ∨ (V_1 ∧ ¬Z)        even-priority and in Z plus odd-priority and not in Z
```

Algorithm 1: The DFI algorithm using BDD operations

Some procedures have been redesigned to better suit a symbolic implementation. Lines 8-14 are the symbolic or set-based version of an explicit iteration of all vertices of priority p that are not frozen. There are different possiblities for computing the various sets. For example, we could choose a different order of operations at line 8. In our initial experiments, the presented algorithm performed best. Similarly, we implement a $\texttt{onestep}_0$ method that only computes the vertices in X (unfrozen and of priority p) that reach the region X that is currently good for player Even.

To update the strategy for the unfrozen vertices of priority p, the explicit algorithm simply overwrites the current strategy in an `int` array with a single chosen successor. Our symbolic approach is different, as the strategy is now a set of edges. First we remove the current strategy (line 12). Then we add all strategies for all vertices controlled by Even that can go to the current even region. Finally we add all strategies for all vertices controlled by Odd that can go to the current odd region. Notice now that the set S can have multiple successors for a single vertex. While we currently do not use this, in a fully symbolic LTL toolchain this may be quite beneficial, as it allows the synthesis algorithm to pick different valid strategies depending on which combination yields the most optimal controller. This is a serious advantage that explicit parity game solving algorithms do not have.

There are a few more explicit iterations over vertices in the original algorithm of [13] that we can replace with set-based operations. To freeze and thaw vertices, the original algorithm uses a simple iteration over vertices. This is a bit more work using BDDs and again we have more options to play with the order in which operations are carried out. See lines 16–21 in Algorithm 1. There can be many different ways to compute the set of unfrozen vertices of a priority less than or equal to p. We found that the presented method worked well, but there is room for improvement here, including precomputing the set $V^{\leq p}$, and using e.g. heuristics to determine the best order of operations at runtime.

Notice that we can remove lines 12–14 and 28–29 without affecting the rest of the algorithm. In some cases, for example if we just want to solve realisability problems instead of full synthesis, it is not required to obtain a winning strategy. The `DFI` algorithm can be run without computing the strategy. Thus we have a variant of symbolic `DFI` with no strategy computation.

3.4 Symbolic `FPJ`

Algorithm 2 outlines the symbolic implementation of `FPJ`. The symbolic `FPJ` algorithm uses 2 additional BDDs to store essential information plus 3 helper sets for subresults:

- the current estimation of the Even winning set Z
- the current justification graph J (which is a set of edges)
- a set of currently unjustified vertices U derived from J
- the set of changed vertices C
- the set of pruned vertices R

On the procedural side of the symbolic FPJ implementation, there are only few significant modifications. Compared to DFI, the symbolic implementation of FPJ is much more similar to its original explicit counterpart because the pseudo-code provided by Lapauw et al. [25] is based on set-operations. Therefore no major changes were required. Only the method $\texttt{strategy}_{\Diamond}$ was changed, as the explicit algorithm iterates over all vertices in U in order to differentiate behavior depending on the winner of the vertex, and whether the winner also owns the vertex. This corresponds to the symbolic implementation of lines 40-44 of algorithm 2. BDDs are first computed for all three cases (*Even* controls and wins, *Odd* controls and wins, *Even* or *Odd* controls and loses). Then the algorithm proceeds to compute winning moves for vertices that are controlled and won by the same player, and compute all outgoing edges for vertices lost by the player which controls it.

```
1  def fpj(PG):
2      Z ← V_0                                   start with "Even wins all even-priority vertices"
3      J ← ∅                                     start with empty justification graph
4      U ← V ∧ ¬exists(J, x')                    compute unjustified vertices
5      while U ≠ ∅ :                             until all vertices justified
6          Z, J ← next(Z, J, U)                  update Z and J
7          U ← V ∧ ¬exists(J, x')                recompute unjustified vertices
8      W_◇ ← Z                                   player Even wins vertices in Z
9      W_□ ← V ∧ ¬Z                              player Odd wins vertices not in Z
10     S_◇ ← J ∧ V_◇ ∧ W_◇                       restrict strategy to winning region
11     S_□ ← J ∧ V_□ ∧ W_□                       restrict strategy to winning region
12     return W_◇, W_□, S_◇, S_□
13 def next(Z, J, U):
14     p ← 0
15     while V^p ∧ U = ∅ : p ← p + 1             find lowest priority
16     U ← V^p ∧ U                               restrict U to vertices of priority p
17     C ← U ∧ xor(Z, phi(Z))                    compute unjustified p-vertices whose winning region changes
18     if C ≠ ∅ :                                did anything Change?
