Symbolic Parity Games: Two Novel Fixpoint Iteration Algorithms with Strategy Derivation

Oebele Lijzenga
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
o.r.lijzenga@student.utwente.nl

ABSTRACT

In this paper, we study symbolic parity game solving using BDDs. The state of the art of symbolic parity game algorithms is improved by implementing two novel fixpoint iteration algorithms. Contrary to current symbolic algorithms, our algorithms also derive winning strategies. Empirical evaluation compares the new symbolic algorithms with a symbolic implementation of Zielonka's recursive algorithm over benchmark sets from SYNTCOMP 2020. We conclude that both new algorithms are competitive with Zielonka's algorithm, while also providing winning strategies.

Keywords

Parity games, binary decision diagrams, complexity theory, formal verification, model checking

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.

Parity games are an interesting field of study for many reasons. In [17], Van Dijk states that "their study has been motivated by their relation 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"(p.291). In formal verification, we want to know if some condition can be satisfied or realised. This corresponds to computing the winning areas of a parity game.

33rd Twente Student Conference on IT July 3rd, 2020, Enschede, The Netherlands.

In synthesis, we want to obtain a controller which satisfies a specification. This corresponds to computing the winning strategy for a parity game.

It is widely believed that a polynomial time solution exists for determining the winner of a parity game [11]. So far, only quasi-polynomial time solutions have been found [3, 7]. For this reason, parity games are also widely studied with regards to complexity theory.

Due to the use of parity games in verification and synthesis problems, parity games can grow quite large. It is common for explicit parity game solvers to require over 8GB of memory for large specifications [16]. In the past, symbolic parity game solvers using BDDs (binary decision diagrams) have been explored [1, 16, 4, 12]. Symbolic parity game solvers replace explicit data-structures with implicit (i.e. symbolic) ones like BDDs, using powerful well-optimised packages like CUDD[1], Sylvan[2] and BuDDy[3]. The implicit nature of such data-structures, if used properly, can lead to massively reduced memory-usage [16].

Well known parity game solving algorithms like Zielonka's recursive algorithm [22] and priority promotion [2] can derive winning strategies and have been implemented symbolically by e.g. Sanchez et al. [16]. However, these symbolic implementations do not include derivation of winning strategies. The results for symbolic parity game solvers are still quite promising, and show that they should be more extensively explored.

This paper aims to improve the state of the art of symbolic parity game algorithms by implementing fixpoint iteration algorithms capable of deriving winning strategies. We implement the fixpoint iteration algorithm with justifications proposed by Lapauw et al. in [15] (FPJ), and the fixpoint iteration algorithm with distractions and freezing proposed by Van Dijk et al. in [19] (DFI). We look for potential advantages in deriving and representing winning strategies symbolically, while replicating the known advantages of symbolic algorithms found in previous studies on symbolic parity games.

2. PRELIMINARIES

2.1 Parity games

We define a parity game PG as a tuple (V, $V_\diamond$, $V_\square$, E, pr). $V_\diamond$ and $V_\square$ partition the set V, indicating which vertices are controlled by player *Even* and *Odd* respectively. Thus, $V_\diamond \cap V_\square = \emptyset$ and $V_\diamond \cup V_\square = V$. We say that vertices with an even priority are of *Even* parity, and vertices with an odd priority are of *Odd* parity, denoted by V_0 and V_1

[1] `https://github.com/johnyf/cudd`
[2] `https://github.com/utwente-fmt/sylvan`
[3] `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/`

respectively. The mapping $E \subseteq V \times V$ describes which moves can be made from each vertex in V, and each vertex has at least one successor. We also write $E(u)$ for all successors of u, and $u \to v$ if $v \in E(u)$. Furthermore the function $pr\colon\ V \to \{0, 1, ..., d\}$ assigns a priority to each vertex, where d is the highest priority in the game. So if vertex v has priority 3, then $pr(v) = 3$. We denote a player as $\alpha \in \{\Diamond, \Box\}$ where $\Diamond$ represents player *Even* and $\Box$ represents player *Odd*. The opponent of player α is denoted by $\bar{\alpha}$. For convenience, we can filter elements of a set of vertices by their parity as follows: $X_{p\in\{0,1\}} = X \cap V_p$. Similarly, for the owner of vertices we have $X_{\alpha\in\{\Diamond,\Box\}} = X \cap V_\alpha$.

Finally, we take notations from the modal μ-calculus[21] to denote successor states. In the context of parity games, we refer to u as a successor of v if $u \to v$. We define the following operators:

$$\Diamond X = \{v \in V \mid \exists u\colon v \to u \wedge u \in X\} \tag{1}$$

$$\Box X = \{v \in V \mid \forall u\colon v \to u \implies u \in X\} \tag{2}$$

Formula 1 computes all vertices that have an edge to X. This is the preimage of X. Formula 2 computes all vertices for which all outgoing edges go to X.

Figure 1 shows an example of a parity game taken from lectures notes of Van Dijk [18]. The diamond-shaped vertices are owned by player *Even*, and the box-shaped vertices are owned by player *Odd*. All vertices of this parity game are won by *Odd* because from all vertices, *Odd* can make moves in such a way that always results in an infinite sequence of vertex priorities where the highest priority is odd, regardless of the moves of *Even*.

