BDD-based Parity Game Solving

Seminar Formal System Analysis

Lisette Sanchez 0855534

January 29, 2018

Abstract

This paper researches the possibility to make parity game solving algorithms more efficient by using BDDs for their implementation. The paper focuses on the Priority Promotion algorithm provided by Benerecetti et al. [1] and provides an implementation of this algorithm based on BDDs. Test results show that the actual algorithm is much more efficient when using BDDs. However, the time it takes to make the initial BDDs is not efficient yet. Future work can include improving these pre-processing times as well as researching whether other parity game solving algorithms can (also) be made more efficient when implementing them using BDDs.

1 Introduction

A currently widely used technique for model checking problems is the use of parity games [4], where parity games are infinite duration games played by two players, player *odd* and player *even*, on a directed finite graph. The game involves moving a token throughout the graph. The vertices in the graph belong either to player *odd* or player *even*, who can decide the next move if the token ends up in one of their vertices. The vertices also contain priorities, which are used to determine the winner. By representing the model checking problem of a labeled transition system and a modal formula as a parity game, one can check whether or not the modal formula holds for said labeled transition system by solving the parity game and seeing if player *even* wins the desired vertex.

While there are several parity game solving algorithms available, they typically rely on an explicit representation in the form of a graph, where the vertices of the graph are represented in some set representation which grows and shrinks throughout the algorithm. Such graphs however, can grow very large very quickly as well as the set representations of these vertices, making the operations of such algorithms perhaps less efficient. Using binary decision diagrams (BDDs) as a data structure to deal with these set representations can potentially offer a way to make such operations more efficient [2].

This paper considers the Priority Promotion algorithm as presented by Benerecetti, Dell'Erba and Mogavero [1] and presents a way to implement this algorithm using binary decision diagrams as the underlying data structure. Results show that the BDD-based version is indeed much faster, at least for the actual algorithm itself. The pre-processing time needed however to make the initial BDDs is not quite efficient yet and requires more work.

First some preliminaries are provided in Section 2 with some background information on parity games, parity game solving algorithms and binary decision diagrams. Section 3 then provides a summary of the Priority Promotion algorithm of Benerecetti et al. [1] and Section 4 presents a method to implement this algorithm using binary decision diagrams by first

explaining the creation of the initial binary decision diagrams, followed by an elaboration on how the operations of the Priority Promotion algorithm can be performed using this data structure. Section 5 explains the experimental setup, for which the results are listed in Section 6 and a conclusion follows in Section 7.

2 Preliminaries

In this section some background information is provided. This includes an elaboration on parity games, on parity game solving algorithms and on binary decision diagrams (BDDs).

2.1 Parity Games

2.1.1 Definition

Parity games are infinite duration games which are played by two players, player *odd* and player *even* on a directed, finite graph which is also called the *arena*. The formal definition of a parity game is as follows:

Definition 2.1. A parity game is a tuple $PG =< V, E, \Omega, (V_\Diamond, V_\Box) >$ with:

- V is a finite set of vertices.
- $E \subseteq V \times V$ is a total edge relation, i.e., for all $v \in V$ we have $(v, w) \in E$ for some $w \in V$.
- $\Omega : V \to \mathbb{N}$ is the priority function assigning a priority to each vertex.
- $V_\Diamond \subseteq V$ is the set of vertices owned by player *even*.
- $V_\Box \subseteq V$ is the set of vertices owned by player *odd*.

Note that we have $V_\Diamond \cap V_\Box = \emptyset$ and $V_\Diamond \cup V_\Box = V$.

The shape of a vertex determines which player owns it, i.e., a box shaped vertex is owned by player *odd* and a diamond shaped vertex is owned by player *even*. An example of a parity game is depicted in Fig 1 [8], where the number inside a vertex represents the priority of that vertex.

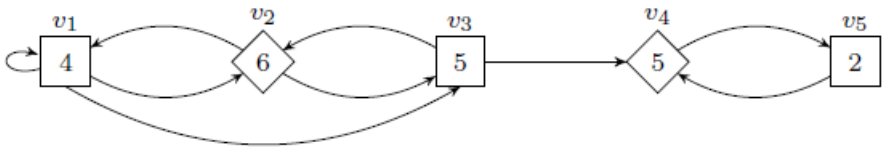


Figure 1: An example of a parity game

The game is played by putting a token on one of the vertices and by letting the players move the token throughout the graph. The player who owns the vertex that currently holds the token, determines where the token moves next. The player can move the token along any outgoing edge of the vertex. For instance, in Fig 1, $v1$ is owned by player *odd* (because it is box shaped), thus when the token resides in $v1$, player *odd* can decide whether the token moves to $v1$, $v2$ or $v3$ (since we have $(v1, v1), (v1, v2), (v1, v3) \in E$).

Let a *path* be an infinite sequence of vertices visited in order and let a *play* be a path chosen by the players. The winner of a game, for a particular *play*, is the parity of the highest priority occurring infinitely often in that *play*, where a parity of 0 means player *even* won and a parity of 1 means player *odd* won. For instance, considering the play that moves between

$v4$ and $v5$ infinitely long is won by player *odd* as the highest priority occurring infinitely often is 5 and the parity of 5 is $5 \equiv_2 1$.

Let a strategy be a function $\sigma : V_\alpha \to V$ such that for all $v \in V_\alpha$ for which $\sigma(v)$ is defined, we have $\sigma(v) \in \{w \mid (v, w) \in E\}$ where α denotes the player, i.e., $\alpha \in \{\Box, \Diamond\}$. A play π is conforming to a strategy if for any $v \in V$ for which v on π and for which $\sigma(v)$ is defined, the edge $(v, \sigma(v))$ is chosen. A strategy is called a winning strategy for player α if any play conforming to this strategy is won by player α.

Intuitively, the players then fight over all the vertices in the graph. In order to determine which player wins a vertex, a game is played where the token is initially placed on this vertex. This vertex is then won by player α if player α has a winning strategy for this game.

2.1.2 Important properties

Some important properties concerning parity games are:

- *A player can always make a move, i.e., for each vertex there is at least one outgoing edge.*
  This follows from the fact that parity games are defined as infinite duration games. If some vertex existed without an outgoing edge, the game would not be infinite.
- *There is always a winner.*
  Since the winner is defined as the parity of the highest priority occurring infinitely often in the *play*, and since parity games are infinite duration games, there will always be at least one vertex occurring infinitely often. If there are more of those vertices, then the one with the highest priority is chosen. Thus there must always be a winner.
- *For every vertex, there is always a way for one of the players to win the game regardless of the opponent's moves.*
  Let $W_\Diamond \subseteq V$ be the set of vertices in V from which player *even* can win regardless of the moves of player *odd*, and let $W_\Box$ be the set of vertices in V from which player *odd* can win regardless of the moves of player *even*. Then $(W_\Diamond, W_\Box)$ partitions V such that $W_\Diamond \cap W_\Box = \emptyset$ and $W_\Diamond \cup W_\Box = V$. The proof for this notion [9] is omitted as it goes beyond the scope of this paper.

2.1.3 Purpose

Though parity games can be used for several purposes, the one that is considered for this paper is the purpose of model checking for modal μ-calculus. Consider a process P and a modal formula M. To determine whether M holds for P, one can create a parity game from the combination of P and M. How this is done exactly goes beyond the scope of this paper [8]. But once this parity game is created, one can use parity game solving algorithms to solve the parity game. If a vertex in the parity game, that represents whether M holds for P, is won by player *even*, then M holds for P, and if it is won by player *odd* then M does not hold for P.

2.2 Parity Game Solving Algorithms

There are multiple algorithms available to solve parity games, such as the one provided by Zielonka [9] which was later improved by Jurdzinski, Paterson, and Zwick [3] and once again by Schewe [5]. However, Jurdzinski also proposed another technique together with Voge [7] which was later improved by Schewe as well [6]. The algorithm considered in this paper is the one provided by Benerecetti et al. [1], called Priority Promotion.