19         R ← reaches(J, C)                     compute justified vertices depending on vertices in C
20         if p mod 2 = 0 :                      if p is even...
21             Z_R ← (Z ∧ ¬(R ∧ V_1)) ∧ (((V^0 ∨ V^1)...) ∨ V^{p-1})   remove odd-priority vertices < p
22         else:                                 otherwise...
23             Z_R ← (Z ∨ (R ∧ V_0)) ∧ (((V^0 ∨ V^1)...) ∨ V^{p-1})    reset even-priority vertices < p
24         V^{>p} ← (((V^{p+1} ∨ V^{p+2})...) ∨ V^d)                   compute V^{>p}
25         Z' ← (Z ∧ V^{>p}) ∨ xor(Z ∧ V^p, C) ∨ Z_R                   update Even winning region Z
26         J' ← (J ∧ ¬R) ∨ strategy_◇(Z', C)     update justification graph (prune, then add)
27     else:                                     did nothing Change?
28         Z' ← Z                                no change to Z
29         J' ← J ∨ strategy_◇(Z', U)            update justification graph (no prune, just add)
30     return Z', J'
31 def reaches(J, X):
32     X' ← ∅                                    start with empty set
33     while X' ≠ X :                            until fixpoint...
34         X' ← X                                update previous X
35         X ← X ∨ (exists(J ∧ subst(X, x → x'), x'))   add preimage in J of X to X
36     return X                                  return all vertices reaching X via J
37 def phi(Z):
38     return (X_◇ ∧ ◇Z) ∨ (X_□ ∧ □Z)            compute onestep_0 basically
39 def strategy_◇(Z, U):
40     W_E ← U ∧ V_◇ ∧ Z                         all Even-controlled vertices in U that are won by Even
41     W_O ← U ∧ V_□ ∧ ¬Z                        all Odd-controlled vertices in U that are won by Odd
42     L ← U ∧ ¬(X_0 ∨ X_1) ∧ V                  (losing) vertices in U but not in X_0 or X_1
43     Z' ← subst(Z, x → x')                     rename Even winning region to x' variables
44     return (W_E ∧ Z' ∧ E) ∨ (W_O ∧ ¬Z' ∧ E) ∨ (L ∧ E)   good edges when winning, else all
```

Algorithm 2: The FPJ algorithm using BDD operations

Notice that contrary to symbolic DFI, computing the winning strategies is now integrated into the algorithm. There is therefore no version of FPJ with no strategy computation.

4 Implementation of the symbolic algorithms

We implemented the symbolic algorithms using Python and the BDD packages provided by the Python library dd[1]. This library offers the CUDD backend using Cython bindings. Furthermore, we used the LineProfiler package[2] to analyse the performance of the algorithms and find optimisations as discussed below. We assume that a BDD package implements the methods exists, forall, binary operators and, or and xor, the method subst for variable renaming, and neg to compute complement sets in constant time using complement edges. These are trivial requirements fulfilled by typical BDD implementations.

The implementations of our symbolic algorithms and the scripts and data of the empirical experiments are available online [3].

Optimisations A few optimisations were tested using LineProfiler. LineProfiler allows for profiling specific functions, and creates an overview of time spent on each line of Python code. This allows for easy analysis of small variations in code.

The order of application of BDD operations can greatly influence the practical performance and even resolve bottlenecks in memory usage if the peak number of BDDs is sufficiently smaller. At several points in algorithm 1, we combine some BDD (A) with another BDD (B) which is a combination of many smaller BDDs ($B_0 \dots B_i$) (i.e. $V_{<p}$ and the set of all frozen vertices). We do this by adding BDDs $B_0 \dots B_i$ to A one by one. This avoids combining two large BDDs at once by adding smaller BDDs to one large BDD instead. Using LineProfiler, we found performance improvements of 11% and 20% for lines 8 and 17 of algorithm 1 for the *full arbiter unreal 3* case from the empirical evaluation in section 5, which is the second largest BDD of the used benchmark sets.

In the symbolic FPJ algorithm, we need the set of unjustified vertices several times. We compute this set once per iteration (lines 4,7 in Algorithm 2). Furthermore, we found that the order of operations at lines 40–42 appears to be most optimal.

