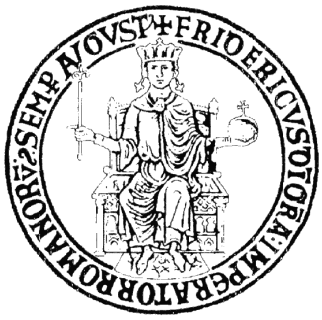
Università Degli Studi Di Napoli Federico II



Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso Di Laurea Magistrale In Informatica

Tesi Sperimentale di Laurea in Informatica

Solving Parity Games using Priority Promotion Techniques

Relatori:
Ch.mo Prof. *Massimo Benerecetti*
Dott. *Fabio Mogavero*

Candidato:
Daniele Dell'Erba
Matr. N97/154

Anno Accademico 2014-2015

Abstract

We consider *parity games*, a special form of two-player infinite-duration games on numerically labelled graphs, whose winning condition requires that the maximal value of a label occurring infinitely often during a play be of some specific parity. The problem has a rather intriguing status from a complexity theoretic viewpoint, since it belongs to the class UPTime ∩ CoUPTime. It is still open whether the question can be solved in polynomial time. Parity games also have great practical interest, as they arise in many fields of theoretical computer science, most notably logic, automata theory, and formal verification and synthesis. In this thesis, we propose a new family of algorithms, based on the idea of promoting vertices to higher priorities during the search for winning regions, which differ on the promotion strategy employed. The resulting family has nice computational properties, exhibiting the best space complexity among the currently known solutions to the problem. Experimental results on both random games and benchmark families show that the approach is also very effective in practice.

Introduction

Parity games [Mos91] are perfect-information two-player turn-based games of infinite duration, usually played on finite directed graphs, whose vertices, labelled by natural numbers called *priorities*, are assigned to one of the two players, named *Even* and *Odd* or, simply, 0 and 1, respectively. The game starts at a vertex and, during its evolution, each player can take a move only at its own positions, which consists in choosing one of the edges outgoing from the current vertex. The moves selected by the players induce an infinite sequence of vertices, called play. If the maximal priority of the vertices occurring infinitely often in the play is *even*, then the play is winning for player 0, otherwise, player 1 wins.

Parity games have been extensively studied in the attempt to find efficient solutions to the problem of determining the winner. From a complexity theoretic perspective, this decision problem lies in NPTime ∩ CoNPTime [EJS93] [EJS01], since it is *memoryless determined* [Mos91] [EJ91] [Mar75] [Mar85], and has been even proved to belong to UPTime ∩ CoUPTime [Jur98], a status shared with the *factorization problem* [FK92a] [FK92b] [AKS04]. They are the simplest class of games in a wider family with similar complexities and containing, *e.g.*, *mean payoff games* [EM79] [GKK90], *discounted payoff games* [ZP96], and *simple stochastic games* [Con92]. In fact, polynomial time reductions exist from parity games to the latter ones. However, despite being the most likely class among those games to admit a polynomial-time solution, the answer to the question whether such a solution exists still eludes the research community.

The effort devoted to provide efficient solutions stems primarily form the fact that many problems in formal verification and synthesis can be reformulated in terms of solving parity games. Emerson, Jutla, and Sistla [EJS93] [EJS01] have shown that computing winning strategies for these games is linear-time equivalent to solving the modal μCalculus model checking problem. Parity games also play a crucial role in automata theory [Mos84] [EJ91] [KV98], where, for instance, they can be applied to solve the complementation problem for alternating automata [GTW02], which, in turn, can be used to solve the satisfiability and model checking problems for expressive logics, such as the modal [Wil01] and alternating [SF06] [AHK02] μCalculus, ATL* [AHK02, Sch08b], Strategy

Logic [CHP10] [MMV10] [MMPV12] [MMPV14], Substructural Temporal Logic [BMM13] [BMM15], and fixed-point extensions of guarded first-order logics [BG01] [BG04], just to cite a few.

Previous solutions mainly divide into two families: those based on decomposing the game into winning regions and those trying to directly build winning strategies for the two players on the entire game. To the first family belong the *divide et impera* solution originally proposed by McNaughton [McN93] for Muller games and adapted to parity games by Zielonka [Zie98]. More recent improvements to that recursive algorithm have been proposed by Jurdziński, Paterson, and Zwick [JPZ06] [JPZ08] and by Schewe [Sch07], which rely on finding suitably closed subsets of winning regions, called dominions, which can then be removed from a game, thus reducing the size of the subgames to be recursively solved. To the second family belong the procedure proposed by Jurdziński [Jur00], which exploits the connection between the notions of progress measures [KK91] and winning strategies, as well as the strategy improvement algorithm proposed by Jurdziński and Vöge [VJ00], based on the idea of iteratively improving an initial non-winning strategy, and its optimized version by Schewe [Sch08a]. From a purely theoretical viewpoint, the best asymptotic behaviour obtained to date is the one exhibited by the solution proposed in [Sch07], which runs in time $\mathrm{O}\left(e \cdot n^{\frac{1}{3}k}\right)$, where n and e are the number of vertices and edges of the underlying graph and k is the number of priorities. As far as the space consumption is concerned, we have two incomparable best behaviours: $\mathrm{O}(k \cdot n \cdot \log n)$, for the small progress measure procedure of [Jur00] and $\mathrm{O}\left(n^2\right)$ for the optimized strategy improvement method of [Sch08a].

The main contribution of the thesis is a new family of algorithms for solving parity games, based on the notions of *region* and *priority promotion*. A region R for player $\alpha \in \{0, 1\}$, called an α-region, is a set of vertices from each of which player α can enforce a winning play that never leaves the region, unless the opponent can escape from R or the only choice for player α itself is to exit from R (*i.e.*, no edge from a vertex of α remains in R). Therefore, a region is essentially a very weak form of winning region. Regions can be ordered by assigning to each of them a priority corresponding to the best priority the opponent can obtain along a play exiting from that region. A crucial property of regions is that, under suitable and easy to check assumptions, a higher priority α-region R_1 and a lower priority α-region R_2 of the same parity, can be merged into a single α-region of the higher priority. We call this merging operation a priority promotion for R_2. The underlying idea of our approach is to iteratively enlarge α-regions by performing suitable sequences of promotions, until a *closed* α-region, namely a region in which player α can force the game to remain, is obtained. When that happens, a winning region for α has been found. Promotions at higher priorities can in general require to revoke promotions previously performed at lower ones and this phenomenon may be the source of exponential behaviours. Therefore, we propose three different algorithms based on the concepts sketched above. The first one implements

the basic idea. The second one builds on top of the first one and tries to avoid deconstructing previous promotions when possible. Finally, the third algorithm extends the second one by delaying promotions as much as possible, performing only those leading to the highest region among those available ones.

We provide soundness and completeness of the algorithms, an upper bound on their time complexity and a $O(n \cdot \log k)$ bound on the memory requirements. While for the first two algorithms an exponential lower bound can inddeed be found, for the third solver no exponential lower bound is currently known, thus leaving still open the possibility for a polynomial-time solution to the problem. Experimental results, comparing the new algorithms with the state of the art solvers, also show that all three algorithms perform very well in practice, most often significantly better than existing ones, on both random games and benchmark families proposed in the literature.

Contents

1 Parity Games **7**

1.1 Games . . . 7

1.2 Existing algorithms . . . 13

1.2.1 Zielonka . . . 14

1.2.2 Small progress measure . . . 15

1.2.3 Dominion decomposition . . . 16

1.2.4 Big Step . . . 16

1.2.5 Strategy improvement . . . 17

1.2.6 Known optimization techniques . . . 17

2 The Priority Promotion technique **19**

2.1 The solving approach . . . 19

2.2 Solution spaces . . . 24

2.2.1 Priority Promotion . . . 25

2.2.2 Parity Memoization . . . 37

2.2.3 Delayed Promotions . . . 44

3 Implementation **53**

3.1 Parity Game Solver . . . 53

3.1.1 Features . . . 53

3.1.2 Representation of basics operation on parity games . . . 56

3.2 Our new algorithm . . . 58

3.2.1 Implementation of Priority Promotion . . . 59

4 Experimental evaluation **63**

4.1 Concrete Problems 64

4.2 Special Families 67

4.3 Random Games 71

5 Conclusions **84**

A Source code **93**

Chapter 1

Parity Games

In this chapter we present parity games, which are the subject of the thesis. We shall give all the needed definitions to formalize the games along with examples showing how this particular games are played.

The main algorithms proposed in the literature will also be described, so as to allow to better understand the differences with respect to the solving approach proposed in this thesis.

Parity games are infinite games where two players move a token along the edges and where the parameter used to establish the winner of the game is the sequence of visited vertices.

1.1 Games

We introduce the family of games to which parity games belong. Most of the notions we introduce in the following like for instance the notions of path, strategy, etc, are also shared with the more general family of games over graphs.

Parity games is a class of coloured games and coloured games are, in turn, a form of games over graphs. So we start this section by describing the family of games over graph.

A graph consists of a set of vertices V and a set of edges E. Since the games we consider are turn based, in each vertex only one player can choose a move. The player are usually denoted as 0 and 1 or *Even* and *Odd*. Therefore the set V of vertices of the game is partitioned in two sets, V_0 and V_1, in such way that player α, with $\alpha \in \{0, 1\}$, is the owner of the vertices in V_α. Moreover, the games we are interested in have infinite duration, therefore we shall only consider *total* graphs, namely graphs whose vertices have at least one outgoing edge (no vertex is a sink). When the token is on a vertex in V_α, player α can take a move which consists in choosing the outgoing edge that must be crossed next.

The moves selected by the players induce an infinite sequence of vertices, called play, corresponding on infinite path in the graph.

Vertices are labeled by a labeling function λ, which is used to define the winning condition for the game.

Definition 1.1.1 (Game arena)**.** *An two player total graph, called* arena*, is a tuple* $A = \langle V_0, V_1, E, C, \lambda \rangle$*, where:*

- $V \triangleq V_0 \cup V_1$ *is the set of vertices;*
- $E \subseteq V \times V$ *is the set of edges, where for all* $u \in V$ *there exists* $v \in V$ *such that* $(u, v) \in E$*;*
- C *is a set of labels, called colours;*
- $\lambda : V \to C$ *is the labeling function.*

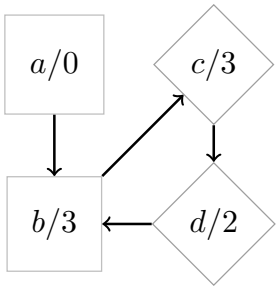


Figure 1.1: A coloured arena.

Figure 1.1 shows an example of arena, where squared shaped vertices belong to Player 1, diamond shaped ones belong to Player 0. The underlying arena is directed and labels are natural numbers.

For every vertex $v \in V$, we denote with $E(v)$ the set of vertices adjacent to v, *i.e.* $E(v) = \{v' \in V : (v, v') \in E\}$.

Playing a game consists in building an infinite sequence of vertices according to the edge relation, hence a paths of the underlying graph.

Definition 1.1.2 (Play)**.** *Given an arena* $A = \langle V_0, V_1, E, \lambda \rangle$*, a play of* A *is an infinite sequence* $\pi = v_0 v_1 \ldots$ *of vertices in* V *such that for all* $i \geq 0$*,* $(v_i, v_{i+1}) \in E$*.*

An example of game with a finite set of plays is reported in Figure 1.1. Its set of plays contains exactly four elements: $a(bcd)^\omega$, $(bcd)^\omega$, $(cdb)^\omega$ and $(dbc)^\omega$, where, for a sequence σ of symbols, the notations σ^ω stands for an infinite repetition of σ. We shall denote with Π_A the set of plays in the arena A.

A finite prefix t of a play π is called a *track* and $\mathsf{lst}(t)$ denotes the last vertex of the track. Examples of tracks of the game in Figure 1.1 are: *abc*, *abcdbcd* and *dbcd*. We shall denote with Trk_A the set of tracks of the arena A.

During the game, the token is moved along the edges to form an infinite play. At any instant the track corresponding to the prefix of the play, ending in the vertex where the token is positioned, encodes the history of the moves performed so far. A history ending in a vertex of V_α is called an α-position.

Definition 1.1.3 (α-positions)**.** *Given an arena $A = \langle V_0, V_1, E, \lambda\rangle$ and a player α with $\alpha \in \{0,1\}$, the set of* α-positions *of A is the set $Pos_\alpha = \{t \in Trk_A : \mathsf{lst}(t) \in V_\alpha\}$.*

When the token is on a vertex $v \in V_\alpha$, player α must choose the next move. Players choose moves according to some strategy, which is encoded as a function from α-positions to vertices.