These parity game solving algorithms mainly use two techniques, namely computing $W_\Diamond$

and $W_\Box$ by repeatedly finding and merging dominions (subgames) and computing winning strategies for both players by improving these strategies. Priority Promotion uses the first technique and directly computes $W_\Diamond$ and $W_\Box$. One can then simply look in which of these sets the initial vertex lies to determine who the winner is.

Though this paper focuses only on Priority Promotion, further studies can determine whether other parity games solving algorithms can also be made more efficient by using BDDs.

2.3 BDDs

In this section some background information concerning binary decision diagrams (BDDs) is provided, by explaining what BDDs are and how they are useful for this topic.

2.3.1 Description

A binary decision diagram is a data structure used to concisely and canonically represent boolean formulas. It is a decision tree represented as a directed acyclic graph, where each level of the tree represents one variable of the boolean formula and where the left edge represents the variable above being true, and the right edge represents it being false. An example of a decision tree is shown in Fig 2.

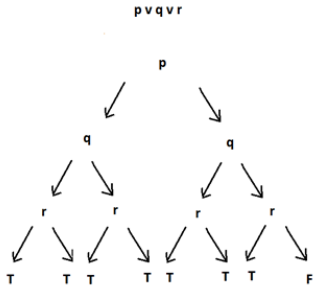


Figure 2: An example of a decision tree

One of the strengths of BDDs is their ability to sometimes greatly reduce the size of a graph, where the size of a graph denotes the number of internal nodes. This size-reduction is implemented by using two reduction techniques, namely *elimination* and *merging*. The merging technique repeatedly merges two nodes with the same variable, the same left subtree and the same right subtree together into one node. This is illustrated in Fig 3, where there are three occurrences of variable r with the same left subtree and the same right subtree merged into one node.

The elimination technique repeatedly replaces, for each variable p for which its left subtree L is equal to its right subtree R, the decision tree with root p with L. This is demonstrated in Fig 4, where in the left subtree of p, the tree with root q can be replaced with T since the left subtree of q is equal to its right subtree. A similar elimination step is performed in the right subtree of p, for the variable r in the left subtree of q.

To turn the decision tree into a decision diagram all *True* nodes are merged into one as well as all *False* nodes, turning the decision tree into a directed acyclic graph, see Fig 5. Applying these techniques to this example shows how the initial decision tree with 7 internal nodes is reduced into a decision diagram with 3 internal nodes.

By representing the sets of a parity game as boolean formulas, BDDs can be used to represent these sets. Another important strength of BDDs is then that set operations such

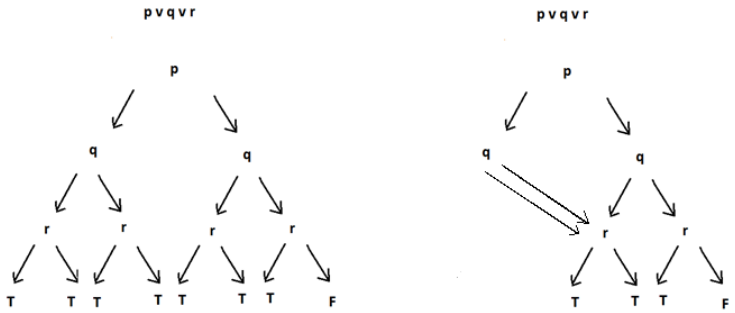


Figure 3: Applying merge to a decision tree: before (left) and after (right)

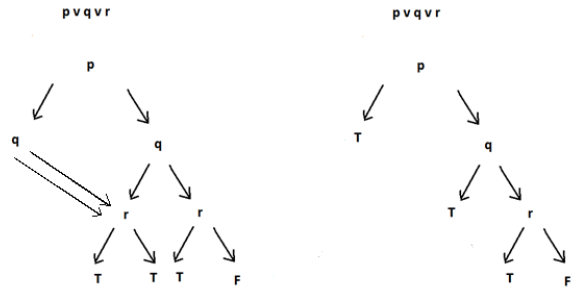


Figure 4: Applying elimination to a decision tree: before (left) and after (right)

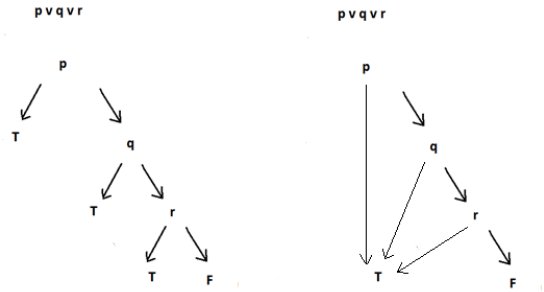


Figure 5: Turning a decision tree into a decision diagram by merging all *True* nodes and all *False* nodes

as intersection and union can be done efficiently on BDDs with a complexity related to the size of the BDDs.

2.3.2 Purpose

Since parity game solving algorithms are mostly based on a graph structure representing the game, and since these graphs can grow exponentially in the worst case, BDDs may be used to concisely represent said graphs. Furthermore, since the operations needed for the Priority Promotion algorithm require basic set operations such as intersection and union, and since these operations can be done efficiently on BDDs, the use of BDDs may prove more efficient for parity game solving when using the Priority Promotion algorithm.

2.3.3 Variable ordering

An important notion for BDDs is that the actual size of a BDD for a given boolean formula depends greatly on the order of the variables, where a variable ordering denotes the order in which the boolean variables occur in the BDD from top to bottom. Where some orderings may result in a relatively small BDD, other orderings for the same boolean formula may result in exponentially large BDDs [2]. Since the complexity of BDD operations depends on the size of BDDs, it is important to try and find variable orderings that lead to minimally sized BDDs.

3 Priority Promotion Algorithm

In this section the Priority Promotion algorithm is presented and explained in more detail. First the global idea is explained, then the algorithm itself is presented and finally an example follows. For more details, the original paper can be consulted [1].

3.1 Global idea

Given a parity game $PG =< V, E, \Omega, (V_\Diamond, V_\Box) >$, the algorithm computes $W_\Diamond$ and $W_\Box$, i.e., the sets of vertices from which player *even* can win regardless of the moves of player *odd* and vice versa, respectively. These sets are created by using the notion of α-dominion and quasi α-dominion, where $\alpha \in \{0, 1\}$ denotes either player *even* or player *odd* respectively. Let $\overline{\alpha}$ denote α's opponent, i.e., if $\alpha = 0$, then $\overline{\alpha} = 1$ and vice versa. The formal definition for an α-dominion is as follows.

Definition 3.1. An α-dominion is a set of vertices $U \subseteq V$, such that player α can enforce a winning play that stays in U from any position in U, i.e., such that the maximal priority visited infinitely often has parity α.

An α-dominion U is called an α-maximal dominion when player α cannot force any other vertex outside U to enter U.

Note that $W_\Diamond$ and $W_\Box$ are thus α-maximal dominions, more precisely an *even*-maximal dominion and an *odd*-maximal dominion respectively. To compute these α-maximal dominions, subsets of these dominions are created first and expanded iteratively. These subsets are called quasi α-dominions, for which the formal definition is as follows.

Definition 3.2. A quasi α-dominion is a set of vertices $U \subseteq V$ such that player α can induce a play from any position in U such that the play is either a winning play that never leaves U or such that the play leaves U at some point.

A quasi α-dominion U is called **α-open** if there is at least one α vertex in U that **must** leave U in one move or if there is at least one $\overline{\alpha}$ vertex that **can** leave U in one move.

A quasi α-dominion U is called **α-closed** if it is not α-open, i.e., if player α can enforce a winning play from any position in U that never leaves U.