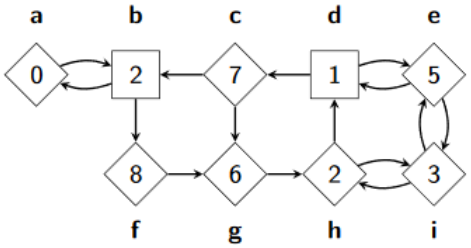

Figure 1. Parity game example (Van Dijk 2019)

2.2 Binary Decision Diagrams

Binary Decision Diagrams [5] (BDDs) are a frequently used data structure for the representation and manipulation of Boolean functions. Figure 2 shows an example of a BDD. This particular BDD is equivalent to the function $Y(\vec{x}) = x_1 \oplus x_2 \oplus x_3$ which can be used to represent the set $Y = \{x \in X \mid Y(x)\}$ in a memory efficient manner as it does not require a copy of each individual element of Y from X. Thus, BDDs can be used to describe sets as Boolean functions. In a BDD, each node always corresponds to a single Boolean variable, and one Boolean variable can correspond to any number of nodes. All nodes have two outgoing edges. One for when the nodes corresponding Boolean variable is true, and the other for when it is false. If input vector $\vec{x}$ for some BDD can represent all possible elements of a set X, then we can obtain any subset of X through a BDD.

One of the strengths of BDDs is that some structures can be very efficiently represented if a favourable variable ordering

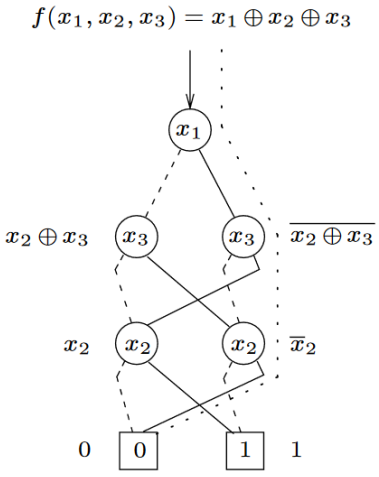

Figure 2. BDD example (Drechsler 2001)

for that structure is found. Conversely, a poor variable ordering can lead to much larger and more inefficient BDDs. For a comprehensive overview on BDDs and what can make them both efficient and inefficient were refer to [6].

Some commonly used BDD packages were previously mentioned. Such packages aim for the fastest possible manipulation of BDDs. To achieve this, BDDs are given a static variable ordering which yields ordered BDDs (OBDD). Variables are encountered in the same order for every path over an OBDD. As a result, an OBDD is *canonical* which means that for each Boolean function there exists only one OBDD which represents it. Due to their canonicity, equivalence checking of OBDDs is trivial.

BDDs are difficult to use as data structures for algorithms in comparison to their explicit counterparts. There are two reasons for this. First, most common problems are provided explicitly, requiring conversion to a symbolic representation which is rather time consuming and often results in inefficient BDDs. Additionally, the result of an algorithm is often used in its explicit form, requiring computations of satisfying assignments (SAT) of the symbolic result. Thus, BDDs are most effective when used in multiple successive stages of some piece of software, as conversions between explicit and symbolic are not desirable. Secondly, lack of knowledge on BDDs and internal operation of BDD packages can result in sub-optimal implementations and hidden complexities. In [20], Van Dijk et al. demonstrate the many pitfalls of implementing symbolic algorithms. Poor order of application of BDD operations results in additional internal operations for the used BDD package which returns an canonical, optimised BDD for each intermediate result in a sequence of BDD operations, regardless of whether this is desirable.

2.3 Fixpoint Iteration

Fixpoint iteration algorithms for parity games are a class of algorithms which aim to iteratively refine an estimate of which vertices are won by which player. An initial estimate might, for example, be that each player wins all vertices of their parity. Each iteration of such an algorithm is a monotonic function which updates the estimate Z, unless the fixpoint is reached. This estimate Z can be the set of vertices estimated to be won by player *Even*, also implicitly estimating vertices won by *Odd*. When the fixpoint is reached, we know that Z contains all vertices won by player *Even*, and $V \setminus Z$ contains all vertices won by player *Odd*.

2.4 DFI

The DFI algorithm proposed by Van Dijk et al. [19] is a fixpoint iteration algorithm built around the idea of *distractions*. Other fixpoint iteration algorithms maintain a set Z of vertices estimated to be won by e.g. player *Even*. Instead, DFI tracks vertices which are a distraction. Distractions are vertices which one intuitively might expect player α to win, but are in reality won by $\bar{\alpha}$. DFI starts with the assumption that a player wins all vertices of their parity, and uses Z to store vertices which violate this assumption. In figure 1, vertex **f** is a distraction because its priority is high and even. But if *Even* always tries to play to **f** then *Odd* wins.

DFI is also capable of deriving winning strategies for parity games using a technique called *freezing*, which also improves the efficiency of the algorithm by omitting unnecessary re-computations when new distractions are found. Re-computations are required because a vertex being a distraction (or not) relies on which successors are won by which player. Similarly to all other fixpoint iteration algorithms, an iteration of DFI is monotonic. An iteration always finds new distractions until all distractions are found and the algorithm finishes. For an extensive explanation of the DFI algorithm and proof of its correctness, we refer to [19].

2.5 FPJ

The FPJ algorithm is the fixpoint iteration algorithm with justifications proposed by Lapauw et al. [15]. A justification graph is used as a data structure to construct a winning strategy, and reset lower vertices when the estimate is updated. This justification graph is updated during each iteration step of the algorithm. Justification graphs as used by Lapauw et al. were first proposed by Hou et al. [9]. For an in-depth overview of the FPJ algorithm we refer to [15].