Definition 1.1.4 (Strategy)**.** *A strategy σ for player α with $\alpha \in \{0,1\}$, is a total function $\sigma : Pos_\alpha \to V$ such that, for all α-positions $t \in Pos_\alpha$, $(\mathsf{lst}(t), \sigma(t)) \in E$.*

For some games, in particular for those we are interested in, a simpler notion of strategy suffices, where the choice of the next move only depends on the current vertex. Such strategies are called *memoryless*.

Definition 1.1.5 (Memoryless strategy)**.** *A memoryless strategy σ for player α is a total function $\sigma : V_\alpha \to V$ such that, for all for all vertices $v \in V_\alpha$, $(v, \sigma(v)) \in E$.*

Once a player adopts some strategy, not necessarily all the plays in Π_A are still feasible. The only feasible ones are those which conform with the choices stipulated by the strategy at the vertices of that player. Such players are called *consistent* with that strategy.

Definition 1.1.6 (Consistent play)**.** *Given a strategy (*resp.*, a memoryless strategy) σ_α for player α, a play $\pi = v_1 v_2 \ldots$ is consistent with σ_α if the following condition holds: for all $i \geq 0$, if $v_i \in V_\alpha$ then $v_{i+1} = \sigma_\alpha(v_0 \ldots v_i)$ (*resp.*, $v_{i+1} = \sigma_\alpha(v_i)$).*

While given a strategy for player α there may be several consistent plays with that strategy starting from the same vertex, when both players adopt a strategy, the outcome play from each vertex is always unique. Given the strategies σ_0 for player 0 and σ_1 for player 1, we denote with $\pi_{\sigma_0,\sigma_1}(v)$ the unique play starting from v and consistent with both σ_0 and σ_1.

As an example, the only play starting from vertex f and consistent with the pair of strategies $\sigma_0 = \{b \to d, d \to e\}$ for player 0 and $\sigma_1 = \{a \to b, c \to d, e \to a, f \to d\}$ for player 1 in the arena in Figure 1.2 is $\pi_{\sigma_0,\sigma_1}(f) = f(deab)^\omega$.

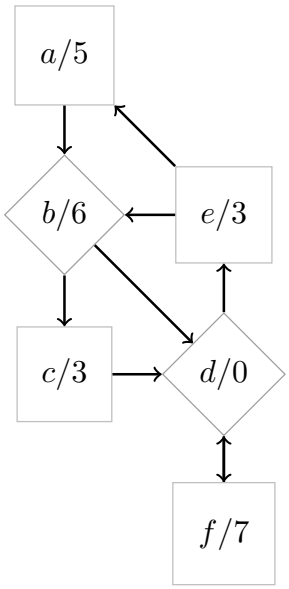


Figure 1.2: A coloured arena.

In order to formally define the notion of game, we first need to introduce the notion of winning play. Given a play $\pi = v_0 v_1 v_2 \ldots$, the corresponding sequence of colours is $\lambda(\pi) = \lambda(v_0)\lambda(v_1)\lambda(v_2)\ldots$. We denote with C^ω the set of infinite words over the set of colours C of an arena A. For each player α, a subset $\Lambda_\alpha \subseteq C^\omega$ is chosen as the set of winning words, called *winning condition*. A play π is winning for Player α if $\lambda(\pi) \in \Lambda_\alpha$. For adversarial games, such as those we consider in this thesis, the winning conditions Λ_0 and Λ_1 for the two players are disjoint. Moreover, we shall only consider games where $\Lambda_1 = C^\omega \setminus \Lambda_0$, *i.e.*, where all the plays not winning for player 0 are winning for player 1.

Definition 1.1.7 (Coloured game)**.** *A two player coloured game is a pair $\mathcal{G} = \langle A, \Lambda \rangle$, where $A = \langle V_0, V_1, E, C, \lambda \rangle$ is a game arena and Λ is the winning condition for player 0.*

A coloured game $\mathcal{G}' = \langle A', \Lambda \rangle$ is called a subgame of $\mathcal{G} = \langle A, \Lambda \rangle$ if A' is a game arena which is also a subgraph of A.

Among all the possible strategies for a player, we are interested in those whose consistent plays are winning, regardless of the strategy employed by the opponent player.

Definition 1.1.8 (Winning strategy)**.** *A strategy σ_α is a winning strategy for player α over the set $X \subseteq V$ if every play starting from a vertex in X is winning for α.*

Note that the notion of winning strategy defined above can be equivalently expressed by the following condition: σ_α is a winning strategy for player α over X if and only if for all $v \in X$ and for all strategy $\sigma_{\overline{\alpha}}$ of the opponent $\overline{\alpha}$, the unique play $\pi_{\sigma_\alpha,\sigma_{\overline{\alpha}}}(v)$ is winning for α. In the rest of the dissertation we shall use the symbol $\overline{\alpha}$ to denote the opponent player to α. Therefore, $\overline{\alpha} = 1$ when $\alpha = 0$, and *vice versa*.

When player α can force the game, *i.e.* has a strategy to stay within the set of vertices $U \subseteq V$ we call the set U an $\overline{\alpha}$-trap.

Definition 1.1.9 (α-trap)**.** *Given an arena $A = \langle V_0, V_1, E, \lambda\rangle$, a subset $U \subseteq V$ is an α-trap, with $\alpha \in \{0,1\}$, if for all strategy σ_α for player α and $v \in U$, there exist a strategy $\sigma_{\overline{\alpha}}$ for player $\overline{\alpha}$ such that the play $\pi_{\sigma_\alpha,\sigma_{\overline{\alpha}}}(v)$ is completely contained in U.*

In the game of Figure 1.2, the set $\{a, b, d, e\}$ is a 1-trap. In fact, player 0 can force all the plays starting from any vertex in the set into the trap. Only vertices b and d have outgoing edges from the 1-trap but both vertices belong to player 0 who choose to adopt a strategy to stay within the trap.

The solution to a coloured game consists of a unique pair of disjoint subsets W_0 and W_1 of V. The set W_α, with $\alpha \in \{0,1\}$, contains all the vertices starting from which player α can win.

Definition 1.1.10 (Winning set)**.** *The winning set for player α in a game $\mathfrak{D}$ is the set $W \subseteq V$ such that exist a strategy σ_α winning aver W.*

Note that a winning set for player α is also an $\overline{\alpha}$-trap. Games that admit, from each vertex $v \in W_\alpha$, a possibly different winning memoryless strategy, also admit a single memoryless strategy which is winning for all $v \in W_\alpha$. This result is summarized by the following proposition.

Proposition 1.1.1 ([GZ05])**.** *Given a coloured game $\mathfrak{D} = \langle A, \Lambda\rangle$, if, for all $v \in V_\alpha$, there exist a memoryless winning strategy σ_α, with $\alpha \in \{0,1\}$, then there exists a memoryless strategy σ^*_α for α, such that for every $v \in W_\alpha$, σ^*_α is winning over $\{v\}$.*

The game of Figure 1.2 is a game in which the winning strategy is not unique, indeed both strategies $(b \to c, d \to e)$ and $(b \to d, d \to e)$ are winning for player 0.

We now introduce two operators that are often used to solve games on graphs: the *predecessor* and the *attractor* operators. Given a set of vertices $X \in V$, the *α-predecessors* of X, in symbols $Pre^\alpha_{\mathfrak{D}}(X)$, are those vertices from which player α can force the game to reach some vertex in X with a single move. Clearly, player α can force its own vertices to reach X in one step as long as they have at least one edge entering X. However, the only way α can force a vertex of the opponent to enter X in a single step is if

the opponent has no other options, *i.e.* all the edged outgoing form the vertex lead to X. This motivates the following definition.

Definition 1.1.11 (α-predecessor)**.** *Given a game $\supset$ over the graph (V_0, V_1, E), a player α, with $\alpha \in \{0,1\}$, and a set of vertices $X \in V_0 \cup V_1$,*

$$Pre^{\alpha}_{\supset}(X) \triangleq \{v \in V_\alpha : \exists v' \in E(v).\ v' \in X\} \cup \{v \in V_{\overline{\alpha}} : \forall v' \in E(v).\ v' \in X\}$$

In the game of Figure 1.2, with $X = \{b\}$ and $\alpha = 0$, both vertices a and e, belong to player 1 and have edges leading to X. However, only a belong to $Pre^{\alpha}_{\supset}(X)$, since e has an edge leading outside X.

The α-attractor generalizes the notion of α-predecessor operator to an arbitrary number of moves.

Definition 1.1.12 (α-attractor)**.** *Given a game $\supset$ a player α with $\alpha \in \{0,1\}$, the attractor operator $\mathsf{atr}^{\alpha}_{\supset}(X) \triangleq \bigcup_{i \geq 0} X^i$, where the sequence of sets of vertices $X^i \subseteq V$ is inductively defined as follows:*

$X^0 = X$

$X^{i+1} = X^i \cup Pre^{\alpha}_{\supset}(X^i)$

Intuitively a vertex v belong to $\mathsf{atr}^{\alpha}_{\supset}(X)$ if player α can force to reach X in a finite numbers of moves. When the set of vertices V of the game is finite, then the $\mathsf{atr}^{\alpha}_{\supset}(X)$ is the union of the finite prefix of the sequence of sets X^i. In fact, in this case the sequence converges after a finite numbers of steps.

In the game of Figure 1.2, with $X = \{b\}$ and $\alpha = 0$, when computing $\mathsf{atr}^{\alpha}_{\supset}(X)$, according to Definition 1.1.12, X^1 only contains vertex a, since $Pre^{0}_{\supset}(X) = \{a\}$. Vertex e is attracted at the next step, because $Pre^{\alpha}_{\supset}(X^1) = \{a\}$ and $X^2 = \{a, b\}$. In the example the computation proceeds and attracts the whole game. Examples in which the set X does not attracts other vertices are $X = \{c\}$ and $X = \{f\}$.

Given a finite set of symbols Σ and a infinite $\rho = r_0 r_1 r_2 \ldots$ sequence with $r_i \in \Sigma$ for all $i \geq 0$, some symbol r_i must occur infinitely often. Given a sequence ρ, we denote with $\mathit{Inf}(\rho)$ the non-empty set of symbols which occur infinitely often in ρ. In the game of Figure 1.1, $\mathit{Inf}(\pi) = \{b, c, d\}$ for each play π in the game.

The notion of *Inf* allows us to formally define parity games.

Definition 1.1.13 (Parity game)**.** *A parity game is a pair $\supset = \langle A, \Lambda \rangle$, where $A = \langle V_0, V_1, E, \lambda \rangle$ is a game arena and $\Lambda = \{\rho \in C^\omega : \mathsf{max}(\mathit{Inf}(\rho)) \equiv_2 0\}$ is the winning condition for player 0 (the symbol $\equiv_2$ denote the algebraic congruence relation modulo 2).*

An important result about parity games is the determinacy theorem, stating that every vertex of the game is winning for one of the two players.

Proposition 1.1.2 ([EJ91]). *Parity game are determined so for every vertex $v \in V$, either Player 0 or Player 1 has a winning strategy therefore v must belong to a winning set W_α with $\alpha \in \{0, 1\}$.*

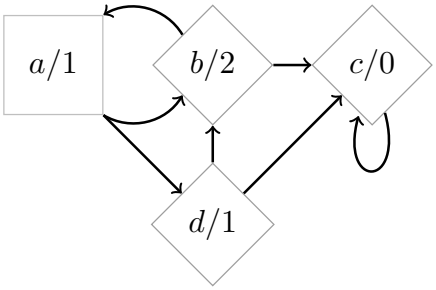


Figure 1.3: A parity game.

We can interpret the game of Figure 1.3 as a parity game. Under this interpretation, based on the previous definition, player 0 can play the strategy $\sigma_0 = \{b \to a, d \to b\}$ and force the game in the 1-trap $\{a, b, d\}$ which is its winning set. The winning set of player 1 is the set $\{c\}$, which is also a 0-trap.

The following proposition states that parity games admit memoryless strategies.

Proposition 1.1.3 ([Zie98]). *Given a parity game $\supset = \langle A, \Lambda \rangle$, for each α with $\alpha \in \{0, 1\}$, there exists a winning memoryless strategy σ_α for α, which is winning from each $v \in W_\alpha$.*

Therefore, α-positions essentially coincide with the vertices of the graph. Henceforth, we call α-position the vertices in V_α and, when the player is not relevant, simply positions.