It then follows that Q is an α-closed quasi α-dominion $\iff$ Q is α-dominion.

To compute quasi α-dominions, the notion of predecessor set and attractor set are used, for which the formal definitions are as follows.

Definition 3.3. The predecessor set $pre^\alpha(U, V_g)$ of a set of vertices U where V_g denotes the set of vertices in the current subgame, is the union of the set of α-vertices in V_g that **can** enter U in one move and the set of $\overline{\alpha}$-vertices in V_g that **must** enter U in one move.

Formally, $pre^{\alpha}(U, V_g) = \{v \in V_g \cap V_{\alpha} \mid \exists u \in U : (v, u) \in E\} \cup \{w \in V_g \cap V_{\overline{\alpha}} \mid \exists u \in U : (w, u) \in E \ \wedge \ \neg\exists u' \in V_g \backslash U : (w, u') \in E\}$.

Definition 3.4. The attractor set $atr^{\alpha}(U, V_g)$ of a set of vertices U where V_g denotes the set of vertices in the current subgame, is the union of the predecessor sets $pre^{\alpha}(U, V_g)$ calculated repeatedly until no new vertices are added.

Formally, $atr^{\alpha}(U, V_g) = atr^{\alpha}(U, V_g)^m$ for some m where $atr^{\alpha}(U, V_g)^m = atr^{\alpha}(U, V_g)^{m+1}$ with:
$atr^{\alpha}(U, V_g)^0 = U$
$atr^{\alpha}(U, V_g)^{n+1} = atr^{\alpha}(U, V_g)^n \cup pre^{\alpha}(atr^{\alpha}(U, V_g)^n, V_g)$

When a quasi α-dominion U is computed it is necessary to determine whether it is α-open or α-closed in the current parity game. This is done by calculating the escape set of U in V_g for $\overline{\alpha}$. The formal definition of an escape set is as follows.

Definition 3.5. The escape set $esc^{\overline{\alpha}}(U, V_g)$ of a set of vertices U where V_g denotes the set of vertices in the current subgame, is the union of the set of $\overline{\alpha}$-vertices in U that **can** leave U in one move to a vertex in V_g and the set of α-vertices in U that **must** leave U in one move to a vertex in V_g.

Formally, $esc^{\overline{\alpha}}(U, V_g) = \{v \in U \cap V_{\overline{\alpha}} \mid \exists u \in V_g \backslash U : (v, u) \in E\} \cup \{w \in U \cap V_{\alpha} \mid \exists u \in V_g \backslash U : (w, u) \in E \ \wedge \ \neg\exists u' \in U : (w, u') \in E\}$.

Note that this is equivalent to: $esc^{\overline{\alpha}}(U, V_g) = U \cup pre^{\overline{\alpha}}(V_g \backslash U)$.

If the escape set $esc^{\overline{\alpha}}(U, V_g)$ is empty, then U is α-closed in the current subgame, otherwise it is α-open.

Given these notions, the global idea of the algorithm is to, starting with the vertices with the highest priority, create quasi α-dominions until an α-closed quasi α-dominion a.k.a an α-dominion is found. Once this α-dominion is found, its attractor set is calculated which will be a subset of W_{α}. The algorithm is then run again on the remaining set of vertices until all vertices are part of a dominion.

When an α-open quasi α-dominion is found, the algorithm applies its priority promotion and then recurses on a subgame, which will be explained in the next section.

3.2 Algorithm

Given a parity game $PG =< V, E, \Omega, (V_{\diamond}, V_{\square}) >$, find all dominion pairs by running the recursive algorithm below with *findAllDominionPairs(V, Ω, maxPr(V))* where $maxPr(V)$ denotes the max priority in V. Let $domPairs$ be a set of dominion pairs, where one dominion pair contains a dominion and some $\alpha \in \{0, 1\}$, and let $domPairs.dom$ denote the union of all dominions in $domPairs$. Let V_g, Ω_g, p_g denote V, Ω, p in the current subgame g respectively, where initially g is thus PG.

```
procedure FINDALLDOMINIONPAIRS(V, Ω, p)
    domPairs ← FINDDOMINIONPAIR(V, Ω, p)
    while V\domPairs.dom != ∅ do          ▷ If not all vertices are in a dominion yet
        remV ← V\domPairs.dom             ▷ Continue with the remaining vertices
        domPairs ← domPairs ∪ FINDDOMINIONPAIR(remV, Ω, maxPr(remV))
    end while
    return domPairs
end procedure
```

procedure findDominionPair(V_g, Ω_g, p_g)
  $\alpha \leftarrow p_g \bmod 2$
  $U \leftarrow \{v \in V_g \mid \Omega_g(v) = p_g\}$
  $atrC \leftarrow atr_g^{\alpha}(U)$ ▷ Attractor set in subgame
  $escT \leftarrow esc_{PG}^{\overline{\alpha}}(atrC)$ ▷ Escape set in total game
  if $escT$ is empty **then** ▷ Closed in total game
    $atrT \leftarrow atr_{PG}^{\alpha}(atrC)$ ▷ Attractor set in total game
    return $atrT, \alpha$ ▷ α-closed dominion pair in PG
  else ▷ Open in total game
    $escC \leftarrow esc_g^{\overline{\alpha}}(atrC)$ ▷ Escape set in subgame
    if $escC$ is empty **then** ▷ Closed in subgame
      $I \leftarrow \{v \in V \backslash atrC \mid \exists u \in atrC \cap V_{\overline{\alpha}} : (u, v) \in E\}$
      $p^* \leftarrow minPr(I, \Omega_g)$
      for $u \in atrC$ **do**
        $\Omega^*(u) = p^*$ ▷ Promote priority
      end for
      for $u' \in V \backslash atrC$ **do**
        if $\Omega_g(u') < p^*$ **then**
          $\Omega^*(u') = \Omega(u')$ ▷ Reset priority
        else
          $\Omega^*(u') = \Omega_g(u')$ ▷ Keep priority as it is
        end if
      end for
      $V^* \leftarrow \{v \in V \mid \Omega^* \leq p^*\}$
      findDominionPair(V^*, Ω^*, p^*)
    else ▷ Open in subgame
      for $u \in atrC$ **do**
        $\Omega^*(u) = p_g$ ▷ Promote priority
      end for
      for $u' \in V_g \backslash atrC$ **do**
        $\Omega^*(u') = \Omega_g(u')$ ▷ Keep priority as it is
      end for
      $p^* \leftarrow maxPr(\Omega^*)$ with $p^* < p_g$
      $V^* \leftarrow V_g \backslash atrC$
      findDominionPair(V^*, Ω^*, p^*)
    end if
  end if
end procedure

3.3 Example

The algorithm is now demonstrated by means of an example. Consider the parity game as displayed in Fig 6, with $V_\diamond = \{b, c, g\}$ and $V_\square = \{a, d, e, f\}$.

Starting with the vertices with the highest priority, the algorithm starts with $p_g = 6$ and $U = \{b\}$. The even attractor set of U is $\{a, b, c\}$ as shown in Fig 7a by orange line indicators, since $\alpha = 6 \equiv_2 0$ and since a is a $\overline{\alpha}$ vertex that **must** move into U and c is an α vertex that **can** move into U. The remaining $\overline{\alpha}$ vertices d, e, f do not necessarily need to move into U and the remaining α vertex g cannot move into U. Thus then $U = \{a, b, c\}$.