Justification J is a subset of E. An edge e can be included in J (justified) for two reasons: e is a winning move, or the starting vertex of e is lost by its owner. When a vertex is estimated to be lost by its owner, all edges from that vertex are added to J. FPJ visits *unjustified* vertices, which are vertices for which no outgoing edges are included in J. When the estimate of winning vertices is updated, all edges from which the updated vertices can be reached over justification J are deleted from J. This causes vertices *depending* on other vertices to win (or lose) to be reset in a topological manner because they become *unjustified*, causing these vertices to be re-visited. When FPJ ends, J is transformed to a winning strategy by only taking the edges from J which are won by their owner.

3. RELATED WORK

As briefly mentioned in the introduction, there exists some past research on symbolic parity games. In [12], Kant et al. implement Zielonka's algorithm symbolically. This implementation is also capable of deriving winning strategies, but this is excluded from the empirical evaluation. The empirical evaluation compares symbolic Zielonka with other symbolic parity game solvers using parity games from real model checking problems. In [4], Stasio et al. implement Zielonka, APT [14] and a small progress measures algorithm. These algorithms are extensively empirically evaluated using several different classes of parity games, but the evaluation lacks a comparison with explicit algorithms. Finally, Sanchez et al. implement Zielonka, fixpoint iteration, APT and priority promotion symbolically without strategy derivation, and evaluate these implementations using random parity games and Keiren's benchmark set[13].

In general, symbolic parity game algorithms which derive winning strategies have received little attention. Presumably because there is still a lot to be learned about symbolic parity games themselves. Furthermore, symbolic implementations require a lot of knowledge on their data-structures, and practical applications are still limited as explicit implementations are usually more convenient. Symbolic parity games are still a valuable topic of research as the previously mentioned work has shown that there are significant benefits to using symbolic parity games over explicit ones, especially when it comes to memory usage.

We also notice that there is still no consensus on how parity game algorithms should be empirically evaluated. Benchmark sets (e.g. Keiren's) include a wide variety of parity games from real formal verification and synthesis problems. This is still far from ideal however, as it would be preferred to have a multitude of parity games for range combinations of number of priorities, average outgoing degree and number of vertices. For this reason, many empirical evaluations use random parity game generators which can be tuned to obtain any number of parity games for any combination of properties. It has however yet to be shown that random parity game generators can consistently replicate parity games originating from real formal verification and synthesis problems. As a result, there is a wide variety of levels in which random parity games have been incorporated into empirical evaluations. Some researchers avoid random parity games entirely, whereas others use them as their only means of evaluation. In addition, evaluations which use both random parity games and parity games used in practice, show a wide variety of value attached to results from random games.

4. BDDS AND PARITY GAMES

In order to solve a parity game symbolically, the parity game itself has to be represented symbolically, in our case using BDDs. Furthermore, data-structures and procedures from DFI and FPJ have to be transformed to their symbolic counterparts. Our implementations are available on Github[4]. The version of this repository at time of writing is tagged as `v1.0`.

4.1 Symbolic Parity Game Algorithms

4.1.1 Tooling

To implement symbolic algorithms, some BDD package is required. For the implementations discussed in this paper, the Python package `dd`[5] was used in combination with the CUDD backend using Cython bindings. The `dd` package provides a generic Python interface on top of several BDD back-ends including CUDD, Sylvan and BuDDy. Optimisations in algorithms were found by analysing their performance using the `LineProfiler`[6] package.

4.1.2 Data Structures

To use BDDs as a data-structure for parity games, explicit data-structures have to be replaced by their symbolic counterparts. Sets are modelled as a function over vector $\vec{x}$. For a parity game with $|V| = 8$, a vector $\vec{x}$ consisting of Boolean variables with $|\vec{x}| = 3$ is used. $\vec{x}$ has $2^{|\vec{x}|} = 8$ possible values, so V can be represented using the Boolean function $\mathcal{V}(\vec{x}) = true$ because all variable assignments of $\vec{x}$ correspond to a vertex in V. In the case that there is no $n \in \mathbb{N}$

[4] `https://github.com/olijzenga/bdd-parity-game-solver`
[5] `https://github.com/tulip-control/dd`
[6] `https://github.com/pyutils/line_profiler`

for which $2^n = |V|$, we take $|\vec{x}| = min\{n \in \mathbb{N} \mid 2^n \geq |V|\}$. Then, to reduce the possible values of $\vec{x}$ for which $\mathcal{V}$ holds true to 6, we treat $\vec{x}$ as a binary number and replicate the $(< |V|)$ operation as a Boolean function. For example, $|V| = 6$ would yield the function $\mathcal{V}(\vec{x}) = \neg(\vec{x}_2 \wedge \vec{x}_3)$ where $|\vec{x}| = 3$.

If we want to update some set of vertices represented by BDD $\mathcal{S}$ by adding all elements of the set represented by $\mathcal{T}$, we can do this by creating a new BDD. $\mathcal{S} \vee \mathcal{T}$ is equivalent to set $S \cup T$, thus we can add T to S by doing $\mathcal{S} \leftarrow \mathcal{S} \vee \mathcal{T}$. Conversely we can remove T from S by doing $\mathcal{S} \leftarrow \mathcal{S} \wedge \neg\mathcal{T}$. The BDD package then reduces the updated BDDs to their compact, canonical form. It is important to keep these reductions in mind because a poor application order of BDD-operations can severely increase the time required for reducing a BDD, whereas a good application order can massively improve it [20].