1.2 Existing algorithms

We describe the most interesting existing algorithms in the literature for solving parity games. These algorithm are the result of years of research, improvement and study on parity game. They adopt different solving technique like decomposition of game and merge of partial solutions or building of strategies to solving the complete game. They also provide complexity bounds of space and time useful to compare to. Some of them inspired the new idea, the majority are simple touchstones. The first one is algorithm of Zielonka [Zie98], followed by Small progress measure [Jur00] and two Zielonka's evolutions, Dominion decomposition [JPZ06] and Big Step [Sch07]. Close this section Strategy improvement algorithm [VJ00].

1.2.1 Zielonka

Presented in [Zie98], this is the first algorithm for solving parity games. The algorithm, reported in Figure 1, is contains two recursive calls on different subgames. The presence of these two recursive calls can, in the worst case, lead to an exponential number of recursions.

The game is solved by decomposing it into subgames, until the empty game is obtained, *i.e.* the base case is reached. The game $\mathfrak{D}$ is analyzed by processing priorities starting from the highest one in the game. At each recursive call, a subgame $\mathfrak{D}'$ is generated by removing the α-attractor A of the set of positions in the current game $\mathfrak{D}$ with the highest priority, where α is the player congruent to the current highest priority. If $\mathfrak{D}'$ is completely won by α, then also the whole $\mathfrak{D}$ is won by α, since the winning strategy for α will be to reach A, where the maximum priority visited infinitely often by all plays is congruent to α. Any other play that does not reach A is also winning for α, since $\mathfrak{D}'$ is won by α.

Algorithm 1: The Recursive Algorithm of Zielonka.

signature $Solve : \mathrm{PG} \to_{\mathfrak{D}} (2^{\mathrm{Ps}_{\mathfrak{D}}} \times 2^{\mathrm{Ps}_{\mathfrak{D}}})$
function $Solve(\mathfrak{D})$
1 **if** $\mathrm{Ps}_{\mathfrak{D}} = \emptyset$ **then**
2 **return** $(\emptyset, \emptyset)$
else
3 $p \leftarrow \mathsf{max}(\mathrm{Pr}_{\mathfrak{D}})$
4 $\alpha \leftarrow n \bmod 2$
5 $\mathrm{N} \leftarrow \mathsf{pr}_{\mathfrak{D}}^{-1}(n)$
6 $\mathrm{A} \leftarrow \mathsf{atr}_{\mathfrak{D}}^{\alpha}(\mathrm{N})$
7 $(\mathrm{W}_{\alpha}', \mathrm{W}_{\overline{\alpha}}') \leftarrow Solve(\mathfrak{D} \setminus \mathrm{A})$
8 **if** $\mathrm{W}_{\overline{\alpha}} = \emptyset$ **then**
9 $(\mathrm{W}_{\alpha}, \mathrm{W}_{\overline{\alpha}}) \leftarrow (\mathrm{Ps}_{\mathfrak{D}}, \emptyset)$
10 **return** $(\mathrm{W}_{\alpha}, \mathrm{W}_{\overline{\alpha}})$
else
11 $\mathrm{B} \leftarrow \mathsf{atr}_{\mathfrak{D}}^{\overline{\alpha}}(\mathrm{W}_{\overline{\alpha}}')$
12 $(\mathrm{W}_{\alpha}'', \mathrm{W}_{\overline{\alpha}}'') \leftarrow Solve(\mathfrak{D} \setminus \mathrm{B})$
13 $(\mathrm{W}_{\alpha}, \mathrm{W}_{\overline{\alpha}}) \leftarrow (\mathrm{W}_{\alpha}'', \mathrm{W}_{\overline{\alpha}}'' \cup \mathrm{B})$
14 **return** $(\mathrm{W}_{\alpha}, \mathrm{W}_{\overline{\alpha}})$

If, however, $\Im'$ is not completely won by α, there exist $W'_{\overline{\alpha}}$, which is winning for $\overline{\alpha}$ in $\Im$, since α can not force the positions in $W'_{\overline{\alpha}}$ to reach A, the portion of $\Im$ missing in $\Im'$. Since player $\overline{\alpha}$ can force positions in A to reach $W'_{\overline{\alpha}}$, the $\overline{\alpha}$-attractor B on $W'_{\overline{\alpha}}$ is obtained to compute the subgame $\Im'' = \Im' \setminus B$, which is then solved by the second recursive call.

1.2.2 Small progress measure

This approach, presented in [Jur00], is based on the notion of progress measure function. Given a parity game with d colours, the progress measure function ϱ assigns to each vertex $v \in V$ a tuple $(c_1, c_2, \ldots, c_d)$ of d colours. A lexicographic order is defined over these tuples with the top value $\top$. Given two tuples $x = (x_1, x_2, \ldots, x_d)$, $y = (y_1, y_2, \ldots, y_d)$ and a colour p, the relation $x >_p y$ holds if the lexicographic order applied to the tuples truncated to the first p colours holds. For example $(1, 2, 3) >_2 (1, 1, 4)$ but $(1, 2, 3) \not>_3 (1, 1, 4)$. The aim of function ϱ is to assign to each vertex a tuple, according the following condition for each edge $(v, w) \in E$:

- if $\lambda(v) \equiv_2 0$ then $\varrho(v) \geq_{\lambda(v)} \varrho(w)$
- otherwise $\varrho(v) >_{\lambda(v)} \varrho(w)$

Given a strategy σ_α for player α, we denote with $\Im_{\sigma_\alpha}$ the game obtained from $\Im$ by removing all the edges $(v, v') \in E$ such that $v \in V_\alpha$ and $\sigma_\alpha(v) \neq v'$, essentially $\Im_{\sigma_\alpha}$ is the subgame of $\Im$ containing all and only the plays consistent with σ_α. We also define an *odd cycle* (*resp.*, even cycle) in a game as the cycles whose maximum vertex colour is odd (*resp.*, even). In [Jur00] the following two main results have been proved. First, a strategy σ_α for player α is winning if and only if there are no $\overline{\alpha}$-cycles in the subgame $\Im_{\sigma_\alpha}$. Second, a game $\Im$ admits a progress measure if and only if there exist a strategy σ_α such that the subgame $\Im_{\sigma_\alpha}$ has no $\overline{\alpha}$-cycles. As a consequence, a progress measure for player α exists if and only if player α has a winning strategy.

The solution proposed, then, reduces to computing such a progress measure for player 0. At the beginning of the computation, all the tuples are null, *i.e.*, all the values are equals to 0. For each vertex v, the algorithm raises the value of the tuple associated to it by means of a lifting function. The lifting of a vertex v is performed according to following the rule: if the parity of $\lambda(v)$ is odd, the maximal tuple among those associated with its successors is assigned to v, otherwise the minimal one its successors is chosen. Vertices connected by odd cycles fall into a lifting loop that rises the value of the tuples until the top value $\top$ is reached. When the algorithm terminates, *i.e.* no lifting is possible, vertices with defined values of ϱ are winning for player 0. While those with value $\top$ are winning for player 1. Unfortunately,

it is possible to create games where vertices exist that need to be lifted an exponential number of times. This algorithm has time complexity $O\left(d \cdot e \cdot \left(\frac{n}{d}\right)^{d/2}\right)$ and space complexity $O(d \cdot e \cdot \log n)$ [Jur00].

1.2.3 Dominion decomposition

The dominion decomposition algorithm [JPZ06] is a recent evolution of the original Zielonka's algorithm. The name dominion decomposition, comes from the notion of *dominion*. An α-dominion is a set of vertices for which there exist a strategy for α such that all the consistent plays are winning and contained in the dominion. Clearly, an α-dominion is also an $\overline{\alpha}$-trap. The winning set for α is itself an α-dominion, but it can contain several smaller α-dominions. The algorithm proceed as follow. At every iteration it tries to compute an α-dominions of $l = \sqrt{2 \cdot n}$ size. If such dominion is found it is removed from the game together with its attractor and the residual subgame is processed. If, on the other hand, no α-dominion is found, the algorithm behaves similarly to the Zielonka algorithm except for the fact that the recursive calls of the original Zielonka's algorithm are replaced with calls to the dominion decomposition algorithm. The search for an α-dominion is performed by an exhaustive generation of all the possible subsets of the chosen dominion size l. The total time for this algorithm is $n^{O(\sqrt{n})}$, while the space is $O(e \cdot n)$.

1.2.4 Big Step

The latest developed solvers for parity games is Big Step [Sch07], which exhibit the lowest time complexity currently known. Similarly to dominion decomposition it is based on the idea of findings dominions to reduce the computation times of the Zielonka's algorithm. However it differs from dominion decomposition in the way α-dominion are computed. In fact instead of performing an exhaustive search in the space of the possible dominions, it tries to directly compute an α-dominion by using a restricted version of the progress measure algorithm described above. The restriction of progress measure consist to limit the number of lifting, given a progress measure f, for each tuple $c = (c_1, c_2, \ldots, c_d)$ with d number of colours, $\sum_{i=0}^{d} f(c_i) \leq u$. The parameter u is set to value $\sqrt[3]{d \cdot n^2}$ that optimize the ratio between the dominions and the game.

Basically, it uses the progress measure approach as a partial solver to find dominions and the dominion decomposition approach on the solved sets to solve the game in larger steps.

With the above mentioned value of u, the resulting time complexity is $O\left(e \cdot n^{d/3}\right)$, while the space complexity is $O(n \cdot (e + d \cdot \log n))$.

1.2.5 Strategy improvement

This approach, presented in [VJ00], works with strategies. The idea is based on a pre-order $\sqsubseteq$ over the strategies of one of the players, say α. The maximum element σ'_α of this pre-order, *i.e.* the only element such that, for all σ_α, $\sigma_\alpha \sqsubseteq \sigma'_\alpha$, is the winning strategy for α on the whole winning set W_α.

Starting form an arbitrary strategy σ_α, the algorithm improves this strategy using an improvement function $improve : strategy_\alpha \to strategy_\alpha$, such that if σ_α is not winning, then $\sigma_\alpha \sqsubset improve(\sigma_\alpha) \not\sqsubseteq \sigma_\alpha$, *i.e.* the current strategy has been improved.

The strategy improvement algorithm given the current strategy σ_α, computes a counter strategy $\sigma_{\overline{\alpha}}$ for the opponent $\overline{\alpha}$. The algorithm assign to each vertex v the value $\mathsf{max}(\mathit{Inf}(\lambda(\pi_{\sigma_\alpha,\sigma_{\overline{\alpha}}}(v)))$. An improvement is possible if and only if player α can modify its strategy σ_α, by choosing a different move for some vertex, and the evaluation of $\pi_{\sigma'_\alpha,\sigma_{\overline{\alpha}}}(v)$ is better than $\pi_{\sigma_\alpha,\sigma_{\overline{\alpha}}}(v)$.

The improvement operation described above is, then, iteratively applied to the initial strategy until a fixpoint is reached.

The cost of a single strategy improvement step is $O(\mid V \mid \cdot \mid E \mid)$, while the maximum number of improvement needed is $O\left(\frac{n}{d}\right)^d$, with d the number of colours. In [Fri09] the possibility of a chain with an exponential number of improvements is provided. In fact, a family of parity games can be defined such that a binary counter can be encoded within the strategy improvement mechanism. The time complexity for this algorithm is $O(2^e \cdot n \cdot e)$, the space complexity is $O(n^2 + n \cdot \log d + e)$.

1.2.6 Known optimization techniques

Some restricted class of games show properties that facilitate their solution. We describe the well known optimizations [FL09] that can be adopted to solve these class of games.

Some techniques to solve parity games analyze positions, considering priority one at a time. Clearly, the higher the number of priorities the higher is the cost of this analysis. Some improvements can be obtained by checking the parity of the contiguous priorities in the game. We say that in a game two priorities p_1 and p_2 with $p_1 < p_2$ are consecutive if there are no positions with a priority $p_3 \in [p_1, p_2]$. If a game has two consecutive priorities p_1 and p_2 and $p_1 \equiv_2 p_2$, raising the priority of the positions labeled p_1 to p_2 will have no effect on the winning sets nor the strategies. This operation reduces the number of priorities in the game which significantly affect the time complexity of the parity game algorithms. Another useful transformation is priority *compaction*. Once a game has been transformed according to the previous transformation, if two consecutive priorities p_1 and p_2 satisfy $p_1 < p_2$ and $p_1 \not\equiv_2 p_2$, then $p_2 \geq p_1 + 1$. If $p_2 > p_1 + 1$ we can lower the priority of the positions in the game from p_2 to $p_1 + 1$

without effecting winning sets nor strategies. Priority compaction decrease the maximal priority in the game, another parameter that often affects the performances of the solvers.