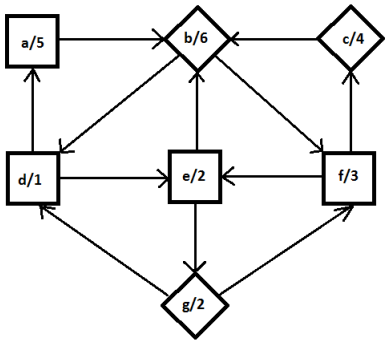
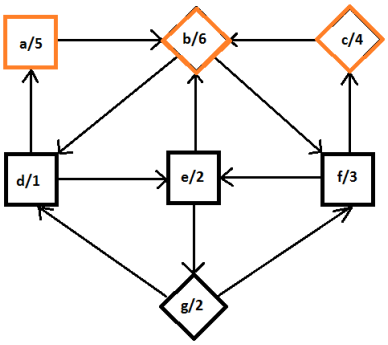
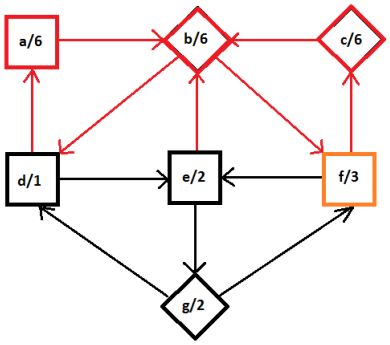


Figure 6: An example of a parity game



(a) Attractor set of $U = \{a, b, c\}$



(b) Applying step 6(a) of the algorithm

Figure 7: An example of a parity game

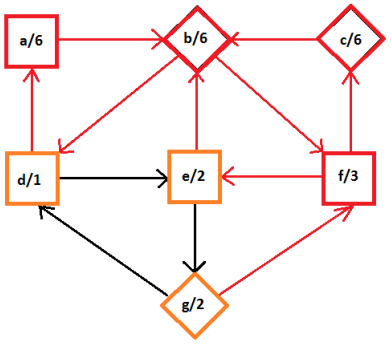
Since the escape set contains vertex b, as that is an α vertex that must leave U in its next move, U is open in the total game and open in the subgame, which at this point are similar games. Thus step *6(a)* of the algorithm is executed. The priority function is updated by promoting the priorities of the vertices in U to 6, the new max priority is 3 and the vertices in U and their edges are no longer considered in the following subgame (as indicated by red lines), see Fig 7b.

Now $U = \{f\}$ and its odd attractor set remains the same as $\alpha = 3 \equiv_2 1$ and there are no α vertices that **can** move into U and no $\overline{\alpha}$ vertices that **must** move into U. The escape set of U contains f since it has to move out of U in its next move. Since f can move to a vertex in the subgame, namely e, U is both open in the subgame as in the total game, hence again step *6(a)* is executed. Since there is only one vertex in U, priority promotion brings no change and the next subgame continues without vertices a, b, c and f, see Fig 8a.

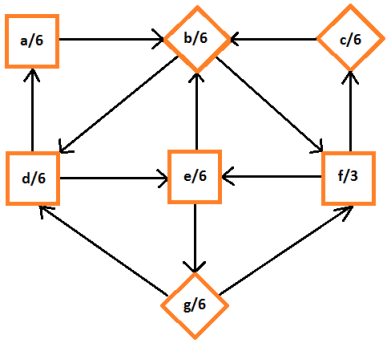
The new max priority is then 2, giving $U = \{e, g\}$ and its even attractor set $U = \{d, e, g\}$ since $\alpha = 2 \equiv_2 0$ and d is a $\overline{\alpha}$ vertex that **must** move into U (in this subgame). The escape set of U is empty in the subgame, but not empty in the total game where it contains a and b. Hence step *6(b)* is executed, where $I = \{a, b\}$ and thus $p^* = 6$. The priorities of the vertices in U are then promoted to 6, while the rest remains the same. Also, since $p^* = 6$, the new subgame contains again all vertices in the total game.

Having $U = \{a, b, c, d, e, g\}$, the even attractor set of U then contains all vertices in the total game as shown in Fig 8b, since $\alpha = 6 \equiv_2 0$ and f is a $\overline{\alpha}$ vertex that **must** move into

U. Since U contains all vertices in the total game, it is obviously closed in the total game and thus this set is returned as $W_\diamond$. In other words, $W_\square$ is empty, meaning player *even* can always win this game no matter where it starts and no matter the moves of player *odd*.



(a) New attractor set $U = \{d, e, g\}$



(b) New attractor set contains all vertices

Figure 8: An example of a parity game

4 BDD-based Priority Promotion Algorithm

In this section the adaptation of the Priority Promotion algorithm is presented and explained in detail by first elaborating on which BDDs will be created and how, followed by a description of how the Priority Promotion operations are implemented using BDDs.

4.1 BDD creation

For the purpose of BDD-based Priority Promotion, several BDDs are initially created. Given a parity game $PG =< V, E, \Omega, (V_\diamond, V_\square) >$, two BDDs are created for V, one BDD each for E, $V_\diamond$ and $V_\square$ as well as two BDDs per priority in Ω, namely PO_p and PC_p for each p in Ω, where PO_p represents the set of vertices in V that originally have priority p and PC_p represents the set of vertices in V that currently have priority p, i.e., in the current subgame. The distinction between these two BDDs per priority is necessary in order to keep track of both the original priority function Ω and the current priority function Ω_g. For a similar reason two BDDs are created for V, namely VO and VC to represent the vertices in the original game and the vertices in the current game respectively.

For VO, VC, $V_\diamond$, $V_\square$ and for the BDDs per priority, the idea is similar, namely to represent each vertex by a boolean formula and by taking the disjunction of these formulas. Representing each vertex by a boolean formula is done by using binary encoding. Given a set of boolean variables, each possible combination of these variables can be used to represent one vertex. The number of boolean variables needed to represent a vertex depends on the total number of vertices, since n boolean variables have 2^n possible combinations and can thus represent a maximum of 2^n vertices.

For example, the parity game provided in Section 3.3 consists of 7 vertices. It would then suffice to use 3 boolean variables. Considering 3 such variables x, y, z, a possible binary encoding for the vertices in this parity game is shown in Fig 9. Which boolean formula is assigned to which vertex has an effect on the speed of the algorithm. Adjacent vertices preferably differ by as few bits as possible, but this is not always possible, especially not when there are cycles.

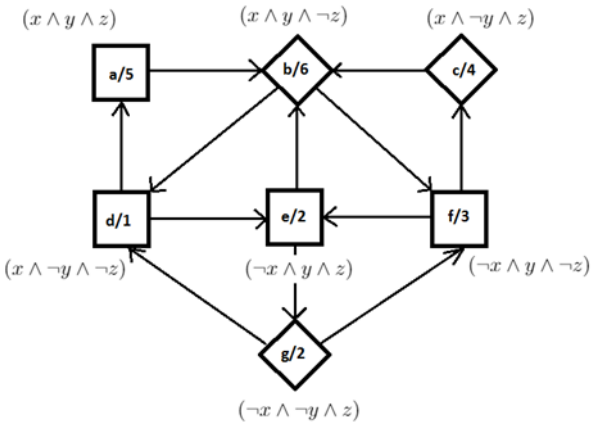


Figure 9: Binary encoding of a parity game

For this binary encoding, a BDD can now easily be created as follows: $VO = VC = \{(x \wedge y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z)$. Note that when all possible combinations of the variables are used, i.e., in this case if all 8 combinations for x, y, z would be used, then the BDD would simply have one vertex named *True*. This is not the case here however, since there are only 7 vertices in the parity game under consideration.

For E a different technique is used to create a boolean formula for the BDD. First, a copy is created for each boolean variable and for each vertex. So, continuing with the example above, 6 boolean variables are used for E namely x, y, z, x', y' and z'. Note that using the following variable ordering is generally more efficient than the previous one: x, x', y, y', z, z'. The copy of each vertex is similar to its original, except that each occurrence of x, y, z is replaced by x', y', z' respectively. For instance, the copy of vertex $(\neg x \wedge y \wedge \neg z)$ is $(\neg x' \wedge y' \wedge \neg z')$.