5 Empirical Evaluation

The empirical evaluation aims to study the performance of the discussed symbolic DFI and FPJ algorithms. We compare four different symbolic algorithms:

- dfi: the standard symbolic DFI algorithm
- dfi-ns: symbolic DFI with no strategy computation
- fpj: symbolic FPJ
- zlk: symbolic Zielonka's recursive algorithm, provided by Sanchez et al [29].

The main goals of the evaluations are to see whether the algorithms that compute strategies are competitive with state-of-the-art work, and furthermore to see whether not computing winning strategies influences the performance of symbolic DFI.

[1] https://github.com/tulip-control/dd

[2] https://github.com/pyutils/line_profiler

[3] https://github.com/olijzenga/bdd-parity-game-solver/tree/v1.1

Set	#games	avg. n	max. n	priorities	avg. out-degree	avg. #bdd nodes
Lily	23	409	3047	3-8	1.47	1064
AMBA	8	2461	18635	3-4	1.43	1780
ltl2dba	58	921	31717	4-7	1.49	1876
Arbiters	15	2650	20928	4	1.65	6068
Detector	2	98	120	4	1.47	373
Buffer	2	14	17	4	1.30	72
Load balancer	11	1149	4712	3-8	1.65	2589

Table 1: Statistics of benchmark sets used for empirical evaluation where n is the number of vertices

Set	computes strategy		no strategy	
	`dfi`	`fpj`	`dfi-ns`	`zlk`
Lily	0.38	0.54	**0.19**	0.39
AMBA	0.39	0.70	**0.29**	0.63
ltl2dba	21.11	16.13	**14.63**	15.08
Arbiters	48.17	23.98	32.48	**17.54**
Detector	0.0066	0.0068	**0.0046**	0.0047
Buffer	0.00082	0.00058	**0.00052**	0.00057
Loadbalancer	0.50	0.91	**0.37**	0.62

Table 2: Cumulative time in sec. average over 5 runs used to solve all games.

Set	computes strategy		no strategy	
	`dfi`	`fpj`	`dfi-ns`	`zlk`
Lily	41647	51747	**37431**	44595
AMBA	34299	36261	**28143**	34729
ltl2dba	212242	263743	**183830**	326327
Arbiters	172235	231625	**153008**	218382
Detector	1400	1393	**1211**	1427
Buffer	244	244	**222**	253
Load balancer	50411	66389	**44407**	60264

Table 3: Average Peak Live BDD Nodes (peak nodes per algorithm run)

We evaluate the symbolic algorithms using the benchmark sets from the PGAME realizability track of SYNTCOMP 2020. More detailed information on these benchmarks can be found in [19]. The SYNTCOMP benchmarks are parity automatons in extended HOA format [28] which were converted to explicit parity games in `PGSolver` format [18] by Knor[4], which is one of the participant tools in the 2020 edition of SYNTCOMP. Contrary to some of the studies in the literature, we avoid using random games as we do not think that random games are representative for practical parity games.

Metrics of the used benchmark sets are shown in Table 1. For the parity games in each set we have the average and highest number of vertices, priorities and the average outgoing degree. The average outgoing degree is the average of $|E|/|V|$ for each parity game. Comparing these two metrics gives an indication of the compression ratio of the BDDs. For example, the *AMBA* benchmark set has a high compression ratio because only few BDD nodes are used to represent relatively many vertices and edges.

Only benchmarks from formal verification and synthesis problems were used for the empirical evaluation. Random parity games were not used, except to test the implementations. In order to test our implementations, we used a combination of test games from the Oink [11] distribution as well as many randomly generated games. The implementations of our symbolic algorithms and the scripts and data of the empirical experiments are available online.

In Table 2 and Table 3, we see the results for all benchmark sets. The provided times are cumulative for the entire benchmark set. The relative standard deviation (shown in parenthesis) is computed as a percentage of the mean time per parity game, and displayed as an average for each benchmark set. Testing was done on an Intel Core i7-7700HQ CPU @ 2.80GHz in conjunction with 16GB of RAM, running Python 3.6.9 on Ubuntu 18.04. None of the algorithms used multi-threading. The best results are highlighted in blue. The recorded times do not include the time for converting the original explicit parity game to a symbolic parity game, but only the difference between the four algorithms.