Using function $\mathcal{V}$ which represents all vertices V in a parity game, the partition of $V_\Diamond$ and $V_\Box$ over V can be defined using BDDs. We define $\mathcal{V}_\Diamond(\vec{x}) = f(\vec{x}) \wedge \mathcal{V}(\vec{x})$ and $\mathcal{V}_\Box(\vec{x}) = \neg f(\vec{x}) \wedge \mathcal{V}(\vec{x})$, where f represents vertices controlled by player *Even*.

A parity game has d priorities. Each vertex is symbolically mapped to a priority by defining a BDD for each priority, where each BDD represents all vertices with that priority: $\mathcal{P}(\vec{x})_{i \in \{1 \ldots d\}} = f_i(\vec{x}) \wedge \mathcal{V}(\vec{x})$ where f partitions V by vertex priorities.

Finally, to model edges E, a symbolic mapping or a set of tuples $\mathcal{E}$ is required. Vector $\vec{x}'$ is introduced which represents a set of successor vertices. In $\mathcal{E}$, $\vec{x}$ and $\vec{x}'$ represent start and destination vertices respectively. Thus, we have an edge from $\vec{x}$ to $\vec{x}'$ iff $\mathcal{E}(\vec{x}, \vec{x}')$. To demonstrate its functionality, edges with their destination vertex in the set represented by $\mathcal{U}(\vec{x})$ can be computed by substituting $\vec{x}$ with $\vec{x}'$ to obtain $\mathcal{U}(\vec{x}')$, and consequently computing $\mathcal{U} \wedge \mathcal{E}$ which is equivalent to $\{\vec{x}, \vec{x}' \mid \mathcal{U}(\vec{x}') \wedge \mathcal{E}(\vec{x}, \vec{x}')\}$.

4.1.3 Procedures

Besides interchanging the explicit data structures with their symbolic counterparts, some parts of explicit algorithms have to be changed to better suit BDDs. All occurrences of iteration through a set (i.e. `foreach` loops) have to be replaced with BDD-operations which avoid iteration through satisfying assignments (SAT) of a BDD. Such computation is very time-consuming. The power of symbolic (i.e. implicit) data structures lies it the fact that it is a *lazy* representation of an entire set, and we operate on that entire set instead of its individual elements. If during the entire algorithm we can avoid *eager* element-wise operations (i.e. SAT computations), then we can exclusively operate on compact (lazy) representations of entire sets. Eager operations on BDDs completely nullify the advantages of a symbolic data structure because it means that the compact implicit data structure is decompressed into its explicit counterpart.

We concluded that symbolic data structures are manipulated and evaluated in an entirely different way from explicit ones. For this reason, some procedures might not transfer as well to symbolic ones. This is especially noticeable when using an explicit map (i.e. hashmap). Implementations like the hashmap can be read in $\mathcal{O}(1)$ in most cases. Such mappings are used in parity games to map vertices to their priorities, and in DFI to map vertices to the priority that they are frozen at. Depending on the usage of the hashmap in the explicit parity game algorithm, finding an efficient symbolic counterpart can be quite difficult. Vertices of priority i can be obtained using $\mathcal{P}(\vec{x})_i$, but to obtain the priority of some vertex, the BDDs for each priority need to be traversed until one is found that intersects. Such a symbolic representation of the hashmap can be significantly less efficient depending on the way it is used. For this reason, some explicit implementations might not translate as well into a symbolic one. As an example, it is not guaranteed that performance improvements for explicit algorithms, like *freezing* in DFI, translate well into a symbolic implementation.

On the other hand, some parts of a symbolic implementation might be significantly more efficient than their explicit counterparts. For example, $\mathcal{P}(\vec{x})_i$ represents all vertices of priority i, but obtaining the same set from a hashmap which maps vertices to their priority is rather cumbersome.

Another point of interest for symbolic parity game algorithms is the way *decisions* are made. Symbolic algorithms perform operations on BDDs in order to achieve their result. They avoid accessing elements of sets represented by BDDs because it requires expensive SAT computations while negating the benefits in memory efficiency of BDDs. Still, all parity game algorithms need to make decisions at some point. Both the DFI and FPJ algorithm check whether their estimation of winning vertices is updated or not, and the behaviour of the algorithms change accordingly. Both discussed checks are, or can be converted to, a comparison to another BDD. Packages like CUDD order BDDs in such a way that they can be compared in $\mathcal{O}(1)$ because only the root node of the BDD needs to be compared. The $\mathcal{O}(1)$ BDD comparison allows for controlling the flow of parity game algorithms depending on the contents of a set represented by a BDD while avoiding SAT computations.

We previously discussed the successor functions $\Diamond$ and $\Box$ as defined in formula's 1 and 2 respectively. Both DFI and FPJ use these functions. First we define the `subst` function which renames occurrences of variable names of $\vec{x}$ to their respective names in $\vec{x}'$. For example:

$$\texttt{subst}(\mathcal{V}(\vec{x})) = \mathcal{V}(\vec{x}') \tag{3}$$

This allows the $\Diamond$ and $\Box$ functions to be implemented symbolically as follows:

$$\Diamond X(\vec{x}) = \{\vec{x} \mid \exists \vec{x}' : \mathcal{E}(\vec{x}, \vec{x}') \wedge \texttt{subst}(X)(\vec{x}')\} \tag{4}$$

$$\Box X(\vec{x}) = \{\vec{x} \mid \forall \vec{x}' : \mathcal{E}(\vec{x}, \vec{x}') \implies \texttt{subst}(X)(\vec{x}')\} \tag{5}$$

Both the existential and universal quantifier are available as a BDD operation in most widely used BDD packages. Thus, no SAT computations are required. Note that the function of equation 5 can also be implemented only using the existential quantifier instead of the universal quantifier. This approach has not been used or evaluated, but could potentially lead to improved performance.