For each vertex v an implication is created where the left-hand side of the implication is the original representation of v and the right-hand side of the implication is the disjunction of the copies of all vertices that v can move to in one move. The entire boolean formula for E is then the conjunction of all above-mentioned implications. Note that, for every vertex representation not used, an implication to *False* must be added in order to create a correct boolean formula for E.

Following the previous example, the resulting boolean formula for E is as follows:

$$
\begin{aligned}
&(x \wedge y \wedge z) \Longrightarrow (x' \wedge y' \wedge \neg z') \wedge \\
&(x \wedge y \wedge \neg z) \Longrightarrow ((x' \wedge \neg y' \wedge \neg z') \vee (\neg x' \wedge y' \wedge \neg z')) \wedge \\
&(x \wedge \neg y \wedge z) \Longrightarrow (x' \wedge y' \wedge \neg z') \wedge \\
&(x \wedge \neg y \wedge \neg z) \Longrightarrow ((x' \wedge y' \wedge z') \vee (\neg x' \wedge y' \wedge z')) \wedge \\
&(\neg x \wedge y \wedge z) \Longrightarrow ((x' \wedge y' \wedge \neg z') \vee (\neg x' \wedge \neg y' \wedge z')) \wedge \\
&(\neg x \wedge y \wedge \neg z) \Longrightarrow ((x' \wedge \neg y' \wedge z') \vee (\neg x' \wedge y' \wedge z')) \wedge \\
&(\neg x \wedge \neg y \wedge z) \Longrightarrow ((x' \wedge \neg y' \wedge \neg z') \vee (\neg x' \wedge y' \wedge \neg z')) \wedge \\
&(\neg x \wedge \neg y \wedge \neg z) \Longrightarrow False
\end{aligned}
$$

How these initially created BDDs are maintained throughout the algorithm is explained in the next section.

4.2 BDD-based operations

Considering the algorithm as depicted in Section 3.2, the following operations are reviewed: computing initial U, computing the attractor set of U, computing the escape set of U, computing Ω^*, computing V^* and computing I.

Computing initial U

Step 1 of the Priority Promotion algorithm is to take U to be the set of all vertices in V_g with priority p_g. This operation can simply be done by taking $VC \cap PC_{p_g}$ which contains all the vertices in V_g that currently have the required priority p_g.

Computing the attractor set of U

To compute the attractor set $atr_g^{\alpha}(U)$ of a set of vertices U in parity game g, the implementation provided by Ioannis Filippidis on GitHub at `https://github.com/johnyf/dd` it used. This implementation uses existential quantification, another set operation that can be done efficiently on BDDs. Applying an existential quantification for a variable x on a BDD Z means to find the BDD Z' such that there exists an assignment for x for which Z' is satisfied, regardless of the assignments of other variables. For example, consider the BDD in Fig 10, where the existential quantification of r in Z results in the BDD Z' containing only *True*. This means that there exists some assignment for r such that the boolean formula *True* is satisfied regardless of the assignments for p and q.

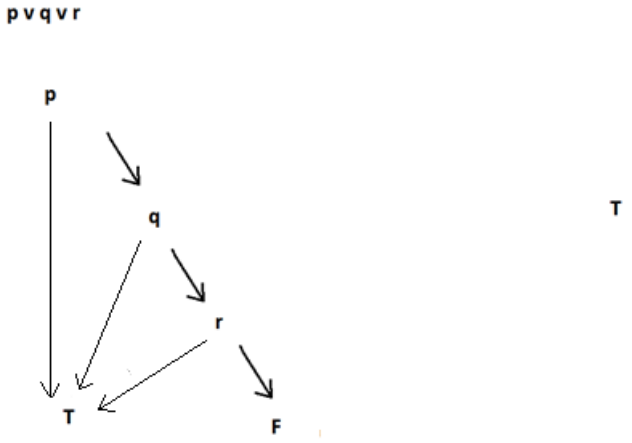


Figure 10: BDD Z (left) and BDD Z' with existential quantification for r in Z (right)

Let $Q[u \to u']$ denote the BDD that results from replacing all boolean variables in BDD Q with their primed counterparts, which are introduced in the previous section and vice versa for $Q[u' \to u]$. Let $EQ(w, u)$ denote the existential quantification of u in w. Furthermore let E denote the BDD representing the edge relation E and let U denote the BDD representing the vertices in U for which the backwards analysis is requested. Computing the attractor set of U is then done as follows.

First the predecessor set is computed. For this, first the intersection of the BDDs E and $U[x \to x']$ is taken, which results in a BDD showing which moves in E end in U and which do not. Using existential quantification for all primed variables x' on this new BDD another BDD is returned, which represents a set P of vertices for which there exists a move to a vertex in U. This computed set P however does not yet meet the requirements as specified in

Definition 3.3. Hence from P first all $\overline{\alpha}$ vertices are removed for which there is another vertex $u \in V \backslash U : (v, u) \in E$, resulting in the correct predecessor set. The algorithm then continues with the next predecessor set as specified in Definition 3.4 (See pseudocode below).

```
procedure GETATR(U, V)
    q ← U
    q_old ← False
    while q != q_old do                ▷ While new vertices are still being added continue
        q_old = q
        predQ ← GETPRED(q)
        a ← V_α ∩ predQ
        predVnotU ← GETPRED(V\U)
        na ← V_ᾱ\predVnotU
        predQ ← (na ∪ a) ∩ V           ▷ Make sure to only get vertices in the current game
        q = q ∪ predQ ∪ U
    end while
    return q
end procedure
procedure GETPRED(U)
    nextZ ← U[u → u']
    w ← E ∩ nextZ
    predU ← EQ(w, u')
    return predU
end procedure
```

Computing the escape set of U

Since by Definition 3.5 it is given that $esc^{\overline{\alpha}}(U, V_g) = U \cup pre^{\overline{\alpha}}(V_g \backslash U)$ the same technique is applied to compute the escape set $esc^{\overline{\alpha}}(U, V_g)$ as for computing the attractor set.

Computing Ω^*

To compute Ω^*, several operations are required depending on whether the set U is α-open or α-closed in the subgame.

If U is α-open, the following operations are needed: for all vertices u in U, $\Omega^*(u) = p_g$ and for all vertices u' in $V \backslash U$, $\Omega^*(u') = \Omega_g(u')$. This can simply be done by taking $PC_{p_g} = PC_{p_g} \cup U$ and $PC_p = PC_p \backslash U$ for all priorities p in Ω such that $p \neq p_g$ (See pseudocode below).

```
...
PC_pg ← PC_pg ∪ U
for p ∈ Pr : p != p_g do
    PC_p = PC_p\U
end for
...
```

If U is α-closed the following operations are needed: for all vertices u in U, $\Omega^*(u) = p^*$, for all vertices u' in $V \backslash U$ with $\Omega_g(u') < p^*$, $\Omega^*(u') = \Omega(u')$ and for all vertices u'' in $V \backslash U$ with $\Omega_g(u'') \geq p^*$, $\Omega^*(u'') = \Omega_g(u'')$. This can be done in a few simple steps. First of all let T be the union of all PC_p for $p < p^*$, followed by for all $p < p*$, $PC_p = (PO_p \cap T) \backslash U$, and finally $PC_{p^*} = PC_{p^*} \cup U$ (See pseudocode below).

```
...
T ← ⋃_{p<p*} PC_p
for p ∈ Pr : p < p* do
    PC_p = (PO_p ∩ T)\U
end for
PC_{p*} ← PC_{p*} ∪ U
...
```

Computing V^*

There are two possible operations for computing V^*, namely V^* is the set of vertices in $V_g \backslash U$ and V^* is the set of vertices v in V such that $\Omega^*(v) \leq p^*$. In the first case the computation can be done by taking $VC = VC \backslash U$ and in the second case by taking $VC = \bigcup_{p \leq p^*} PC_p$. Since computing Ω^* precedes computing V^*, PC_p will already be properly updated for this operation, for all $p \leq p^*$.