The first observation is that the implemented algorithms have a very similar performance compared to our reference symbolic solver, symbolic Zielonka. There is only one benchmark class where Zielonka is the fastest solver, although this is simultaneously the benchmark class where all solvers require most time. As one might expect, `dfi-ns` and `zlk`, which do not compute winning strategies, were almost always faster than the algorithms that do compute winning strategies. We also notice that `dfi` and `dfi-ns` are typically faster than `fpj`, but not for the *parameterized arbiter* specifications, where `fpj` outperformed `dfi` and `dfi-ns` for the *full arbiter* cases, including one of 20928 vertices. Further study might reveal why the `fpj` algorithm is particularly effective for these cases.

To give an idea of the memory usage of the evaluated algorithms, the peak number of live BDD nodes was recorded for each run, using CUDD's reporting of the peak number of nodes. In table 3, we see that `dfi-ns` consistently used the least amount of BDD nodes. `dfi` used fewer BDD nodes than `zlk` for all benchmark sets. Zielonka constructs subgames which are copies of the original games, which causes Zielonka's algorithm to be less memory efficient despite not computing winning strategies. `fpj` also uses fewer BDD nodes than `zlk` for some cases, but this difference is neither as consistent nor as significant as it is between `dfi` and `zlk`. We conclude that a symbolic implementation of `dfi` may be a suitable choice when memory-usage must be as low as possible, although the differences are quite small.

To obtain insight into where `dfi`, `dfi-ns` and `fpj` spend most of their time, `LineProfiler` was again used. This showed that all profiled algorithms were consistently using 90-99% of their time on preimage computations. Both DFI and FPJ use these computations to update their estimate of the winning area (`onestep` and `phi` functions respectively), and to compute winning moves. Preimage computations consist of several steps, all resulting in intermediate BDDs rather than doing all computations at once

[4] `https://github.com/trolando/knor`

and then producing one final optimised and reordered result.

In general, the `dfi-ns` algorithm performs best across most of the evaluated benchmark sets. This shows that DFI can be a good alternative to Zielonka for both explicit (as shown in [13]) and symbolic parity games. For the strategy derivation algorithms `dfi` and `fpj` there is no clear winner. Both `dfi` and `dfi-ns` have shown to be most memory efficient, especially when compared to Zielonka. In terms of time efficiency, it is hard to draw any major conclusions from our empirical evaluation as it remains hard to predict when an algorithm will out-perform the other, and different benchmark sets have entirely different properties and structures.

6 Conclusion and discussion

We have proposed and implemented two symbolic parity game algorithms that yield winning strategies. This is an important step towards fully symbolic LTL synthesis via parity games, which requires the winning strategies in order to construct controllers. Our evaluation on real practical games derived from LTL synthesis demonstrates that the method is as fast as other symbolic parity game solvers, while producing winning strategies.

This paper represents a first step into symbolic algorithms that derive strategies and there is plenty of room to improve. Most interesting would be to look at the encoding and variable reordering, but this is also the most difficult direction as we would need to consider the full LTL synthesis toolchain, including symbolic encoding of LTL to parity automata. In this work, we used a simple variable ordering that places all pre-variables before all post-variables; alternatives such as interleaved variable orderings which are common in model checking would be recommended as a first step to try. Easier improvements can be found by considering crafting specialised BDD operations that combine several operations in fewer steps. In particular preimage computation dominates the runtime of our algorithms so any improvements to preimage computation immediately improves the entire algorithm. Furthermore, we currently make use of sets $V^{>p}$ and $V^{<p}$ that could be precomputed. We do not yet make use of parallel computation, as we used CUDD for this study, but if we switch to a BDD package like Sylvan, we could also find parallel speedups. We did not investigate tuning CUDD with respect to automated variable reordering, which has shown to be beneficial for the related field of safety synthesis, although variable reordering tends to dominate runtimes of algorithms that depend on it. We did not compare with explicit algorithms like the implementations in Oink, but our experiments (Table 2) already show such small runtimes that a comparison is almost meaningless. For a proper comparison with explicit algorithms, we need examples of practical parity games that are much larger, actually challenging explicit parity game algorithms, and that might have some kind of structure that can be exploited by symbolic algorithms. Current benchmark sets with large practical parity games unfortunately target explicit solvers and do not retain any structure of the original specifications. Also, the implementations in Oink are highly optimized whereas our symbolic algorithms are not. We could consider using multi-terminal BDDs to obtain additional compression, but the current BDDs are small enough that approaches using multi-terminal BDDs are currently not attractive. Both `dfi` and `fpj` produce potentially many winning strategies, unlike explicit algorithms, since they do not select a single good successor when there are multiple good successors. It may be interesting to compare the two algorithms with respect to the number of obtained strategies, if there is a significant difference. Due to time and space constraints, we are unable to implement many of these ideas ourselves at this time.