4.2 DFI Implementation

Algorithm 1 outlines a symbolic implementation of DFI. The DFI algorithm uses three data structures. Namely, the current estimation of distraction vertices Z, the mapping $F : V \to \{-, 0, \ldots, d\}$ of vertices to the priority that they are frozen at, and partial strategy $\mathcal{S}$. Z is a subset of V which can easily be modelled by BDD $\mathcal{Z}(\vec{x})$ which determines if vertex $\vec{x}$ is a distraction or not. The function $\mathcal{Z}$ will be updated throughout the runtime of the algorithm to include or exclude particular vertices. $\mathcal{F}$ operates similarly to $\mathcal{P}$, where $\mathcal{F}_{i \in \{0 \ldots d\}}(\vec{x})$ represents all vertices frozen at

priority i, and $\mathcal{F}_-(\vec{x}) = \neg(\mathcal{F}_0(\vec{x}) \vee \ldots \vee \mathcal{F}_d(\vec{x}))$ represents vertices which are not frozen. Partial strategy S is represented by BDD $\mathcal{S}$ which defines a subset of $\mathcal{E}$, of the edges which are good moves for the player that owns that vertex.

Some procedures have been redesigned to better suit a symbolic implementation. Lines 9-16 is the symbolic counterpart of an explicit iteration through the set of vertices X. For each vertex in X, the explicit algorithm uses `onestep` to evaluate a single vertex and computes a good move if available. For symbolic implementations it is desirable to perform such operations on an entire set. For this reason, `onestep` only computes new distractions, but does this for all vertices in X at once (lines 9-12). Then in lines 14-16, old moves from vertices in X are deleted from S, and new moves are added. Note that all correct moves from vertices in X are added to (partial) strategy S whereas explicit DFI only stores one move for each vertex. Computing and storing all correct moves is much easier (and faster) to do symbolically, than it would be explicitly. Additionally, computing only one correct move with BDDs is not desirable because it would require an SAT computation. Finally, the Boolean functions for `onestep`, `even` and `odd` are equivalent to the modal μ-calculus notations of these functions as provided by Van Dijk et al. [19].

More explicit iterations of sets of vertices have been removed for the symbolic DFI algorithm. At lines 18-22 of algorithm 1, the set of vertices X is computed beforehand. Then BDDs $\mathcal{F}_p$ and Z are updated using X. In the explicit algorithm, the set X is explicitly iterated over, updating F and Z with elements of X one by one. A similar adaptation was made for line 25 where the algorithm thaws vertices frozen at lower priorities.

In the empirical evaluation of the symbolic DFI algorithm, two variants of DFI are evaluated. The second DFI implementation, `dfi-ns`, works the same as DFI, but does not compute winning strategies. This algorithm was obtained by removing all occurrences of strategy S from DFI. If we find an advantage in leaving out strategy computation, then `dfi-ns` is useful for realisability problems which do not require strategy derivation.

```
1  def dfi(PG):
2      Z ← ∅
3      F_0 ← ∅, ..., F_d ← ∅
4      S ← ∅
5      p ← 0
6      while p ≤ d do
7          α ← p mod 2
8          X ← P_p ∧ F_− ∧ ¬Z
9          if α is even then
10             Z' ← V ∧ ¬onestep_0(X, Z)
11         else
12             Z' ← onestep_0(X, Z)
13         Z ← Z ∨ Z'
14         S ← (S ∧ ¬X)
15             ∨ (X ∧ V_◇ ∧ E ∧ subst(even(Z)))
16             ∨ (X ∧ V_□ ∧ E ∧ subst(odd(Z)))
17         if Z' ≠ ∅ then
18             X ← V_<p ∧ F_−
19             if α is even then W ← even(Z);
20             else W ← odd(Z);
21             F_p ← (X ∧ ¬W) ∨ F_p
22             Z ← ¬W ∧ Z
23             p ← 0
24         else
25             F_p ← F_p ∧ ¬V_<p
26             p ← p + 1
27     W_◇ ← even(Z)
28     W_□ ← odd(Z)
29     S_◇ ← W_◇ ∧ V_◇ ∧ S
30     S_□ ← W_□ ∧ V_□ ∧ S
31     return W_◇, W_□, S_◇, S_□
32 def onestep_0(X,Z):
33     return (X_◇ ∧ ◇even(Z)) ∨ (X_□ ∧ □even(Z))
34 def even(Z):
35     return (V_0 ∧ ¬Z) ∨ (V_1 ∧ Z)
36 def odd(Z):
37     return (V_0 ∧ Z) ∨ (V_1 ∧ ¬Z)
```

Algorithm 1: The DFI algorithm using BDDs

4.3 FPJ Implementation

The symbolic implementation of FPJ is shown in algorithm 2. Only two data-structures are used. Z represents all vertices which are currently estimated to be won by player *Even*. This is easily replaced by a BDD which defines a subset over V. Justification J is a set of edges. J is modelled very similarly to $\mathcal{E}$, where we define a mapping from $\vec{x}$ to $\vec{x}'$. The set of unjustified vertices is denoted by $U(J)$.

On the procedural side of the symbolic FPJ implementation, there are only few significant adaptations. 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. [15] is based on set-operations. Therefore no major overhauls were required. Only the $\texttt{strategy}_\diamond$ was changed because the explicit algorithm iterates over all vertices in U in order to differentiate behaviour depending on the winner of the vertex, and whether the winner also owns the vertex. This corresponds to the symbolic implementation of lines 36-38 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. Furthermore, line 31 uses a modified version of the $\diamond$ operator with J as its transition relation instead of $\mathcal{E}$.