Clearly a game in which $V = V_\alpha$ is a single player game, *i.e.* all vertices belongs to the same player. However, some non single player games may behave like single players ones. This is the case of games in which every vertex of player α has only one leaving edge. In fact player α has a single choice in each of its positions. Hence the only player actually playing the game is the opponent. Both type of games is known can be solved in time $O(n^3)$ [Ver13].

Strongly connected components (SCC) are maximal subsets of mutually reachable vertices. Given a graph $\eth$, vertices of the same SCC in $\eth$ are represented by a single vertex in the SCC graph $\eth_{SCC}$. SCC graphs are directed acyclic graphs. We can solve the subgame induced by the vertices of a sink of $\eth_{SCC}$ independently from the rest of $\eth$ vertices. Once the subgame is solved ,*i.e.* the pair of winning sets W_0, W_1 has been computed, to proceed the solving of $\eth$ we need to compute the α-attractors of W_0 and W_1 in $\eth$. If some vertex of a SCC is attracted, the SCC must be computed again to iterate the process.

Chapter 2

The Priority Promotion technique

In this chapter we present the new idea for solving parity games. The shared structure of the new algorithms is presented in the first section. The three algorithms we propose use this new idea and are explained in the second section. They fall into the category of the solution techniques where the game is decomposed into winning set, also called winning regions. In this sense, the approach is comparable to the Zielonka's approach and its variants.

2.1 The solving approach

A solution for a game $\mathcal{D} \in \mathrm{PG}$ can trivially be obtained by iteratively computing winning regions, namely sets of positions from where one player has a strategy to win the game. Once a winning region R, say for player α, is found, its α-attractor $\mathsf{atr}^{\alpha}_{\mathcal{D}}(\mathrm{R})$ gives an α-maximal winning region containing R such that the structure $\mathcal{D} \setminus \mathsf{atr}^{\alpha}_{\mathcal{D}}(\mathrm{R})$ resulting from subtracting the positions in $\mathsf{atr}^{\alpha}_{\mathcal{D}}(\mathrm{R})$ from $\mathcal{D}$ is still a game (in fact, $\mathsf{atr}^{\alpha}_{\mathcal{D}}(\mathrm{R})$ is an $\overline{\alpha}$-trap). Such a subgame can then be solved by iterating the process. Algorithm 2 makes this idea precise.

It takes as input a game $\mathcal{D}$ and recursively solves it: Lines 1-2 provide the base case for the empty game; Lines 3-4 compute a winning region R and its attractor $\mathrm{R}^{\star} \triangleq \mathsf{atr}^{\alpha}_{\mathcal{D}}(\mathrm{R})$; Line 5 solves the subgame $\mathcal{D} \setminus \mathrm{R}^{\star}$; finally, Lines 6-7 compute the overall solution by adding the winning region $\mathrm{R}^{\star}$ to the solution of the subgame and return it. Therefore, the real fundamental problem to solve consists in computing a winning region for some player in a game $\mathcal{D}$. The difficulty in finding such regions is that, in general, no unique priority exists with which the winner can satisfy the winning condition along all the plays inside the region, since that value depends on the strategy chosen by opponent.

Algorithm 2: The Parity Game Solver.

signature $\mathsf{sol}_\Gamma : \mathrm{PG} \to_\ni (2^{\mathrm{Ps}_\ni} \times 2^{\mathrm{Ps}_\ni})$

function $\mathsf{sol}_\Gamma(\ni)$

1 **if** $\mathrm{Ps}_\ni = \emptyset$ **then**

2 **return** $(\emptyset, \emptyset)$

else

3 $(\mathrm{R}, \alpha) \leftarrow \mathsf{win}_\Gamma(\ni)$

4 $\mathrm{R}^\star \leftarrow \mathsf{atr}^\alpha_\ni(\mathrm{R})$

5 $(\mathrm{W}_0', \mathrm{W}_1') \leftarrow \mathsf{sol}_\Gamma(\ni \setminus \mathrm{R}^\star)$

6 $(\mathrm{W}_\alpha, \mathrm{W}_{\overline{\alpha}}) \leftarrow (\mathrm{W}'_\alpha \cup \mathrm{R}^\star, \mathrm{W}'_{\overline{\alpha}})$

7 **return** $(\mathrm{W}_0, \mathrm{W}_1)$

Our solution to this problem is to proceed in a bottom-up fashion, starting from a weaker notion of winning region for player α, called simply α-region, and suitably compose α-regions to obtain a winning region for the same player. A set of vertices R is a region for player α if exists a strategy σ_α such that for every consistent play with σ_i, if the play remain in the region, it is winning for player α. To define boundaries of a region we need a preliminary definition that we call α-*escape*. Given a set of vertices R, its α-*escape* is the possibly empty subset of R from which player α can leave R in one move. This set includes $\overline{\alpha}$-positions with no edge leading to R itself.

Definition 2.1.1 (α-escape)**.** *Let $\ni$ be a parity game, $\alpha \in \{0,1\}$ and $\mathrm{R} \in V$. The* α-escape *set is defined as:*

$$\mathsf{esc}^\alpha_\ni(\mathrm{R}) \triangleq Pre^\alpha_\ni(V \setminus \mathrm{R}) \cap \mathrm{R}.$$

Intuitively $\mathsf{esc}^{\overline{\alpha}}_\ni(\mathrm{R})$ gives us the border positions at the borders of R. The notion of region is formalized be following definition.

Definition 2.1.2 (Regions)**.** *Let $\ni \in \mathrm{PG}$ be a game and $\alpha \in \{0,1\}$ a player. A non-empty set of positions $\mathrm{R} \subseteq \mathrm{Ps}_\ni$ is an* α-region *in $\ni$ if the set $\mathrm{WStr}^\alpha_\ni(\mathrm{R}) \subseteq \mathrm{Str}^\alpha_\ni(\mathrm{R})$ of* weakly winning α-strategies *$\sigma_\alpha \in \mathrm{WStr}^\alpha_\ni(\mathrm{R})$ defined as follows is non-empty: for all $\overline{\alpha}$-strategies $\sigma_{\overline{\alpha}} \in \mathrm{Str}^{\overline{\alpha}}_\ni(\mathrm{R})$ and positions $v \in \mathrm{R}$, the induced play $\pi = \mathsf{play}_\ni((\sigma_0, \sigma_1), v)$ satisfies $\mathsf{pr}_\ni(\pi) \equiv_2 \alpha$ and, if π is finite, then $\mathsf{lst}(\pi) \in \mathsf{esc}^{\overline{\alpha}}_\ni(\mathrm{R})$.*

In the game of Figure 2.1 the highest priority is 4, so the parity player is 0. The set of position with priority 4 is $\mathrm{R} = \{d, f\}$. Note that R is a 0-region, its escape is $\mathsf{esc}^1_\ni(\mathrm{R}) = \{f\}$, since position f have

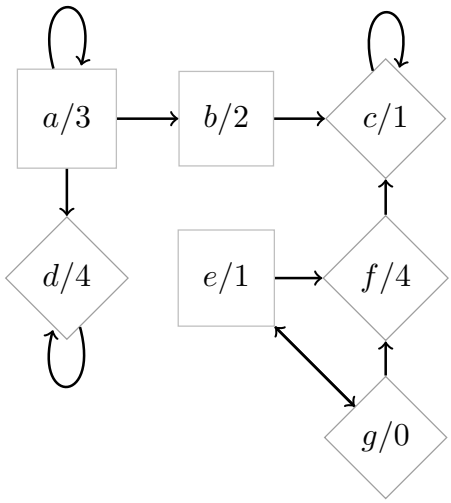


Figure 2.1: An explicative parity game for regions.

no edge leading to R, but d have only one edge leading to R. Its attractor is $\mathsf{atr}^{0}_{\mathcal{D}}(\mathrm{R}) = \{d, e, f, g\}$ that is again an 0-region. Note that $\mathsf{atr}^{0}_{\mathcal{D}}(\mathrm{R})$ contains a winning cycle for player 1, but one position of this cycle belongs to player 0 whose have a strategy that not close this cycle. Finally note that both sets we have generated are not 1-region, because there is a cycle winning for player 0. Given a region R such that $\mathsf{esc}^{\overline{\alpha}}_{\mathcal{D}}(\mathrm{R}) = \emptyset$, we know that player α has a strategy such that if a consistent play exits from R, it must end with a position in $\mathsf{esc}^{\overline{\alpha}}_{\mathcal{D}}(\mathrm{R})$. The fact that $\mathsf{esc}^{\overline{\alpha}}_{\mathcal{D}}(\mathrm{R}) = \emptyset$ implies that there are no consistent plays exiting from R and, since all remaining consistent plays are winning for player α by the definition of regions, the set R is a winning set for α. We call such regions α-closed and, of course, they are also $\overline{\alpha}$-traps.

If a region R has $\mathsf{esc}^{\overline{\alpha}}_{\mathcal{D}}(\mathrm{R}) \neq \emptyset$, it is not a winning region and we call it α-open. We can partition the set of α-regions $\mathrm{Rg}_{\mathcal{D}}$ into two disjoint sets: open α-regions $\mathrm{Rg}^{-}_{\mathcal{D}}$ and closed α-regions $\mathrm{Rg}^{+}_{\mathcal{D}}$. In the game of Figure 2.1 the set $A = \{d, e, f, g\}$ is an open region. We obtain a subgame removing A from the game. The highest priority in the subgame is 3 and its parity is 1. The region of positions with priority 3 is $\mathrm{R} = \{a\}$. Its attractor, $\mathsf{atr}^{1}_{\mathcal{D}}(\mathrm{R}) = \{a\}$ is a region 1-closed. During the search for a winning region, we explore a suitable partial order, whose elements, called *states*, record information about the open regions computed so far. The search starts from the top element, devoid of any information about regions. At each step, a query is performed on the current state to extract a new region, which is then used to compute a successor state, if such a region is open. If, on the other hand, it is closed, the search is over.

Different query and successor operations can in principle be defined, even on the same partial order. However, such operations cannot be completely independent. To account for this intrinsic dependence, we introduce a compatibility relation between states and regions that can be extracted by the query operation, which also forms the domain of the successor function. The partial order together with the query and successor operations and the compatibility relation form what we call a *region space*.

Definition 2.1.3 (Region Space)**.** *A* region space *for a game* $\partial \in \mathrm{PG}$ *is a tuple* $\mathcal{R} \triangleq \langle \partial, \mathcal{S}, \succ, \Re, \downarrow \rangle$*, where:*

- $\mathcal{S} \triangleq \langle \mathrm{S}, \top, \prec \rangle$ *is a well-founded partial order* w.r.t. $\prec \subset \mathrm{S} \times \mathrm{S}$ *with maximum* $\top \in \mathrm{S}$*;*
- $\succ \subseteq \mathcal{S} \times \mathrm{Rg}_{\partial}$ *is the* compatibility relation*;*
- $\Re : \mathrm{S} \to \mathrm{Rg}_{\partial}$ *is the* query function *mapping each element* $s \in \mathrm{S}$ *to a region pair* $(\mathrm{R}, \alpha) = \Re(s) \in \mathrm{Rg}_{\partial}$ *such that if* $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\partial}$ *then* $s \succ (\mathrm{R}, \alpha)$*;*
- $\downarrow : \succ \to \mathrm{S}$ *is the* successor function *mapping each pair* $(s, (\mathrm{R}, \alpha)) \in \succ$ *to the element* $s^\star \triangleq s{\downarrow}(\mathrm{R}, \alpha) \in \mathrm{S}$ *such that* $s^\star \prec s$*.*

Algorithm 3 encodes the process of extraction of regions and evolution of states, until a closed region is found.

Termination of Algorithm 2 immediately follows from the finiteness of the parity game in input and the well-foundedness of the associated region space. Therefore, the maximal number of recursive calls is bounded by the number of positions in the game. As a consequence, the algorithm runs in time $\mathrm{O}(e + n \cdot \mathsf{T}_{\mathsf{win}_\Gamma}(n, e, k))$, where $\mathsf{T}_{\mathsf{win}_\Gamma}(n, e, k)$ represents the time to compute a winning region for ∂. Note that the additional term $\mathrm{O}(e)$ corresponds to the cumulated cost to compute all the attractors. Given the top element of the region space $\mathcal{R} = \Gamma(\partial)$ associated with game ∂, Algorithm3 searches for a winning region of a player by querying the current state s for a region pair (R, α). If this is closed in ∂, it is returned as a winning region for α. Otherwise, a successor state $s{\downarrow_{\mathcal{R}}}(\mathrm{R}, \alpha)$ is computed and the search proceeds recursively from it. Clearly, since the partial order is well-founded, termination of the $\mathsf{win}_{\mathcal{R}}$ procedure is guaranteed.