Acknowledgements

The second author of this paper is supported by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 893732.

References

[1] Marco Bakera, Stefan Edelkamp, Peter Kissmann & Clemens D. Renner (2008): *Solving μ-Calculus Parity Games by Symbolic Planning*. In: *MoChArt*, *Lecture Notes in Computer Science* 5348, Springer, pp. 15–33, doi:10.1007/978-3-540-92221-6.

[2] Massimo Benerecetti, Daniele Dell'Erba & Fabio Mogavero (2018): *Solving parity games via priority promotion*. *Formal Methods Syst. Des.* 52(2), pp. 193–226, doi:10.1016/0304-3975(95)00188-3.

[3] George Boole (1854): *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*. 2, Walton and Maberly.

[4] Florian Bruse, Michael Falk & Martin Lange (2014): *The Fixpoint-Iteration Algorithm for Parity Games*. In: *GandALF*, *EPTCS* 161, pp. 116–130, doi:10.4204/EPTCS.161.12.

[5] Randal E. Bryant (1992): *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*. *ACM Comput. Surv.* 24(3), pp. 293–318, doi:10.1145/42282.46161.

[6] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill & L. J. Hwang (1992): *Symbolic Model Checking: 10^20 States and Beyond*. *Inf. Comput.* 98(2), pp. 142–170, doi:10.1016/0890-5401(92)90017-A.

[7] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li & Frank Stephan (2017): *Deciding parity games in quasipolynomial time*. In: *STOC*, ACM, pp. 252–263, doi:10.1145/3055399.3055409.

[8] Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger & Alexander Svozil (2018): *Quasipolynomial Set-Based Symbolic Algorithms for Parity Games*. In: *LPAR*, *EPiC Series in Computing* 57, EasyChair, pp. 233–253, doi:10.29007/5z5k.

[9] Wojciech Czerwinski, Laure Daviaud, Nathanaël Fijalkow, Marcin Jurdzinski, Ranko Lazic & Pawel Parys (2019): *Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games*. In: *SODA*, SIAM, pp. 2333–2349, doi:10.1137/1.9781611975482.142.

[10] Tom van Dijk (2018): *Attracting Tangles to Solve Parity Games*. In: *CAV (2)*, *Lecture Notes in Computer Science* 10982, Springer, pp. 198–215, doi:10.1007/978-3-319-96142-2_14.

[11] Tom van Dijk (2018): *Oink: An Implementation and Evaluation of Modern Parity Game Solvers*. In: *TACAS (1)*, *Lecture Notes in Computer Science* 10805, Springer, pp. 291–308, doi:10.1016/S0304-3975(98)00009-7.

[12] Tom van Dijk & Jaco van de Pol (2017): *Sylvan: multi-core framework for decision diagrams*. *Int. J. Softw. Tools Technol. Transf.* 19(6), pp. 675–696, doi:10.1007/s10009-016-0433-2.

[13] Tom van Dijk & Bob Rubbens (2019): *Simple Fixpoint Iteration To Solve Parity Games*. In: *GandALF*, *EPTCS* 305, pp. 123–139, doi:10.4204/EPTCS.305.9.

[14] Rolf Drechsler & Bernd Becker (1998): *Binary Decision Diagrams - Theory and Implementation*. Springer, doi:10.1007/978-1-4757-2892-7.

[15] Rolf Drechsler & Detlef Sieling (2001): *Binary decision diagrams in theory and practice*. *Int. J. Softw. Tools Technol. Transf.* 3(2), pp. 112–136, doi:10.1007/s100090100056.

[16] E. Allen Emerson & Charanjit S. Jutla (1991): *Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)*. In: *FOCS*, IEEE Computer Society, pp. 368–377, doi:10.1109/SFCS.1991.185392.