4.4 Optimisations

A few optimisations were attempted and evaluated 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.

As previously mentioned, the order of application of BDD operations can influence their performance. In lines 8, 18 and 25 of algorithm 1, some BDD (A) is combined with another BDD (B) which is a combination of many smaller BDDs ($B_0 \ldots \mathrm{B}_i$) (i.e. $V_{<p}$ and $\mathcal{F}_-$). In practice, this is done by adding BDDs $B_0 \ldots \mathrm{B}_i$ 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, 20 and 41 percent for lines 8, 18 and 25 of algorithm 1 respectively 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. These optimisations were not applied to FPJ due to time constraints.

In lines 4, 12 and 13 of algorithm 2 we compute of the set $U(J)$. Within each iteration, the result of $U(J)$ is the same, so in practice the result of $U(J)$ on line 4 is passed onto the `next` function and thus used on lines 12 and 13.

```
1  def fpj(PG):
2      Z ← V_0
3      J ← ∅
4      while U(J) ≠ ∅ do
5          Z, J ← next(Z, J)
6      W_◇ ← Z
7      W_□ ← V ∧ ¬Z
8      S_◇ ← J ∧ V_◇ ∧ W_◇
9      S_□ ← J ∧ V_□ ∧ W_□
10     return W_◇, W_□, S_◇, S_□
11 def next(Z, J):
12     p ← min{ p | P_p ∧ U(J) = ∅}
13     U ← P_p ∧ U(J)
14     U_pd ← (phi(Z)ΔZ) ∧ U
15     if U_pd ≠ ∅ then
16         R ← reaches(J, U_pd)
17         if p is even then
18             Z_r ← (Z ∧ ¬R_1) ∧ V_<p
19         else
20             Z_r ← (Z ∨ R_0) ∧ V_<p
21         Z' ← (Z ∧ V_>p) ∨ (Z_pΔU_pd) ∨ Z_r
22         J' ← (J ∧ ¬R) ∨ strategy_◇(Z', U_pd)
23     else
24         Z' ← Z
25         J' ← J ∨ strategy_◇(Z', U)
26     return Z', J'
27 def reaches(J, X):
28     X' ← ∅
29     while X' ≠ X do
30         X' ← X
31         X ← X ∨ (J ∧ ◇X)
32     return X
33 def phi(Z):
34     return (X_◇ ∧ ◇Z) ∨ (X_□ ∧ □Z)
35 def strategy_◇(Z, U):
36     X_0 ← U ∧ V_◇ ∧ Z
37     X_1 ← U ∧ V_□ ∧ ¬Z
38     X_d ← U ∧ ¬(X_0 ∨ X_1) ∧ V
39     Z' ← subst(Z)
40     return (X_0 ∧ Z' ∧ E)
41         ∨ (X_1 ∧ ¬Z' ∧ E)
42         ∨ (X_d ∧ E)
```

Algorithm 2: The FPJ algorithm using BDDs

Furthermore, using `LineProfiler` we found that the order of application of BDD operations on line 36-38 are the most optimal ones.

5. EMPIRICAL EVALUATION

The empirical evaluation aims to study the performance of discussed, symbolic, `dfi`, `dfi-ns` and `fpj` implementations, hopefully giving us a better understanding of the implications of using symbolic data structures and tools to derive winning strategies. We compare the performance of these algorithms to the performance of the symbolic implementation of Zielonka's recursive algorithm [22] (`zlk`) provided by Sanchez et al. [16].

The discussed algorithms are evaluated using the benchmark sets from the *pgame* realisability track of SYNTCOMP 2020. More detailed information on these benchmarks can be found in [10]. All benchmark sets consist of parity automatons which were converted to explicit parity games in `PGSolver` [8] format using Knor[7]. A naive binary encoding was used to create symbolic parity games from these explicit games. As a result, variable ordering of BDDs was not optimised. First all variables $x_0 \dots x_n$ were added, followed by $x'_0 \dots x'_n$.

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, and weights based on number of vertices in each game were not incorporated. The last two columns of table 1 show the average number nodes, and the number of satisfying assignments of the symbolic parity games. 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. The SAT count equals the count of V, three partitions on V (owner, parity, priority) and edges E and therefore equals $4|V| + |E|$ for each parity game. Generally speaking, games that have a vertex count close, or equal to some 2^x have a relatively high compression ratio due to the fact that only few variable assignments of $\vec{x}$ have to be excluded in $\mathcal{V}$ to represent V.

Only benchmarks from formal verification and synthesis problems were used for the empirical evaluation. Random parity games were not used for this empirical evaluation. There is no clear reason as to why randomly generated parity games can be a good representative of parity games used for real model checking and synthesis cases. Randomly generated parity games were only used to provide a reasonable indication of correctness of the parity game algorithms by comparing the winning areas computed by each algorithm, using `zlk` as a source of truth. Note that this does not *prove* the correctness of the evaluated `dfi`, `dfi-ns` and `fpj` implementations. The results of the benchmark sets were also cross-validated in the same manner. Furthermore, derived winning strategies passed a quick sanity check which makes sure that each strategy covers all vertices owned and won by the same player, and that no moves go to the winning area of the other player. This does not guarantee that the strategy is correct, as there exist losing moves to the winning area of the same player, but its a rough indication that no big errors exist. For correctness of the symbolic algorithms, we rely on the correctness of their explicit counterparts and absence of errors in conversion to a symbolic algorithm.