Algorithm 3: The Winning Region Searcher.

```
signature win_Γ : PG →_∋ Rg⁺_∋
function win_Γ(∋)
1   return win_{Γ(∋)}(⊤_{Γ(∋)})
signature win_R : S_R → Rg⁺_{∋_R}
function win_R(s)
1   (R, α) ← ℜ_R(s)
2   if (R, α) ∈ Rg⁺_{∋_R} then
3       return (R, α)
    else
4       return win_R(s↓_R(R, α))
```

The total number of recursive calls is, therefore, the execution depth $\mathsf{d}_{\mathcal{R}}(n, e, k)$ of the region space $\mathcal{R}$. Hence, $\mathsf{win}_{\mathcal{R}}$ runs in time $\mathrm{O}(\mathsf{d}_{\mathcal{R}}(n, e, k) \cdot (\mathsf{T}_{\Re}(n, e) + \mathsf{T}_{\downarrow}(n, e))$, where $\mathsf{T}_{\Re}(n, e)$ and $\mathsf{T}_{\downarrow}(n, e)$ denote the time needed by the query and successor functions, respectively. Thus, the total time to solve a game is $\mathrm{O}(e + n \cdot \mathsf{d}_{\mathcal{R}}(n, e, k) \cdot (\mathsf{T}_{\Re}(n, e) + \mathsf{T}_{\downarrow}(n, e)))$. Since the query and successor functions of all the region spaces considered in the rest of the thesis can be computed in linear time *w.r.t.* both n and e, the algorithm sol_{Γ} terminates in time $\mathrm{O}(n \cdot (n + e) \cdot \mathsf{d}_{\mathcal{R}}(n, e, k))$.

As to the space requirements, observe that sol_{Γ} can easily be turned into a tail recursive algorithm, while $\mathsf{win}_{\mathcal{R}}$ already is. Therefore, we can write both algorithms as one iterative algorithm 4. Hence, the upper bound on memory only depends on the space needed to encode the states of a region space, namely $O(\log \|\mathcal{R}\|)$, where $\|\mathcal{R}\|$ is the size of the region space $\mathcal{R}$.

Soundness of the approach follows from the observation that α-regions closed in the whole game are winning for player α and so are their α-attractors. Completeness, instead, is ensured by the nature of region spaces. In fact, algorithm $\mathsf{win}_{\mathcal{R}}$ always terminates by well-foundedness of the underlying partial order and, when it eventually does, a winning region for some player is returned.

Algorithm 4: The Monolithic Solver.

signature $\mathsf{sol}_{\Gamma} : \mathrm{PG} \to_{\mathfrak{D}} (2^{\mathrm{Ps}_{\mathfrak{D}}} \times 2^{\mathrm{Ps}_{\mathfrak{D}}})$
function $\mathsf{sol}_{\Gamma}(\mathfrak{D})$
1 $(\mathrm{W}_0, \mathrm{W}_1) \leftarrow (\emptyset, \emptyset)$
2 **while** $\mathrm{Ps}_{\mathfrak{D}} \neq \emptyset$ **do**
3 $s \leftarrow \top_{\Gamma(\mathfrak{D})}$
4 $(\mathrm{R}, \alpha) \leftarrow \Re_{\Gamma(\mathfrak{D})}(s)$
5 **while** $(\mathrm{R}, \alpha) \notin \mathrm{Rg}^{+}_{\mathfrak{D}}$ **do**
6 $s \leftarrow s\downarrow_{\Gamma(\mathfrak{D})}(\mathrm{R}, \alpha)$
7 $(\mathrm{R}, \alpha) \leftarrow \Re_{\Gamma(\mathfrak{D})}(s)$
8 $\mathrm{R}^{\star} \leftarrow \mathsf{atr}^{\alpha}_{\mathfrak{D}}(\mathrm{R})$
9 $\mathfrak{D} \leftarrow \mathfrak{D} \setminus \mathrm{R}^{\star}$
10 $\mathrm{W}_{\alpha} \leftarrow \mathrm{W}_{\alpha} \cup \mathrm{R}^{\star}$
11 **return** $(\mathrm{W}_0, \mathrm{W}_1)$

2.2 Solution spaces

As anticipated above, to have a chance to satisfy the definitional constraints of a region space, its states need to encode information about the open regions already computed. To do this, we distinguish open regions by associating with them values based on the priorities of some of the positions they contain. More precisely, according to Definition 2.1.2, open α-regions admit plays exiting from them. To each of those plays, the priority corresponding to the maximal value occurring in it is assigned. Hence, the value associated with each region is chosen as the best priority the opponent can obtain among all the escaping paths. Such a value, for an α-region R in a game $\mathfrak{D} \in \mathrm{PG}$, is called the *best escaping priority* of $\overline{\alpha}$ from R in $\mathfrak{D}$ and denoted by $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}) \in \mathrm{Pr}$. To formally introduce this concept, first let $\mathrm{TStr}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}, \sigma_{\alpha}) \subseteq \mathrm{Str}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}) \times \mathrm{R}$, called the set of *weakly tying $\overline{\alpha}$-strategies and positions*, be defined as the collection of pairs $(\sigma_{\overline{\alpha}}, v) \in \mathrm{TStr}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}, \sigma_{\alpha})$ for which the induced play $\pi = \mathsf{play}_{\mathfrak{D}}((\sigma_0, \sigma_1)v)$ is finite. Then, $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R})$ is set as the maximum over the weakly winning α-strategies $\sigma_{\alpha} \in \mathrm{WStr}^{\alpha}_{\mathfrak{D}}(\mathrm{R})$ of the minimum over the weakly tying $\overline{\alpha}$-strategies and positions $(\sigma_{\overline{\alpha}}, v) \in \mathrm{TStr}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}, \sigma_{\alpha})$ of the priority associated with the play $\pi = \mathsf{play}_{\mathfrak{D}}((\sigma_0, \sigma_1), v)$. Formally, $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}) \triangleq \mathsf{max}_{\sigma_{\alpha} \in \mathrm{WStr}^{\alpha}_{\mathfrak{D}}(\mathrm{R})} \mathsf{min}_{(\sigma_{\overline{\alpha}}, v) \in \mathrm{TStr}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}, \sigma_{\alpha})} \mathsf{pr}_{\mathfrak{D}}(\pi)$. It is easy to observe that the priorities of all positions in the escaping set of a region are at least as big as its best escaping priority, *i.e.*, $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}) \leq \mathsf{pr}(v)$, for all $v \in \mathsf{esc}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R})$. Moreover, at least one of such positions

$v \in \mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R})$ assumes that value, *i.e.*, $\mathsf{pr}(v) = \mathsf{bep}^{\overline{\alpha}}_{\partial}(\mathrm{R})$. Due to the particular instances of the region spaces introduced in the following, we only consider regions that are *uniform* in the priorities associated to the escaping paths, *i.e.*, for all positions $v_1, v_2 \in \mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R})$, it holds that $\mathsf{pr}(v_1) = \mathsf{pr}(v_2)$.

2.2.1 Priority Promotion

At this point, we have all the tools to explain the crucial steps underlying the search procedure. Open regions are not winning, as the opponent can force a tie by exiting from them. Therefore, in order to build a winning region starting from open ones, we look for a suitable sequence of regions that can be merged together until a closed one is found. Obviously, the merging operation needs to be applied only to regions belonging to the same player. Moreover, in case the result of a merge is still open, a bep value needs to be assigned to it. To this end, a mechanism is proposed, where two α-regions, R and $\mathrm{R}^\natural$, can be merged if the only moves exiting from $\overline{\alpha}$-positions of the lower priority one, say R, lead to higher priority α-regions and $\mathrm{R}^\natural$ has the lowest priority among them. As we shall see, this ensures that the new region $\mathrm{R}^\star \triangleq \mathrm{R}^\natural \cup \mathrm{R}$ has the same bep as $\mathrm{R}^\natural$. This merging operation, whose correctness is based on the following proposition, is called *promotion* of the lower region to the higher one.

Proposition 2.2.1 (Region Merging)**.** *Let* $\partial \in \mathrm{PG}$ *be a game,* $(\mathrm{R}^\natural, \alpha) \in \mathrm{Rg}_\partial$ *a region pair, and* $\mathrm{R} \subseteq \mathrm{Ps}_{\partial \setminus \mathrm{R}^\natural}$ *an* α*-region in the subgame* $\partial \setminus \mathrm{R}^\natural$ *such that:*

1. *if* $p \triangleq \mathsf{bep}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\natural) < \infty$*, then* $\mathsf{pr}_\partial(v) \leq p$*, for all positions* $v \in \mathrm{R}$ *with* $\mathsf{pr}_\partial(v) \not\equiv_2 p$ *and;*
2. $\mathsf{bep}^{\overline{\alpha}}_{\partial \setminus \mathrm{R}^\natural}(\mathrm{R}) \in \{p, \infty\}$.

Then, $\mathrm{R}^\star \triangleq \mathrm{R}^\natural \cup \mathrm{R}$ *is an* α*-region in* ∂ *with* $\mathsf{bep}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\star) \in \{p, \infty\}$*. Moreover, if both* $\mathrm{R}^\natural$ *and* R *are* α*-maximal in* ∂ *and* $\partial \setminus \mathrm{R}^\natural$*, respectively, then* $\mathrm{R}^\star$ *is* α*-maximal in* ∂ *as well.*

The proof of such a proposition is quite trivial, since it suffices to show that the union $\sigma_{\mathrm{R}^\star} \triangleq \sigma_{\mathrm{R}} \cup \sigma_{\mathrm{R}^\natural}$ of two weakly winning strategies σ_{R} and $\sigma_{\mathrm{R}^\natural}$, for the disjoint regions R and $\mathrm{R}^\natural$, is a weakly winning strategy for $\mathrm{R}^\star$. Indeed, all finite plays π induced by $\sigma_{\mathrm{R}^\star}$ and exiting from $\mathrm{R}^\star$ necessarily pass trough an escaping position in $\mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\star) \subseteq \mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\natural) \cup \mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R})$. Thus, the required properties $\mathsf{pr}_\partial(\pi) \equiv_2 \alpha$ and $\mathsf{lst}(\pi) \in \mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\star)$ are verified, due to the assumed hypotheses. The same holds for the plays that eventually remain inside one of the two regions, since they are won by α. If, instead, the play passes infinitely often through both R and $\mathrm{R}^\natural$, it necessarily visits infinitely often an escaping position in $\mathsf{esc}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\natural)$. Therefore, by Item (1), we have that $\mathsf{pr}_\partial(\pi) = \mathsf{bep}^{\overline{\alpha}}_{\partial}(\mathrm{R}^\natural)$. Hence, α wins this play as well.

Information about the computed regions during the search is collected in the form of a suitable priority function $\mathsf{r} \in \Delta_\partial \triangleq \mathrm{Ps}_\partial \rightharpoonup \mathrm{Pr}_\partial$, called *region function*, assigning to the positions occurring in

each obtained region a priority, namely the bep of that region. Based on the intuitions above, the search procedure evolves as follows. Priorities are considered starting from the highest one in the original game $\mathcal{D}$. A region of the same parity $\alpha \in \{0,1\}$ of the priority p under consideration is formed by first collecting the set of positions $\mathrm{N} \triangleq \mathsf{pr}_{\mathcal{D}^\star}^{-1}(p)$ with that priority in the subgame $\mathcal{D}^\star$, which is derived from $\mathcal{D}$ by removing the previously computed regions. Then, by computing its attractor $\mathrm{R} \triangleq \mathsf{atr}_{\mathcal{D}^\star}^{\alpha}(\mathrm{N})$, an α-maximal set of positions is obtained, from where the corresponding player can ensure the visit to a priority p. If the α-region R is open in $\mathcal{D}^\star$, we proceed and process the next priority. In this case, we are sure that $\mathsf{bep}_{\mathcal{D}^\star}^{\overline{\alpha}}(\mathrm{R}) = p$. Otherwise, one of two situations may arise: either R is closed in the whole game $\mathcal{D}$ or the only $\overline{\alpha}$-moves exiting from R lead to higher regions of the same parity. In the former case, R is a winning region and the search stops. In the latter case, a promotion of R to a higher region $\mathrm{R}^\natural$ is performed and the search restarts from the priority of such region, after erasing all the lower computed ones. The region $\mathrm{R}^\star$ resulting from the union of $\mathrm{R}^\natural$ and R will then be reprocessed and, possibly, extended in order to make it α-maximal. If R can be promoted to more than one region, the one with the lowest priority is chosen, so as to ensure the correctness of the merging operation. Due to the property of maximality, no $\overline{\alpha}$-moves from R to previously computed $\overline{\alpha}$-regions exist. Therefore, only regions of the same parity need to be considered in the promotion step. The correctness of region construction and extension, the remaining two fundamental steps in the proposed approach, is formalized by the following two propositions.