Computing I

For this operation, the same implementation is used as for computing the attractor set and the escape set, but it is slightly altered to find a 'successor' set rather than the predecessor set.

First the intersection is created of E and U. Then existential quantification for all original variables u is used to return a BDD R showing for which vertices a move is possible starting from a vertex in U. These vertices are still represented in their primed form however, so they need to be translated back to their original form by doing $R[u' \to u]$. To compute the set I, this method is applied to $V_{\overline{\alpha}} \cap U$ and from the result all vertices are extracted that are in the total game but not in U. The pseudocode to compute I is shown below.

```
...
i ← getSuc(V_ᾱ ∩ U)
I ← (i ∩ V)\U
...
procedure getSuc(U)
    w ← E ∩ U
    sucU ← EQ(w, u)                    ▷ Existential quantification of u in w
    sucU ← sucU[u' → u]
    return sucU
end procedure
```

5 Experimental setup

In this section a description is given on how the difference is tested between the original Priority Promotion and the BDD-based version.

Both algorithms are first implemented using Python. The code for both implementations can be found in the appendix. Several test cases are created in the form of parity games. These test cases differ in the number of vertices, the number of priorities and the number of boolean variables as those parameters are assumed to have an effect on the efficiency of BDD-based Priority Promotion. These test cases are run on the original Priority Promotion algorithm and on the BDD-based version. The results are evaluated on correctness and on

speed, where a difference in speed is monitored between the time needed for pre-processing the data and the time needed for executing the algorithm. Some details on the test cases are provided in Table 1. The first two test cases are added to the Appendix, whilst the latter two are omitted due to readability and size. These test cases are added to the paper as text files as well as the text files for all test cases containing the format used for the BDD version.

Table 1: Details on the test cases

	#vertices	#priorities	#Booleans
mini1	9	7	8
mini2	7	6	6
chatbox1	98305	2	36
chatbox2	163840	3	36

6 Results

In this section the results of the experiments are provided and discussed.

The results of running both versions of the algorithm on the test cases is shown in Table 2, where for each test case the result is mentioned as well as the number of dominion pairs that have been found and the number of recursive calls that were made. Note that in all cases all vertices are won by player even but that chatbox2 required running the algorithm twice to find two separate dominions both for player even. One can clearly see that both algorithms return the same results when run on the test cases, as is desired, since other results would most likely indicate flaws in the implementation.

Table 2: Results for the test cases on outcome

	PP			BDD-based PP		
	Result	#dom.	#rec. calls	Result	#dom.	#rec. calls
mini1	all even	1	19	all even	1	19
mini2	all even	1	4	all even	1	4
chatbox1	all even	1	1	all even	1	1
chatbox2	all even	2	4	all even	2	4

The performance results of running both versions of the algorithm on the test cases is shown in Table 3. A distinction is made between the time it takes to pre-process the data and the time it takes to run the actual algorithm, where the time is listed in seconds. For the original algorithm the pre-processing involves parsing the data from a text files into sets and for the BDD-based version the pre-processing involves turning some hardcoded data into actual BDDs using the library provided by Ioannis Filippidis on GitHub at `https://github.com/johnyf/dd`.

As one can see from the results in Table 3 the BDD-based version performs better when considering larger parity games. The majority of the time however is consumed by pre-processing of the data, i.e., creating the actual BDDs. These pre-processing times however may differ per computer and also per implementation, meaning that the BDD-based version of Priority Promotion is certainly worth pursuing more. If this pre-processing time can also

Table 3: Results for the test cases on performance

	PP			BDD-based PP		
	Total	Prep	Alg	Total	Prep	Alg
mini1	0.016	0.0	0.016	1.638	0.874	0.764
mini2	0.016	0.0	0.016	0.827	0.671	0.156
chatbox1	10.888	4.555	6.333	7.316	7.051	0.265
chatbox2	45.645	8.330	37.315	12.761	12.121	0.640

be reduced significantly, the BDD-based version would provide an even bigger improvement over the original version, at least when considering larger priority games.

7 Conclusion

After implementing the original Priority Promotion algorithm as well as its BDD-based version and after running several test cases differing in number of vertices, number of priorities and number of Booleans, the conclusion can be drawn that the BDD-based version runs much faster than the original version, when considering larger parity games. Since the bulk of the runtime for this version is consumed by the pre-processing of the data, future work can determine whether this pre-processing time can also be reduced significantly, thereby rendering the BDD-based version of Priority Promotion even more efficient than the original algorithm.

Furthermore, since the results for the BDD-based version of Priority Promotion indicate a potential great improvement over the original version, future work may also include testing other parity game solving algorithms for their potential for increased efficiency when implemented by using BDDs.

References

[1] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Solving parity games via priority promotion. In *CAV (2)*, volume 9780 of *Lecture Notes in Computer Science*, pages 270–290. Springer, 2016.

[2] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[3] Marcin Jurdzinski, Mike Paterson, and Uri Zwick. A deterministic subexponential algorithm for solving parity games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.

[4] Andrzej Mostowski. Games with forbidden positions. Technical Report 78, University of Gdansk, 1991.

[5] Sven Schewe. Solving parity games in big steps. In *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 449–460. Springer, 2007.

[6] Sven Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *CSL*, volume 5213 of *Lecture Notes in Computer Science*, pages 369–384. Springer, 2008.

[7] Jens Vöge and Marcin Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 202–215. Springer, 2000.

[8] Tim A. C. Willemse and Maciej Gazda. Parity games. In *Encyclopedia of Algorithms*, pages 1532–1537. 2016.

[9] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.