In tables 2 and 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 result for an algorithm that computes a winning strategy is highlighted in green, and the best result without computing a winning strategy is highlighted in blue. Note that the recorded times do not include the time for conversion from the original explicit parity game to symbolic. Furthermore, the explicit parity games were optimised by *Oink* [17]. This includes the removal of self-cycles and trivial cycles, and renumbering priorities (removing gaps in priorities so that all priorities 0...d are used).

As one might expect, `dfi-ns` and `zlk`, which do not compute winning strategies, were almost always faster than their counterparts which do compute winning strategies. The only exception being `fpj` performing better than both

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

Set	#games	avg. n	max. n	priorities	avg. Δ	avg. #bdd nodes	avg. SATs
Lily	23	409	3047	3-8	1.47	1064	2315
AMBA	8	2461	18635	3-4	1.43	1780	13495
ltl2dba	58	921	31717	4-7	1.49	1876	5208
Arbiters	15	2650	20928	4	1.65	6068	15067
Detector	2	98	120	4	1.47	373	533
Buffer	2	14	17	4	1.30	72	72
Load balancer	11	1149	4712	3-8	1.65	2589	6620

Table 1. Statistics of benchmark sets used for empirical evaluation where n is the number of vertices, and Δ is the out degree

Set	strategy				no strategy			
	`dfi`		`fpj`		`dfi-ns`		`zlk`	
Lily	0.37981	(8.1)	0.54191	(10.0)	0.18991	(6.3)	0.38784	(10.8)
AMBA	0.38570	(4.5)	0.69732	(4.0)	0.29235	(3.7)	0.63022	(9.4)
ltl2dba	21.10625	(11.4)	16.12822	(10.5)	14.62706	(12.6)	15.07688	(12.0)
Arbiters	48.16686	(2.8)	23.98140	(2.7)	32.48197	(5.0)	17.53781	(7.8)
Detector	0.00660	(5.3)	0.00682	(4.3)	0.00455	(3.6)	0.00465	(8.5)
Buffer	0.00082	(4.2)	0.00058	(5.6)	0.00052	(5.2)	0.00057	(7.6)
Load-balancer	0.49547	(2.5)	0.90731	(7.1)	0.37213	(13.4)	0.62326	(3.1)

Table 2. Cumulative time in sec. average over 5 runs used to solve all games (left), and average standard deviation of results per parity game (right, in parenthesis) as a percentage of the mean time.

Set	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)

`dfi` *and* `dfi-ns` for the *parameterized arbiter* specifications. More specifically, `fpj` consistently outperformed all other algorithms for the *full arbiter* cases, including one of 20928 vertices. This leads us to believe that mechanisms of the `fpj` algorithm specifically suit these cases very well.

To give an idea of the memory usage of the evaluated algorithms, the peak number of live BDD nodes was recorded for each solve. Memory usage was not explicitly recorded because it is hard to do this accurately without potentially influencing results. Peak live node count is always recorded by CUDD internally, so this information can safely be used. To give a rough indication of memory usage in practice, CUDD reported 16.814536 MB memory usage after constructing BDDs of 49 nodes altogether, and 28.217544 MB after constructing BDDs of 46804 nodes in total. In table 3, we see that `dfi-ns` consistently used the least amount of BDD nodes. More interestingly, `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 can conclude that a symbolic implementation of `dfi` might be a suitable choice when memory-usage must be as low as possible.

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 such 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 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 [19]) 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. DISCUSSION

The empirical evaluation of the implemented algorithms shows that deriving winning strategies for parity games using symbolic solvers is possible. This does come at a cost however. Strategy derivation makes these algorithms a bit slower in comparison to their counterparts. Many computations produce intermediate BDDs which are could be avoided (e.g. preimage, $V_{<p}$, F_{-}). Implementing custom low-level BDD operators to minimise the number of unnecessary intermediate BDDs could further refine symbolic parity game algorithms, improving their performance.

In addition, performance of symbolic parity game algorithms using BDDs could be improved by optimising their encoding. In the empirical evaluation, no advantage was taken of improved variable orderings to more efficiently represent structures and patterns in parity games. More com-

pact symbolic encodings of parity games further improve the memory efficiency of their algorithms. Additionally, more compact BDDs could potentially improve time performance of symbolic algorithms due to smaller input sizes for BDD operations. However, compact symbolic encodings of explicit parity games are hard to obtain. If symbolic parity games are used in practice, it might be desirable to directly convert the source of the specification (e.g. LTL) to a symbolic parity game instead, possibly maintaining structures from the original specification rather than trying to recognise them in explicit games.

During the implementation of the discussed algorithms, (psuedo) random parity games were used for a rough indication of performance and correctness. Initially, `dd`'s pure Python BDD backend was used. Switching to the well-optimised and established CUDD package as the backend for the `dd` package using Cython bindings, resulted in a massive improvement in performance of at least $\times 10$. This shows how both a difference in quality of implementation, and programming language (Python is known to be slower than C/C++ if used properly) can impact the performance of algorithms. Therefore, empirically evaluating our DFI and FPJ implementations by comparing them to parity game solvers like `Oink`[17] or `PGSolver`[8] might not be very valuable.