Proposition 2.2.2 (Region Construction)**.** *Let* $\mathcal{D} \in \mathrm{PG}$ *be a game and* $\mathrm{N} \triangleq \mathsf{pr}_{\mathcal{D}}^{-1}(p)$ *the set of positions having maximal priority* $p \triangleq \mathsf{pr}(\mathcal{D})$ *of parity* $\alpha \triangleq p \bmod 2$*. Then,* $\mathrm{R} \triangleq \mathsf{atr}_{\mathcal{D}}^{\alpha}(\mathrm{N})$ *is a* α*-maximal* α*-region in* $\mathcal{D}$ *with* $\mathsf{bep}_{\mathcal{D}}^{\overline{\alpha}}(\mathrm{R}) \in \{p, \infty\}$.

Proposition 2.2.3 (Region Extension)**.** *Let* $\mathcal{D} \in \mathrm{PG}$ *be a game and* $(\mathrm{R}^\star, \alpha) \in \mathrm{Rg}_{\mathcal{D}}$ *a region pair such that, if* $p \triangleq \mathsf{bep}_{\mathcal{D}}^{\overline{\alpha}}(\mathrm{R}^\star) < \infty$*, then* $\mathsf{pr}_{\mathcal{D}}(v) < p$*, for all positions* $v \in \mathrm{Ps}_{\mathcal{D}} \setminus \mathrm{R}^\star$ *with* $\mathsf{pr}_{\mathcal{D}}(v) \not\equiv_2 p$*. Then,* $\mathrm{R} \triangleq \mathsf{atr}_{\mathcal{D}}^{\alpha}(\mathrm{R}^\star)$ *is an* α*-maximal* α*-region in* $\mathcal{D}$ *with* $\mathsf{bep}_{\mathcal{D}}^{\overline{\alpha}}(\mathrm{R}) \in \{p, \infty\}$.

Figure 2.2 and Table 2.1 illustrate the search procedure on an example game, where diamond shaped positions belong to player 0 and square shaped ones to the opponent 1. The two 0-regions containing all the positions in the game and those in the set $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{g}, \mathtt{i}\}$ are both winning regions for 0. Each cell of the table contains a computed region with priority corresponding to the index of its row. A downward arrow denotes a region that is open in the subgame where it is computed, while an upward arrow means that the region gets to be promoted to the priority in the subscript. Following the idea sketched above, the first region obtained is the single-position 0-region $\{\mathtt{a}\}$, which is open because of the two moves leading to $\mathtt{d}$ and $\mathtt{e}$. At priority 5, the open 1-region $\{\mathtt{b}, \mathtt{f}, \mathtt{h}\}$ is formed by attracting both $\mathtt{f}$ and $\mathtt{h}$ to $\mathtt{b}$,

which is open in the subgame where {a} is removed. Similarly, the 0-region {c} at priority 4 and the 1-region {d} at priority 3 are open once removed {a, b, f, h} and {a, b, c, f, h}, respectively. At priority 2, the 0-region {e} is closed in the corresponding subgame. However, it is not closed in the whole game, since it has an move leading to c, *i.e.*, to region 4. A promotion of {e} to 4 is then performed, resulting in the new 0-region {c, e}. The search resumes at the corresponding priority and, after computing the extension of such a region via the attractor, we obtain that it is still open in the corresponding subgame. Consequently, the 1-region of priority 3 is recomputed and, then, priority 1 is processed to build the 1-region {g}. The latter is closed in the associated subgame, but not in the original game, because of a move leading to position d. Hence, another promotion is performed, leading to closed region in Row 3 and Column 3, which in turn triggers a promotion to 5. Observe that every time a promotion to a higher region is performed, all regions at lower priorities are destroyed. The iteration of the region forming and promotion steps proceeds until the configuration in Column 7 is reached. Here only two 0-regions are present: the open region 6 containing {a, b, d, g, i} and the closed region 4 containing {c, e, f, h}. The second one has an move leading to the first one, hence, it is promoted to its priority. This last operation forms a 0-region containing all the positions of the game. It is obviously closed in the whole game and is, therefore, a winning region for 0.

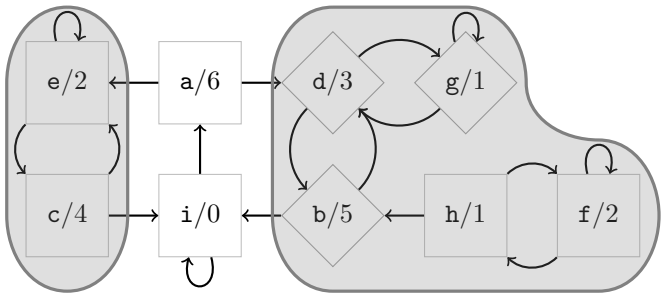


Figure 2.2: Running example.

	1	2	3	4	5	6	7
6	a↓	···	···	···	···	a, b, d, g, i↓	···
5	b, f, h↓	···	···	b, d, f, g, h↓	···		
4	c↓	c, e↓	···	c↓	c, e↓	c↓	c, e, f, h↑$_6$
3	d↓	d↓	d, g↑$_5$				
2	e↑$_4$			e↑$_4$		e, f, h↑$_4$	
1		g↑$_3$					
0					i↑$_6$		

Table 2.1: PP simulation.

Note that, 0-region $\{\mathtt{c},\mathtt{e}\}$ is removed, when 1-region $\{\mathtt{d},\mathtt{g}\}$ in Column 3 is promoted to 5. Similarly, when 0-region $\{\mathtt{i}\}$ in Column 5 is promoted to 6, both regions $\{\mathtt{b},\mathtt{d},\mathtt{f},\mathtt{g},\mathtt{h}\}$ and $\{\mathtt{c},\mathtt{e}\}$, highlighted by the gray areas, gets deleted. This is actually necessary for correctness, at least in general. In fact, if region $\{\mathtt{b},\mathtt{d},\mathtt{f},\mathtt{g},\mathtt{h}\}$ were not deleted, the promotion of $\{\mathtt{i}\}$ to 6, which also attracts $\mathtt{b}$, $\mathtt{d}$, and $\mathtt{g}$, would leave $\{\mathtt{f},\mathtt{h}\}$ as a 1-region of priority 5. However, according to Definition 2.1.2, this is not a 1-region. Even worse, it would also be considered a closed 1-region in the entire game, without being a winning region for player 1, since it is actually an open 0-region. This shows that, in principle, promotions to an higher priority require the deletion of previously built regions of lower priorities.

In the rest of this section, we shall formalize the intuitive idea described above. The necessary conditions under which promotion operations can be applied are also stated. Finally, suitable query and successor algorithms are provided, which ensure that such conditions are easy to check and always met when promotions are performed.

In order to define the region space induced by the *priority-promotion mechanism* (PP, for short), we need to introduce some additional notation. We say that a region pair $(\mathrm{R},\alpha) \in \mathrm{Rg}_{\mathfrak{D}}$ is saturated if $\mathsf{pr}_{\mathfrak{D}}(v) < \mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R})$, for all positions $v \in \mathrm{Ps}_{\mathfrak{D}} \setminus \mathrm{R}$ such that $\mathsf{pr}_{\mathfrak{D}}(v) \not\equiv_2 \alpha$. Intuitively, R is α-saturated if the only priorities greater than the bep of R in the residual game $\mathfrak{D} \setminus \mathrm{R}$ have parity α. Given a priority function $\mathsf{r} \in \Delta_{\mathfrak{D}}$ and a priority $p \in \mathrm{Pr}$, we denote by $\mathsf{r}^{(\geq p)}$ (*resp.*, $\mathsf{r}^{(>p)}$) the function obtained by restricting the domain of r to the positions with priority greater than or equal to p (*resp.*, greater than p). Formally, $\mathsf{r}^{(\geq p)} \triangleq \mathsf{r}{\restriction}\{v \in \mathsf{dom}(\mathsf{r}) : \mathsf{r}(v) \geq p\}$ and $\mathsf{r}^{(>p)} \triangleq \mathsf{r}{\restriction}\{v \in \mathsf{dom}(\mathsf{r}) : \mathsf{r}(v) > p\}$. By $\mathfrak{D}^{\leq p}_{\mathsf{r}}$ (*resp.*, $\mathfrak{D}^{<p}_{\mathsf{r}}$) we denote the largest subgame contained in the structure $\mathfrak{D} \setminus \mathsf{dom}\big(\mathsf{r}^{(>q)}\big)$ (*resp.*, $\mathfrak{D} \setminus \mathsf{dom}\big(\mathsf{r}^{(\geq q)}\big)$), which is obtained by removing from $\mathfrak{D}$ all the positions in the domain of $\mathsf{r}^{(>p)}$ (*resp.*, $\mathsf{r}^{(\geq p)}$). Finally, given a Boolean flag ζ, we set $p(\zeta) \triangleq p$, if $\zeta = \mathtt{f}$, and $p(\zeta) \triangleq p+1$, otherwise.

A priority function $\mathsf{r} \in \mathbb{R}_{\mathfrak{D}} \subseteq \Delta_{\mathfrak{D}}$ in $\mathfrak{D}$ is a *region function* iff, for all priorities $q \in \mathsf{rng}(\mathsf{r})$ with $\alpha = q \bmod 2$, it holds that $\mathrm{R} \triangleq \mathsf{r}^{-1}(q) \cap \mathrm{Ps}_{\mathfrak{D}^{\leq q}_{\mathsf{r}}}$ is an α-region in $\mathfrak{D}^{\leq q}_{\mathsf{r}}$ such that $q = \mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}^{\leq q}_{\mathsf{r}}}(\mathrm{R})$, if it is also α-open in this game. In addition, r is *maximal* (*resp.*, *saturated*) if all regions R associated with the priorities q are α-maximal (*resp.*, α-saturated).

To account for the current status of the search of a winning region, states s of the corresponding region space need to contain the current region function r, a Boolean flag ζ discriminating whether it is the result of a promotion, and the current priority p reached by the search in $\mathfrak{D}$. To each of such states $s \triangleq (\mathsf{r},\zeta,p)$, we associate then the *subgame at* s defined as $\mathfrak{D}_s \triangleq \mathfrak{D}^{<p(\zeta)}_{\mathsf{r}}$, which represents the portion of the original game that still has to be processed and partitioned in regions.

We can now formally define the *Priority Promotion* region space, by characterizing the corresponding state space and compatibility relation. Moreover, suitable algorithms to compute the query and successor functions defined on that space are provided.

Definition 2.2.1 (State Space for PP)**.** *A* PP state space *is a tuple* $\mathcal{S}_{\mathfrak{D}}^{\mathrm{PP}} \triangleq \langle \mathrm{S}_{\mathfrak{D}}^{\mathrm{PP}}, \top_{\mathfrak{D}}^{\mathrm{PP}}, \prec_{\mathfrak{D}}^{\mathrm{PP}} \rangle$*, where its components are defined as prescribed in the following:*

1. $\mathrm{S}_{\mathfrak{D}}^{\mathrm{PP}} \subseteq \mathbb{R}_{\mathfrak{D}} \times \{\mathtt{t}, \mathtt{f}\} \times \mathrm{Pr}_{\mathfrak{D}}$ *is the set of all triples* $s \triangleq (\mathsf{r}, \zeta, p)$*, called* states*, composed by a region function* $\mathsf{r} \in \mathbb{R}_{\mathfrak{D}}$*, a Boolean flag* $\zeta \in \{\mathtt{t}, \mathtt{f}\}$*, and a priority* $p \in \mathrm{Pr}_{\mathfrak{D}}$ *such that* (a) $\mathsf{r}^{(\geq p(\zeta))}$ *is maximal,* (b) *if* $\zeta = \mathtt{f}$ *then* $p = \mathsf{pr}(\mathfrak{D}_s)$ *else* $p \in \mathsf{rng}(\mathsf{r})$*,* (c) r *is saturated, and* (d) $p \leq \mathsf{min}(\mathsf{rng}(\mathsf{r}))$*;*

2. $\top_{\mathfrak{D}}^{\mathrm{PP}} \triangleq (\varnothing, \mathtt{f}, \mathsf{pr}(\mathfrak{D}))$*;*

3. *for any two states* $s_1 \triangleq (\mathsf{r}_1, \zeta_1, \_), s_2 \triangleq (\mathsf{r}_2, \zeta_2, \_) \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{PP}}$*, it holds that* $s_1 \prec_{\mathfrak{D}}^{\mathrm{PP}} s_2$ *iff either* (a) *there exists a priority* $q \in \mathsf{rng}(\mathsf{r}_1)$ *such that* (a.i) $\mathsf{r}_1{}^{(>q)} = \mathsf{r}_2{}^{(>q)}$ *and* (a.ii) $\mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$*, or* (b) *both* (b.i) $\mathsf{r}_1 = \mathsf{r}_2$ *and* (b.ii) $\zeta_1 \neq \zeta_2 = \mathtt{t}$ *hold.*

The state space precisely specifies the configurations in which the priority promotion procedure can reside and the relative order that the successor function must to satisfy. In particular, for a given state $s \triangleq (\mathsf{r}, \zeta, p)$, every region $\mathsf{r}^{-1}(q)$ recorded in the region function r has to be maximal, except, possibly, for the last one $\mathsf{r}^{-1}(p)$ depending on the flag ζ. This implies that $\mathsf{r}^{-1}(q) \subseteq \mathrm{Ps}_{\mathfrak{D}_{\mathsf{r}}^{\leq q}}$. Similarly, all such regions need to be saturated too, so that no priorities greater than the current one p are left unprocessed. Finally, p needs to be the current maximal priority in the game still to be analysed or the least priority of a region in r, when the latter has to be reprocessed after a promotion. As far as the order is concerned, a state s is strictly smaller than another state s' if either there is a region recorded in s that strictly contains the corresponding one in s' and all regions with higher priorities are equal in the two states, or s is obtained from s' as the result of a promotion operation.