A Original Priority Promotion implementation

```
from time import process_time

# main loop
def findDominionPair(Psg, r, p) :
    global remainingVertices, countPairs, countCalls
    countCalls = countCalls + 1
    # get initial V
    V = getV(Psg, r, p)
    alpha = p%2
    if alpha == 0 :
        alphaSet = Ps0
        notAlphaSet = Ps1
    else :
        alphaSet = Ps1
        notAlphaSet = Ps0
    # get attractor set of V
    V = getAtr(V, alphaSet, notAlphaSet, Psg)

    #if closed in total game
    if isClosedInGame(V, alphaSet, notAlphaSet, Ps) :
        #print("closed in total game")
        atrV = getAtr(V, alphaSet, notAlphaSet, Ps)
        #print("found dominion : ", atrV, " for player: ", alpha);
        print("found dominion for player: ", alpha);
        if len(atrV) < remainingVertices :
            remainingVertices = remainingVertices - len(atrV)
            countPairs = countPairs + 1
            for v in atrV :
                Ps[v] = False
            newP = 0
            for v in range(len(Ps)) :
                if Ps[v] == True and pr[v] > newP :
                    newP = pr[v]
            findDominionPair(Ps[:], pr[:], newP)
        else :
            print("algorithm completed")
        return
    #if closed in subgame
    elif isClosedInGame(V, alphaSet, notAlphaSet, Psg) :
        ##get I
        I = set()
        for v in V.intersection(notAlphaSet) :
            for s in succ[v].difference(V) :
                if Ps[s] == True :
                    I.add(s)
```

```
        #get min priority in I
        newP = getMinPr(I, r)
        #get new region function prNew
        prNew = r[:]
        for id in range(len(pr)) :
            if id in V :
                prNew[id] = newP
            elif r[id] != None and r[id] < newP :
                prNew[id] = pr[id]
        #get new set of vertices psgNew
        psgNew = Ps[:]
        for v in range(numVertices) :
             if prNew[v] != None and prNew[v] > newP :
                 psgNew[v] = False
    #if open in subgame
    else :
        #get new region function prNew
        prNew = r[:]
        for id in range(len(pr)) :
            if id in V :
                prNew[id] = p
            else :
                prNew[id] = r[id]
        # get new max priority
        newP = 0;
        for prio in prNew :
            if prio != None and prio > newP and prio < p :
                newP = prio
        #get new set of vertices psgNew
        psgNew = Psg
        for v in V :
             psgNew[v] = False
    return findDominionPair(psgNew, prNew, newP);


# Get the min priority of a set of nodes given a region function
def getMinPr(I, r) :
    min = max(pr)
    for i in I :
        if r[i] < min :
            min = r[i]
    return min;

# Get the original set V given a set of vertices, ...
# ...a region function and a priority
def getV(Psx, r, p) :
    V = set()
```

```
    for x in range(numVertices) :
        # if the current vertex is in the subgame ...
        # ...and its priority is p, add it to V
        if Psx[x] == True and r[x] == p :
            V.add(x)
    return V;

# Get the attractor set of V, given a set of vertices and alpha
def getAtr(V, alphaSet, notAlphaSet, Psx) :
    Vold = V
    Vnew = getPre(Vold, alphaSet, notAlphaSet, Psx)
    #print("update: ", Vnew)
    while Vold != Vnew :
        Vold = Vnew
        Vnew = getPre(Vold, alphaSet, notAlphaSet, Psx)
        #print("update: ", Vnew)
    return Vnew;

# Get the predecessor set of V, given a set of vertices and alpha
def getPre(V, alphaSet, notAlphaSet, Psx) :
    Vpre = set()
    for x in alphaSet :
        if Psx[x] == True:
            for sx in succ[x] :
                    if Psx[sx] == True and sx in V:
                        Vpre.add(x)
                        break
    for x in notAlphaSet :
        if Psx[x] == True :
            empty = True
            subset = True
            for sx in succ[x] :
                if Psx[sx] == True and sx not in V:
                    empty = False
                    subset = False
                    break
                elif Psx[sx] == True :
                    empty = False
            if subset and not empty :
                Vpre.add(x)
    return V|Vpre ;

# Check if (V, alpha) is closed in game Ps
def isClosedInGame(V, alphaSet, notAlphaSet, Psx) :
    for x in alphaSet.intersection(V) :
        oneInPs = False
        oneInV = False
```

```
        for sx in succ[x] :
            if Psx[sx] == True :
                oneInPs = True
                if sx in V:
                    oneInV = True
                    break
        if oneInPs and not oneInV :
            return False;
    for x in notAlphaSet.intersection(V) :
        for sx in succ[x] :
            if Psx[sx] == True and not sx in V :
                return False;
    return True;


startCreating = process_time()
# read parity game document
pGFile = open("D:\Documents\School\Jaar_5\Q2_Seminar\PG\chatbox2.pgsolver", "r"

# initialize all required data structures
Lines = pGFile.readlines()[1:]
numVertices = len(Lines)
Ps = list()
Ps0 = set()
Ps1 = set()
Pr = set()
pr = [None] * numVertices
succ = list()
numPriorities = len(Pr)

# parse data from file into data structures
for x in range(numVertices):
    vertex = Lines[x].split(" ")
    Ps.append(True)
    Pr.add(int(float(vertex[1])))
    pr[x] = int(float(vertex[1]))
    if vertex[2] == "0" :
        Ps0.add(int(float(vertex[0])))
    else :
        Ps1.add(int(float(vertex[0])))
    succX = set()
    for s in range(3, len(vertex)-1)  :
        succX.add(int(float(vertex[s][:-1])))
    succX.add(int(float(vertex[len(vertex)-1][:-2])))
    #add this set to the list of successors per vertex
    succ.append(frozenset(succX))
remainingVertices = len(Ps)
```

```
countPairs = 1;
countCalls = 0;

# run the algorithm
startAlg = process_time()
findDominionPair(Ps[:], pr[:], max(Pr))
end = process_time()
print("number of dominion pairs: ", countPairs)
print("number of recursive calls: ", countCalls)
print("total time elapsed: ", end - startCreating)
print("time needed for parsing data: ", startAlg - startCreating)
print("time needed for algorithm: ", end - startAlg)
```