[17] John Fearnley, Sanjay Jain, Bart de Keijzer, Sven Schewe, Frank Stephan & Dominik Wojtczak (2019): *An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space*. *Int. J. Softw. Tools Technol. Transf.* 21(3), pp. 325–349, doi:10.1016/S0304-3975(98)00009-7.

[18] Oliver Friedmann & Martin Lange (2009): *The PGSolver collection of parity game solvers*. University of Munich, pp. 4–6.

[19] Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur & Leander Tentrup (2017): *The 4th Reactive Synthesis Competition (SYNTCOMP 2017): Benchmarks, Participants & Results*. In: *SYNT@CAV*, *EPTCS* 260, pp. 116–143, doi:10.4204/EPTCS.260.10.

[20] Marcin Jurdzinski (1998): *Deciding the Winner in Parity Games is in UP \cap co-Up*. Inf. Process. Lett. 68(3), pp. 119–124, doi:10.1016/S0020-0190(98)00150-1.

[21] Marcin Jurdzinski (2000): *Small Progress Measures for Solving Parity Games*. In: *STACS*, Lecture Notes in Computer Science 1770, Springer, pp. 290–301, doi:10.1007/3-540-46541-3_24.

[22] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom & Tom van Dijk (2015): *LTSmin: High-Performance Language-Independent Model Checking*. In: *TACAS*, Lecture Notes in Computer Science 9035, Springer, pp. 692–707, doi:10.1016/S0304-3975(98)00009-7.

[23] Gijs Kant & Jaco van de Pol (2014): *Generating and Solving Symbolic Parity Games*. In: *GRAPHITE*, *EPTCS* 159, pp. 2–14, doi:10.4204/EPTCS.159.2.

[24] Orna Kupferman & Moshe Y. Vardi (1998): *Weak Alternating Automata and Tree Automata Emptiness*. In: *STOC*, ACM, pp. 224–233, doi:10.1145/276698.276748.

[25] Ruben Lapauw, Maurice Bruynooghe & Marc Denecker (2020): *Improving Parity Game Solvers with Justifications*. In: *VMCAI*, Lecture Notes in Computer Science 11990, Springer, pp. 449–470, doi:10.1016/S0304-3975(98)00009-7.

[26] Philipp J. Meyer, Salomon Sickert & Michael Luttenberger (2018): *Strix: Explicit Reactive Synthesis Strikes Back!* In: *CAV (1)*, Lecture Notes in Computer Science 10981, Springer, pp. 578–586, doi:10.1007/978-3-319-96145-3_31.

[27] Pawel Parys (2019): *Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time*. In: *MFCS*, *LIPIcs* 138, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:13, doi:10.4230/LIPIcs.MFCS.2019.10.

[28] Guillermo A. Pérez (2019): *The Extended HOA Format for Synthesis*. *CoRR* abs/1912.05793.

[29] Lisette Sanchez, Wieger Wesselink & Tim A. C. Willemse (2018): *A Comparison of BDD-Based Parity Game Solvers*. In: *GandALF*, *EPTCS* 277, pp. 103–117, doi:10.4204/EPTCS.277.8.

[30] Claude E Shannon (1938): *A symbolic analysis of relay and switching circuits*. Electrical Engineering 57(12), pp. 713–723, doi:10.1109/EE.1938.6431064.

[31] Fabio Somenzi (2001): *Efficient manipulation of decision diagrams*. Int. J. Softw. Tools Technol. Transf. 3(2), pp. 171–181, doi:10.1007/s100090100042.

[32] Antonio Di Stasio, Aniello Murano & Moshe Y. Vardi (2018): *Solving Parity Games: Explicit vs Symbolic*. In: *CIAA*, Lecture Notes in Computer Science 10977, Springer, pp. 159–172, doi:10.1016/S0304-3975(98)00009-7.

[33] Jens Vöge & Marcin Jurdzinski (2000): *A Discrete Strategy Improvement Algorithm for Solving Parity Games*. In: *CAV*, Lecture Notes in Computer Science 1855, Springer, pp. 202–215, doi:10.1016/0304-3975(95)00188-3.

[34] Igor Walukiewicz (2002): *Monadic second-order logic on tree-like structures*. Theor. Comput. Sci. 275(1-2), pp. 311–346, doi:10.1016/S0304-3975(01)00185-2.

[35] Wieslaw Zielonka (1998): *Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees*. Theor. Comput. Sci. 200(1-2), pp. 135–183, doi:10.1016/S0304-3975(98)00009-7.