At this point, we can determine the regions that are compatible with a given state. They are the only regions that the query function is allowed to return and that can then be used by the successor function and make the search progress in the region space. Intuitively, a region pair (R, α) is compatible with a state $s \triangleq (\mathsf{r}, \zeta, p)$ if it is an α-region in the subgame $\mathfrak{D}_s$ yet to be explored. Moreover, if such region is α-open in that game, it has to be α-maximal, its bep needs to be equal to the current priority p, and it has to necessarily contain the already computed region $\mathsf{r}^{-1}(p)$, in case the latter needs to be reprocessed, *i.e.*, if $\zeta = \mathtt{t}$. These three accessory properties ensure that the successor function is always able to cast R inside the current region function r and obtain a new state.

Definition 2.2.2 (Compatibility Relation for PP)**.** *An open region-pair* $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\mathfrak{D}}$ *is* PP compatible *with a state* $s \triangleq (\mathsf{r}, \zeta, p) \in \mathrm{S}^{\mathrm{PP}}_{\mathfrak{D}}$*, in symbols* $s \succ^{\mathrm{PP}}_{\mathfrak{D}} (\mathrm{R}, \alpha)$*, iff*

(1) $\mathrm{R} \subseteq \mathrm{Ps}_{\mathfrak{D}_s}$,

(2) *if* R *is* α*-open in* $\mathfrak{D}_s$ *then*

(2.a) R *is* α*-maximal in* $\mathfrak{D}_s$,

(2.b) *if* $\zeta = \mathtt{t}$ *then* $\mathsf{r}^{-1}(p) \subseteq \mathrm{R}$*, and*

(2.c) $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}_s}(\mathrm{R}) = p$.

Algorithm 5 provides a possible implementation for the query function compatible with the priority-promotion mechanism. Let $s \triangleq (\mathsf{r}, \zeta, p)$ be the current state. Line 1 simply computes the parity α of the priority to process in that state. Line 2 collects a new α-region. If s is the result of a promotion, *i.e.*, if $\zeta = \mathtt{t}$, then it extracts the α-region of priority p currently contained in r and resulting from that promotion. If $\zeta = \mathtt{f}$, instead, it gathers the set of positions with priority p in the subgame $\mathfrak{D}_s$ associated with that state. Finally, Line 3 computes the attractor R of the resulting set N *w.r.t.* the current player α in $\mathfrak{D}_s$. The result is, therefore, a maximal α-region. Observe that, due to the properties required on the states, the subgame $\mathfrak{D}_s$ does not contain any position with priorities higher than p. Moreover, the possible additional positions in R are only those attracted to N, and, if the latter is α-open, $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}_s}(\mathrm{N}) = p$. By the property of α-attractors, the only way the opponent can exit from R is via a path passing through N. Therefore, $\mathsf{bep}^{\overline{\alpha}}_{\mathfrak{D}_s}(\mathrm{R}) = p$, if R is α-open in $\mathfrak{D}_s$, as required by the compatibility relation.

Algorithm 5: Query Function.

signature $\Re_{\mathfrak{D}} : \mathrm{S}_{\mathfrak{D}} \to (2^{\mathrm{Ps}_{\mathfrak{D}}} \times \{0, 1\})$
function $\Re_{\mathfrak{D}}(s)$
  let $(\mathsf{r}, \zeta, p) = s$ **in**
1 $\alpha \leftarrow p \bmod 2$
2 $\mathrm{N} \leftarrow$ **if** ζ **then** $\mathsf{r}^{-1}(p)$ **else** $\mathsf{pr}^{-1}_{\mathfrak{D}_s}(p)$
3 $\mathrm{R} \leftarrow \mathsf{atr}^{\alpha}_{\mathfrak{D}_s}(\mathrm{N})$
4 **return** (R, α)

Algorithm 6: Successor Function.

signature $\downarrow_{\partial} : \succ_{\partial} \to \Delta_{\partial} \times \{\mathtt{t}, \mathtt{f}\} \times \mathrm{Pr}_{\partial}$
function $s{\downarrow_{\partial}}\,(\mathrm{R}, \alpha)$
  let $(\mathsf{r}, \_, p) = s$ **in**
1 **if** $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\partial_s}$ **then**
2 $\mathsf{r}^\star \leftarrow \mathsf{r}[\mathrm{R} \mapsto p]$
3 $p^\star \leftarrow \mathsf{pr}(\partial_s \backslash \mathrm{R})$
4 $\zeta^\star \leftarrow \mathtt{f}$
  else
5 $p^\star \leftarrow \mathsf{brp}^{\overline{\alpha}}_{\partial}(\mathrm{R}, \mathsf{r})$
6 $\mathsf{r}^\star \leftarrow \mathsf{r}^{(\geq p^\star)}[\mathrm{R} \mapsto p^\star]$
7 $\zeta^\star \leftarrow \mathtt{t}$
8 **return** $(\mathsf{r}^\star, \zeta^\star, p^\star)$

Before continuing with the description of the implementation of the successor function, we need to introduce the notion of *best region priority* for player $\overline{\alpha}$ *w.r.t.* an α-region R and a region function r in the whole game ∂. Informally, such a value represents the best priority associated with an α-region contained in r and reachable by $\overline{\alpha}$ when escaping from R. To formalize this concept, let $I \triangleq Mv_{\partial} \cap ((\mathrm{R} \cap \mathrm{Ps}^{\overline{\alpha}}_{\partial}) \times (\mathsf{dom}(\mathsf{r}) \backslash \mathrm{R}))$ be the *interface relation* between R and r, *i.e.*, the set of $\overline{\alpha}$-moves exiting from R and reaching some position within a region recorded in r. Then, $\mathsf{brp}^{\overline{\alpha}}_{\partial}(\mathrm{R}, \mathsf{r})$ is set to the minimal priority among those of the regions containing positions reachable by a move in I. Formally, $\mathsf{brp}^{\overline{\alpha}}_{\partial}(\mathrm{R}, \mathsf{r}) \triangleq \mathsf{min}(\mathsf{rng}(\mathsf{r} \restriction \mathsf{rng}(I)))$.

Algorithm 6 implements the successor function informally described at the beginning of the section. Given the current state s and an compatible region pair (R, α) open in the whole game as inputs, it produces a successor state $s^\star \triangleq (\mathsf{r}^\star, \zeta^\star, p^\star)$ in the region space. It first checks whether R is open also in the subgame ∂_s. If this is the case, it assigns priority p to region R and stores it in the new region function $\mathsf{r}^\star$ (Line 2). The new current priority $p^\star$ is computed as the highest priority in the residual subgame $\partial_s \setminus \mathrm{R}$ (Line 3). The new flag $\zeta^\star$ is set to $\mathtt{f}$ (Line 4), since the region in input does not require a promotion. If, on the other hand, R is closed in ∂_s, then a promotion merging R with some other α-region contained in r is required. The next priority $p^\star$ is then set to the brp of R for player $\overline{\alpha}$ in the entire game ∂ *w.r.t.* r (Line 5). Region R is promoted to priority $p^\star$ and all the promotions below $p^\star$

contained in the current region function r are revoked (Line 6). Finally, the flag $\zeta^\star$ of the successor state is set to $\mathtt{t}$ (Line 7), to mark $s^\star$ as the result of a promotion.

The following theorem provides the soddenness of the proposed priority-promotion mechanism.

Theorem 2.2.1 (PP Region Space)**.** *Given a game $\mathcal{D}$, the* PP structure $\mathcal{R}^{\mathrm{PP}}_{\mathcal{D}} \triangleq \langle \mathcal{D}, \mathcal{S}^{\mathrm{PP}}_{\mathcal{D}}, \succ^{\mathrm{PP}}_{\mathcal{D}}, \Re^{\mathrm{PP}}_{\mathcal{D}}, \downarrow^{\mathrm{PP}}_{\mathcal{D}} \rangle$, *where $\mathcal{S}^{\mathrm{PP}}_{\mathcal{D}}$ is the structure of Definition 2.2.1, $\succ^{\mathrm{PP}}_{\mathcal{D}}$ is the relation of Definition 2.2.2, and $\Re^{\mathrm{PP}}_{\mathcal{D}}$ and $\downarrow^{\mathrm{PP}}_{\mathcal{D}}$ are the functions computed by the Algorithms 5 and 6 with $\mathcal{S}_{\mathcal{D}} \triangleq \mathcal{S}^{\mathrm{PP}}_{\mathcal{D}}$ and $\succ_{\mathcal{D}} \triangleq \succ^{\mathrm{PP}}_{\mathcal{D}}$ is a region space.*

To prove the statement of Theorem 2.2.1, we have to show that the three components $\mathcal{S}^{\mathrm{PP}}_{\mathcal{D}}$, $\Re^{\mathrm{PP}}_{\mathcal{D}}$, and $\downarrow^{\mathrm{PP}}_{\mathcal{D}}$ of the structure $\mathcal{R}^{\mathrm{PP}}_{\mathcal{D}}$ satisfy the properties required by Definition 2.1.3 of region space. We do this through the Lemmas 2.2.1, 2.2.2, and 2.2.3.

Lemma 2.2.1 (State Space for PP)**.** *The* PP *state space $\mathcal{S}^{\mathrm{PP}}_{\mathcal{D}} = \langle \mathrm{S}^{\mathrm{PP}}_{\mathcal{D}}, \top^{\mathrm{PP}}_{\mathcal{D}}, \prec^{\mathrm{PP}}_{\mathcal{D}} \rangle$ for a game $\mathcal{D} \in \mathrm{PG}$ is a well-founded partial order* w.r.t. $\prec^{\mathrm{PP}}_{\mathcal{D}}$ *with maximal element $\top^{\mathrm{PP}}_{\mathcal{D}}$.*

Proof. Since $\mathrm{S}^{\mathrm{PP}}_{\mathcal{D}}$ is a finite set, to show that $\prec^{\mathrm{PP}}_{\mathcal{D}}$ is a well-founded partial order on $\mathrm{S}^{\mathrm{PP}}_{\mathcal{D}}$, it is enough to prove that it is simply a strict partial order on the same set, *i.e.*, an irreflexive and transitive relation.

For the irreflexive property, by Item 3 of Definition 2.2.1, it is immediate to see that $s \not\prec^{\mathrm{PP}}_{\mathcal{D}} s$, for all states $s \triangleq (\mathsf{r}, \zeta, \_) \in \mathrm{S}^{\mathrm{PP}}_{\mathcal{D}}$, since neither there exists a priority $q \in \mathsf{rng}(\mathsf{r})$ such that $\mathsf{r}^{-1}(q) \subset \mathsf{r}^{-1}(q)$ nor $\zeta \neq \zeta$.

For the transitive property, instead, consider three states $s_1 \triangleq (\mathsf{r}_1, \zeta_1, \_), s_2 \triangleq (\mathsf{r}_2, \zeta_2, \_), s_3 \triangleq (\mathsf{r}_3, \zeta_3, \_) \in \mathrm{S}^{\mathrm{PP}}_{\mathcal{D}}$ for which $s_1 \prec^{\mathrm{PP}}_{\mathcal{D}} s_2$ and $s_2 \prec^{\mathrm{PP}}_{\mathcal{D}} s_3$ hold. Due to Items 3.a and 3.b of the same definition, four cases can arise, only the first three of which are realizable.