B BDD-based Priority Promotion implementation

```
from dd import autoref as _bdd
from copy import deepcopy
from time import process_time

startCreating = process_time()

#create BDD's
Var = _bdd.BDD()
Var.declare("x", "x'", "y", "y'", "z", "z'", "w", "w'")
s = ("(~x /\ y /\ z /\ w) \/ "
     "(x /\ y /\ z /\ w) \/ "
     "(x /\ y /\ z /\ ~w) \/ "
     "(x /\ y /\ ~z /\ ~w) \/"
     "(~x /\ ~y /\ z /\ w) \/"
     "(x /\ ~y /\ z /\ w) \/ "
     "(x /\ ~y /\ z /\ ~w) \/"
         "(x /\ ~y /\ ~z /\ ~w) \/"
         "(~x /\ ~y /\ ~z /\ ~w)")
VO = Var.add_expr(s)
VC = Var.add_expr(s)

t = ("(~x /\ y /\ z /\ w) \/ "
     "(x /\ y /\ z /\ w) \/"
     "(~x /\ ~y /\ z /\ w) \/"
     "(x /\ ~y /\ z /\ w) \/ "
     "(x /\ ~y /\ ~z /\ ~w) \/"
         "(~x /\ ~y /\ ~z /\ ~w)")
V_odd = Var.add_expr(t)
V_even = VO & ~ V_odd

PO_0 = Var.add_expr("(x /\ ~y /\ z /\ w)")
```

```
PC_0 = Var.add_expr("(x /\ ~y /\ z /\ w)")
PO_1 = Var.add_expr("(x /\ y /\ ~z /\ ~w) \/ (x /\ ~y /\ ~z /\ ~w)")
PC_1 = Var.add_expr("(x /\ y /\ ~z /\ ~w) \/ (x /\ ~y /\ ~z /\ ~w)")
PO_2 = Var.add_expr("(~x /\ y /\ z /\ w) \/ (~x /\ ~y /\ ~z /\ ~w)")
PC_2 = Var.add_expr("(~x /\ y /\ z /\ w) \/ (~x /\ ~y /\ ~z /\ ~w)")
PO_3 = Var.add_expr("(x /\ y /\ z /\ ~w)")
PC_3 = Var.add_expr("(x /\ y /\ z /\ ~w)")
PO_4 = Var.add_expr("(~x /\ ~y /\ z /\ w)")
PC_4 = Var.add_expr("(~x /\ ~y /\ z /\ w)")
PO_5 = Var.add_expr("(x /\ ~y /\ z /\ ~w)")
PC_5 = Var.add_expr("(x /\ ~y /\ z /\ ~w)")
PO_6 = Var.add_expr("(x /\ y /\ z /\ w)")
PC_6 = Var.add_expr("(x /\ y /\ z /\ w)")

# create dictionary for priorities
curPr = {0 : PC_0, 1 : PC_1, 2 : PC_2, 3 : PC_3, 4 : PC_4, 5 : PC_5, 6 : PC_6}
origPr = {0 : PO_0, 1 : PO_1, 2 : PO_2, 3 : PO_3, 4 : PO_4, 5 : PO_5, 6 : PO_6}

u = ("((~x /\ y /\ z /\ w) => ((~x' /\ y' /\ z' /\ w') \/ (~x' /\ ~y' /\ z' /\
     "((x /\ y /\ z /\ w) => ((~x' /\ y' /\ z' /\ w') \/ (x' /\ y' /\ z' /\ ~w'
     "((x /\ y /\ z /\ ~w) => ((x' /\ ~y' /\ z' /\ ~w') \/ (x' /\ y' /\ ~z' /\
     "((x /\ y /\ ~z /\ ~w) => ((x' /\ y' /\ ~z' /\ ~w') \/ (x' /\ y' /\ z' /\
     "((~x /\ ~y /\ z /\ w) => ((~x' /\ y' /\ z' /\ w') \/ (x' /\ ~y' /\ z' /\
     "((x /\ ~y /\ z /\ w) => ((x' /\ ~y' /\ z' /\ w') \/ (x' /\ y' /\ z' /\ w'
     "((x /\ ~y /\ z /\ ~w) => ((x' /\ ~y' /\ z' /\ w') \/ (x' /\ y' /\ z' /\ ~w
          "((x /\ ~y /\ ~z /\ ~w) => ((x' /\ ~y' /\ z' /\ ~w') \/ (~x' /\ ~y' /\
     "((~x /\ ~y /\ ~z /\ ~w) => (((~x' /\ ~y' /\ z' /\ ~w') \/ (x' /\ ~y' /\ ~
          "((x /\ y /\ ~z /\ w) => FALSE) /\ "
          "((x /\ ~y /\ ~z /\ w) => FALSE) /\ "
          "((~x /\ y /\ z /\ ~w) => FALSE) /\ "
          "((~x /\ y /\ ~z /\ w) => FALSE) /\ "
          "((~x /\ y /\ ~z /\ ~w) => FALSE) /\ "
          "((~x /\ ~y /\ z /\ w) => FALSE) /\ "
          "((~x /\ ~y /\ ~z /\ w) => FALSE)")
E = Var.add_expr(u)

unprime = {"x'": "x", "y'": "y", "z'": "z", "w'" : "w"}
prime = {"x": "x'", "y": "y'", "z": "z'", "w" : "w'"}
qvars = {"x'", "y'", "z'", "w'"}
yvars = {"x", "y", "z", "w"}


countCalls = 0;
countPairs = 1;
```

```
# main loop
def findDominionPair(p) :
    global VO, VC, E, V_even, V_odd, curPr, origPr, countPairs, countCalls
    countCalls = countCalls + 1
    # get initial U
    U = curPr[p] & VC
    alpha = p%2
    if alpha == 0 :
        alphaSet = V_even
        notAlphaSet = V_odd
    else :
        alphaSet = V_odd
        notAlphaSet = V_even
    # get attractor set of U
    U = getAtr(U, VC, alphaSet, notAlphaSet)

    # CHECK IF U CLOSED IN TOTAL GAME
    # create escape set of U (for not alpha)
    predU = getPred(U)
    predVOnotU = getPred(VO & ~ U)
    # find all alpha nodes in U that are not in Pre(U)
    a = alphaSet & U & ~ predU
    # find all non-alpha nodes in U that are in Pre(VO\U)
    na = notAlphaSet & U & predVOnotU
    # combine both
    escTotal = a | na

    if escTotal == Var.false :
            # closed in total game: get its attractor set
            dom = getAtr(U, VO, alphaSet, notAlphaSet)
            print("found dominion: ", dom.to_expr(), " for player: ", alpha)
            # check if there are vertices not in dominion
            res = VO & ~ dom
            if not res == Var.false :
                # continue with next dominion
                countPairs = countPairs + 1
                VO = res
                VC = res
                maxP = getMaxPr(VC)
                findDominionPair(maxP)
            else :
                print("algorithm complete")
            return;

    # CHECK IF CLOSED IN SUBGAME
    # create escape set of U (for not alpha)
    predU = getPred(U)
```

```
predVCnotU = getPred(VC & ~ U)
# find all alpha nodes in U that are not in Pre(U)
a1 = alphaSet & U & ~ predU
# find all alpha nodes in U that are in Pre(VC\U)
a2 = alphaSet & U & predVCnotU
# find all alpha nodes that MUST leave U
a = a2 & a1
# find all non-alpha nodes in U that are in Pre(VC\U)
na = notAlphaSet & U & predVCnotU

# combine both sets
escSub = a | na
if escSub == Var.false :
    # closed in subgame: get I and newP
    I = getSuc(notAlphaSet & U) & ~ U & VO
    newP = getMinPr(I)
    # get union of PCp for all p < newP
    union = Var.false
    for k in sorted(curPr) :
        if k < newP :
            union = union | curPr[k]
        else :
            break
    # for all vertices with p < newP reset priority
    for k in sorted(curPr) :
        if k < newP :
            curPr[k] = origPr[k] & union & ~ U
        else :
            break
    # promote all vertices in U to newP
    curPr[newP] = curPr[newP] | U
    # get new set of vertices
    VC = Var.false
    for k in sorted(curPr) :
        if k <= newP :
            VC = (VC | curPr[k]) & VO
        else :
            break
else :
    # open in subgame
    newP = 0;
    # promote all vertices in U to priority p
    curPr[p] = curPr[p] | U
    # remove these vertices from other priority groups...
    #... and find new max priority
    for k in curPr :
        if k != p :
```

```
                curPr[k] = curPr[k] & ~ U
                if k > newP and k < p and not curPr[k] & VO == Var.false:
                    newP = k
        # find new set of vertices
        VC = VC & ~ U
    return findDominionPair(newP);

def getPred(z) :
    global E, qvars, prime
    # rename the variables in z to their primed version
    next_z = Var.let(prime, z)
    # find all moves to a node in z
    w = E & next_z
    # existential quantification over x', y', z' ...
    # ...to find all nodes that can move into z
    return  Var.quantify(w, qvars, forall=False)

def getSuc(z) :
    global E, yvars, unprime
    # find all moves from a node in z
    w = E & z
    # existential quantification over x, y, z ...
    # ...to find all nodes that some node in U can move to
    sucz = Var.quantify(w, yvars, forall=False)
    # rename the variables in z to their original version
    return Var.let(unprime, sucz)

def getAtr(U, V, aS, naS) :
    # start from empty set
    q = U
    qold = Var.false
    # fixpoint reached ?
    while q != qold:
        qold = q
        predQ = getPred(q)
        predVnotU = getPred(V & ~ U)
        # take all non-alpha nodes that cannot make a move into VC\U
        na = naS & ~ predVnotU
        # take all non alpha nodes in na ...
        # ...and all alpha nodes that are in the current game
        # note since na not in Pre(VC\U) na must be in Pre(U)
        predq = (na | (predQ & aS)) & V
        q = q | predq | U
    return q

def getMinPr(V) :
    global curPr
```

```
    Keys = list(curPr.keys())
    Keys.sort(reverse=False)
    # if there is no vertex in V with the current lowest priority...
    #...consider next lowest priority
    for k in Keys :
        if curPr[k] & V != Var.false :
            newP = k
            break
    return newP


def getMaxPr(V) :
    global curPr
    Keys = list(curPr.keys())
    Keys.sort(reverse=True)
    # if there is no vertex in V with the current lowest priority...
    #...consider next lowest priority
    for k in Keys :
        if curPr[k] & V != Var.false :
            newP = k
            break
    return newP

# run the algorithm
startAlg = process_time()
findDominionPair(getMaxPr(VC))
end = process_time()
print("number of dominion pairs: ", countPairs)
print("number of recursive calls: ", countCalls)
print("total time elapsed: ", end - startCreating)
print("time needed for creating BDDS: ", startAlg - startCreating)
print("time needed for algorithm: ", end - startAlg)
```

C Test case mini

```
parity 9;
0 6 1 3, 4;
1 5 0 3, 8;
2 4 1 4, 8;
3 3 0 1, 6;
4 2 1 2, 4;
5 2 1 5, 7;
6 1 0 3, 6;
7 1 1 1, 5;
8 0 1 0, 8;
```

D Test case mini2

```
parity 7;
0 5 1 1;
1 6 0 3, 5;
2 4 0 1;
3 1 1 0, 4;
4 2 1 1, 6;
5 3 1 2, 4;
6 2 0 3, 5;
```