Another interesting point of discussion is how easily `dfi` can be transformed to `dfi-ns`. Leaving out the computation of winning strategies for DFI is quite trivial because the functionality of DFI does not rely on strategy computations. This makes DFI a bit more versatile because it we can leverage the improved performance of `dfi-ns` when a winning strategy is not needed. On the other hand, FPJ relies on justification J and the `reaches` function to reset lower vertices. As a result, we cannot simply exclude the computation of winning strategies from the algorithm. This would require an entirely different mechanism for resetting lower vertices which most likely results in an entirely different algorithm.

Furthermore, symbolic DFI is relatively memory efficient as found in the empirical evaluation. This is further improved when excluding strategy computation. Zielonka's algorithm performs poorly when it comes to memory usage due to the many copies of subgames. FPJ also uses more memory than DFI in most cases. FPJ's justification graph uses more memory than DFI's strategy because it also stores all outgoing edges of vertices which are lost by their owner. This seems result in more additional memory usage than DFI's set of frozen vertices does. An even more memory efficient version of `dfi-ns` could potentially be obtained by removing the freezing mechanism at the cost of worse time performance.

Finally, a substantial benefit to deriving winning strategies using BDDs was found. It is natural for BDDs to produce a multitude of winning strategies. As a result, we may often obtain multiple variations of a strategy. Some applications of parity games might be able to take advantage of this if it has some preference of structure in a winning strategy. Despite obtaining multiple winning strategies, a single winning strategy can trivially be derived by picking a move from each vertex from the strategy at random.

7. CONCLUSION

This paper has set another step in research on symbolic parity games. More specifically, we studied symbolic parity game algorithms which derive winning strategies. Empirical evaluation showed that strategy derivation for symbolic algorithms is definitely possible, but does introduce some performance drawbacks. We also discovered that there is still a lot of room for improvement for symbolic parity game solvers, as many intermediate BDDs are generated but not used. This leaves potential for improved time and memory efficiency of symbolic parity game algorithms. The next step in research on symbolic parity games is to experiment with pure BDD-based model checking or synthesis tools, avoiding cumbersome and inefficient conversion between explicit and symbolic parity games.

8. REFERENCES

[1] M. Bakera, S. Edelkamp, P. Kissmann, and C. D. Renner. Solving μ-calculus parity games by symbolic planning. In D. A. Peled and M. J. Wooldridge, editors, *Model Checking and Artificial Intelligence*, pages 15–33, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[2] M. Benerecetti, D. Dell'Erba, and F. Mogavero. Solving parity games via priority promotion. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 270–290, Cham, 2016. Springer International Publishing.

[3] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 252–263, New York, NY, USA, 2017. Association for Computing Machinery.

[4] A. Di Stasio, A. Murano, and M. Y. Vardi. Solving parity games: Explicit vs symbolic. In C. Câmpeanu, editor, *Implementation and Application of Automata*, pages 159–172, Cham, 2018. Springer International Publishing.

[5] R. Drechsler and B. Becker. *Binary Decision Diagrams - Theory and Implementation*. Springer, 1998.

[6] R. Drechsler and D. Sieling. Binary decision diagrams in theory and practice. *International Journal on Software Tools for Technology Transfer*, 3(2):112–136, May 2001.

[7] J. Fearnley, S. Jain, B. de Keijzer, S. Schewe, F. Stephan, and D. Wojtczak. An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space. volume 21, pages 325–349, Jun 2019.

[8] O. Friedmann and M. Lange. The pgsolver collection of parity game solvers. *University of Munich*, pages 4–6, 2009.

[9] P. HOU, B. DE CAT, and M. DENECKER. Fo(fd): Extending classical logic with rule-based fixpoint definitions. *Theory and Practice of Logic Programming*, 10(4-6):581–596, 2010.

[10] S. Jacobs, N. Basset, R. Bloem, R. Brenguier, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, T. Michaud, G. Pérez, J.-F. Raskin, O. Sankur, and L. Tentrup. The 4th reactive synthesis competition (syntcomp 2017): Benchmarks, participants results. *Electronic Proceedings in Theoretical Computer Science*, 260:116–143, 11 2017.

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

[12] G. Kant and J. van de Pol. Generating and solving symbolic parity games. In *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering*

(GRAPHITE 2014), Electronic Proceedings in Theoretical Computer Science, pages 2–14. EPTCS, 4 2014. 10.4204/EPTCS.159.2.

[13] J. J. A. Keiren. Benchmarks for parity games. In M. Dastani and M. Sirjani, editors, *Fundamentals of Software Engineering*, pages 127–142, Cham, 2015. Springer International Publishing.

[14] O. Kupferman and M. Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 224–233, 1998.

[15] R. Lapauw, M. Bruynooghe, and M. Denecker. Improving parity game solvers with justifications. In D. Beyer and D. Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 449–470, Cham, 2020. Springer International Publishing.

[16] L. Sanchez, W. Wesselink, and T. A. Willemse. A comparison of bdd-based parity game solvers. *Electronic Proceedings in Theoretical Computer Science*, 277:103–117, Sep 2018.

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

[18] T. van Dijk. Lecture notes on parity games. *Available at:* `http://tvandijk.nl/pdf/2019softwarescience.pdf`, 2019.

[19] T. van Dijk and B. Rubbens. Simple Fixpoint Iteration To Solve Parity Games. In *GandALF*, EPTCS, 2019.

[20] T. van Dijk and J. van de Pol. Multi-core symbolic bisimulation minimisation. *International Journal on Software Tools for Technology Transfer*, 20(2):157–177, Apr 2018.

[21] I. Walukiewicz. Monadic second order logic on tree-like structures. In C. Puech and R. Reischuk, editors, *STACS 96*, pages 399–413, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

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