- Item 3.a for both $s_1 \prec^{\mathrm{PP}}_{\mathcal{D}} s_2$ and $s_2 \prec^{\mathrm{PP}}_{\mathcal{D}} s_3$: there exist two priorities $q_1 \in \mathsf{rng}(\mathsf{r}_1)$ and $q_2 \in \mathsf{rng}(\mathsf{r}_2)$ such that ${\mathsf{r}_1}^{(>q_1)} = {\mathsf{r}_2}^{(>q_1)}$, ${\mathsf{r}_2}^{(>q_2)} = {\mathsf{r}_3}^{(>q_2)}$, $\mathsf{r}_2^{-1}(q_1) \subset \mathsf{r}_1^{-1}(q_1)$, and $\mathsf{r}_3^{-1}(q_2) \subset \mathsf{r}_2^{-1}(q_2)$. Let $q \triangleq \mathsf{max}\{q_1, q_2\}$. If $q = q_1 = q_2$ then ${\mathsf{r}_1}^{(>q)} = {\mathsf{r}_2}^{(>q)} = {\mathsf{r}_3}^{(>q)}$ and $\mathsf{r}_3^{-1}(q) \subset \mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$. If $q = q_1 > q_2$ then ${\mathsf{r}_1}^{(>q)} = {\mathsf{r}_2}^{(>q)} = ({\mathsf{r}_2}^{(>q_2)})^{(>q)} = ({\mathsf{r}_3}^{(>q_2)})^{(>q)} = {\mathsf{r}_3}^{(>q)}$ and $\mathsf{r}_3^{-1}(q) = \mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$. Finally, if $q = q_2 > q_1$ then ${\mathsf{r}_1}^{(>q)} = ({\mathsf{r}_1}^{(>q_1)})^{(>q)} = ({\mathsf{r}_2}^{(>q_1)})^{(>q)} = {\mathsf{r}_2}^{(>q)} = {\mathsf{r}_3}^{(>q)}$ and $\mathsf{r}_3^{-1}(q) \subset \mathsf{r}_2^{-1}(q) = \mathsf{r}_1^{-1}(q)$. Moreover, $q \in \mathsf{rng}\big({\mathsf{r}_2}^{(>q_1)}\big) = \mathsf{rng}\big({\mathsf{r}_1}^{(>q_1)}\big) \subseteq \mathsf{rng}(\mathsf{r}_1)$. Summing up, it holds that $s_1 \prec^{\mathrm{PP}}_{\mathcal{D}} s_3$.

- Item 3.a for $s_1 \prec^{\mathrm{PP}}_{\mathcal{D}} s_2$ and Item 3.b for $s_2 \prec^{\mathrm{PP}}_{\mathcal{D}} s_3$: there exists a priority $q \in \mathsf{rng}(\mathsf{r}_1)$ such that ${\mathsf{r}_1}^{(>q)} = {\mathsf{r}_2}^{(>q)}$ and $\mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$; moreover, $\mathsf{r}_2 = \mathsf{r}_3$. Thus, ${\mathsf{r}_1}^{(>q)} = {\mathsf{r}_2}^{(>q)} = {\mathsf{r}_3}^{(>q)}$ and $\mathsf{r}_3^{-1}(q) = \mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$. Consequently, $s_1 \prec^{\mathrm{PP}}_{\mathcal{D}} s_3$.

- Item 3.a for $s_2 \prec_\partial^{\mathrm{PP}} s_3$ and Item 3.b for $s_1 \prec_\partial^{\mathrm{PP}} s_2$: there exists a priority $q \in \mathsf{rng}(\mathsf{r}_2)$ such that ${\mathsf{r}_2}^{(>q)} = {\mathsf{r}_3}^{(>q)}$ and $\mathsf{r}_3^{-1}(q) \subset \mathsf{r}_2^{-1}(q)$; moreover, $\mathsf{r}_1 = \mathsf{r}_2$. Thus, ${\mathsf{r}_1}^{(>q)} = {\mathsf{r}_2}^{(>q)} = {\mathsf{r}_3}^{(>q)}$, $\mathsf{r}_3^{-1}(q) \subset \mathsf{r}_2^{-1}(q) = \mathsf{r}_1^{-1}(q)$, and $q \in \mathsf{rng}(\mathsf{r}_1)$. Consequently, $s_1 \prec_\partial^{\mathrm{PP}} s_3$.
- It is impossible to satisfy Item 3.b for both $s_1 \prec_\partial^{\mathrm{PP}} s_2$ and $s_2 \prec_\partial^{\mathrm{PP}} s_3$, since we would have had $\zeta_2 \neq \zeta_3$ but $\zeta_2 = \mathtt{t} = \zeta_3$, rising so a contradiction.

Finally, it is not hard to see that $\top_\partial^{\mathrm{PP}} \triangleq (\varnothing, \mathtt{f}, \mathsf{pr}(\partial))$ is the maximal element of $\mathrm{S}_\partial^{\mathrm{PP}}$ *w.r.t.* $\prec_\partial^{\mathrm{PP}}$. Indeed, let $(\mathsf{r}, \zeta, p) \in \mathrm{S}_\partial^{\mathrm{PP}}$ be a state such that $(\mathsf{r}, \zeta, p) \neq (\varnothing, \mathtt{f}, \mathsf{pr}(\partial))$. By Items 1.b and 1.c of Definition 2.2.1, it holds that $\mathsf{r} \neq \varnothing$, otherwise, would have had $\zeta = \mathtt{f}$ and $p = \mathsf{pr}(\partial)$, which contradict our assumption. Hence, $(\mathsf{r}, \zeta, p) \prec_\partial^{\mathrm{PP}} (\varnothing, \mathtt{f}, \mathsf{pr}(\partial))$, by Item 3.a of the same definition. Moreover, there is no state $(\mathsf{r}, \zeta, p) \in \mathrm{S}_\partial^{\mathrm{PP}}$ such that $(\varnothing, \mathtt{f}, \mathsf{pr}(\partial)) \prec_\partial^{\mathrm{PP}} (\mathsf{r}, \zeta, p)$. Indeed, again by Item 3, the only possibility is to have $\mathsf{r} = \varnothing$ and $\zeta = \mathtt{t}$. However, this contradicts the requirement of Item 1.b. □

Lemma 2.2.2 (Query Function for PP)**.** *The function* $\Re_\partial^{\mathrm{PP}}$ *is an query function,* i.e.*, for all states* $s \in \mathrm{S}_\partial^{\mathrm{PP}}$*, it holds that* (1) $\Re_\partial^{\mathrm{PP}}(s) \in \mathrm{Rg}_\partial$ *and* (2) *if* $\Re_\partial^{\mathrm{PP}}(s) \in \mathrm{Rg}_\partial^-$ *then* $s \succ_\partial^{\mathrm{PP}} \Re_\partial^{\mathrm{PP}}(s)$*.*

Proof. Let $s \triangleq (\mathsf{r}, \zeta, p) \in \mathrm{S}_\partial^{\mathrm{PP}}$ be a state and $(\mathrm{R}, \alpha) \triangleq \Re_\partial^{\mathrm{PP}}(s)$ the pair of a set of positions $\mathrm{R} \subseteq \mathrm{Ps}_\partial$ and a player $\alpha \in \{0, 1\}$ obtained by computing the function $\Re_\partial^{\mathrm{PP}}$ on s. Due to Line 1 of Algorithm 5, it holds that $\alpha \equiv_2 p$.

If $\zeta = \mathtt{f}$, by Item 1.b of Definition 2.2.1, we have that $p = \mathsf{pr}(\partial_s)$. Thus, by Lines 2-3 of the same algorithm and Proposition 2.2.2, it follows that R is an α-region in ∂_s and, so, in ∂. Hence, $\mathrm{R} \subseteq \mathrm{Ps}_{\partial_s}$ and $(\mathrm{R}, \alpha) \in \mathrm{Rg}_\partial$. In addition, R is α-maximal in ∂_s and, if α-open, it also satisfies the equality $\mathsf{bep}_{\partial_s}^{\overline{\alpha}}(\mathrm{R}) = p$. Consequently, if $(\mathrm{R}, \alpha) \in \mathrm{Rg}_\partial^-$ then $s \succ_\partial^{\mathrm{PP}} (\mathrm{R}, \alpha)$, since all requirements of Definition 2.2.2 are satisfied.

If $\zeta = \mathtt{t}$, by Item 1.b of Definition 2.2.1, it holds that $p \in \mathsf{rng}(\mathsf{r})$. Thus, by Item 1.a and the definition of region function, we have that $\mathsf{r}^{-1}(p)$ is an α-region in $\partial_{\mathsf{r}}^{\leq p}$, the latter being equal to ∂_s, with $\mathsf{bep}_{\partial_s}^{\overline{\alpha}}(\mathsf{r}^{-1}(p)) \in \{p, \infty\}$. At this point, by Lines 2-3 of Algorithm 5 and Proposition 2.2.3 with $\mathrm{R}^\star \triangleq \mathsf{r}^{-1}(p)$, it follows that $\mathrm{R} \supseteq \mathrm{R}^\star$ is an α-region in ∂_s and, so, in ∂. Hence, $\mathrm{R} \subseteq \mathrm{Ps}_{\partial_s}$ and $(\mathrm{R}, \alpha) \in \mathrm{Rg}_\partial$. In addition, R is α-maximal in ∂_s and, if α-open, it also satisfies the equality $\mathsf{bep}_{\partial_s}^{\overline{\alpha}}(\mathrm{R}) = \mathsf{bep}_{\partial_s}^{\overline{\alpha}}(\mathrm{R}^\star) = p$. Consequently, if $(\mathrm{R}, \alpha) \in \mathrm{Rg}_\partial^-$ then $s \succ_\partial^{\mathrm{PP}} (\mathrm{R}, \alpha)$, since all requirements of Definition 2.2.2 are satisfied. □

Lemma 2.2.3 (Successor Function for PP)**.** *The function* $\downarrow_\partial^{\mathrm{PP}}$ *is an successor function,* i.e.*, for all states* $s \in \mathrm{S}_\partial^{\mathrm{PP}}$ *and region pairs* $(\mathrm{R}, \alpha) \in \mathrm{Rg}_\partial^-$ *with* $s \succ_\partial^{\mathrm{PP}} (\mathrm{R}, \alpha)$*, it holds that* (1) $s \downarrow_\partial^{\mathrm{PP}} (\mathrm{R}, \alpha) \in \mathrm{S}_\partial^{\mathrm{PP}}$ *and* (2) $s \downarrow_\partial^{\mathrm{PP}} (\mathrm{R}, \alpha) \prec_\partial^{\mathrm{PP}} s$*.*

Proof. Let $s \triangleq (\mathsf{r}, \zeta, p) \in \mathrm{S}_\partial^{\mathrm{PP}}$ be a state, $(\mathrm{R}, \alpha) \in \mathrm{Rg}_\partial^-$ an open region-pair in ∂ that is PP compatible with s, and $s^\star = (\mathsf{r}^\star, \zeta^\star, p^\star) \triangleq s \downarrow_\partial^{\mathrm{PP}} (\mathrm{R}, \alpha)$ the result obtained by computing the function $\downarrow_\partial^{\mathrm{PP}}$ on s and

(R, α). Due to Item 1 of Definition 2.2.1, we have that *(1)* $\mathsf{r}^{(\geq p(\zeta))}$ is maximal, *(2)* if $\zeta = \mathtt{f}$ then $p = \mathsf{pr}(\partial_s)$ else $p \in \mathsf{rng}(\mathsf{r})$, *(3)* r is saturated, and *(4)* $p \leq \mathsf{min}(\mathsf{rng}(\mathsf{r}))$. Moreover, by Item 1 of Definition 2.2.2, it holds that $\mathrm{R} \subseteq \mathrm{Ps}_{\partial_s}$, which implies *(5)* $\mathsf{dom}\big(\mathsf{r}^{(\geq p(\zeta))}\big) \cap \mathrm{R} = \emptyset$.

On the one hand, suppose that R is α-open in ∂_s, *i.e.*, $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\partial_s}$. By Lines 1-4 of Algorithm 6, we have that *(6)* $\mathsf{r}^\star = \mathsf{r}[\mathrm{R} \mapsto p]$, *(7)* $\zeta^\star = \mathtt{f}$, and *(8)* $p^\star = \mathsf{pr}(\partial_s \backslash \mathrm{R})$. Also, by Item 2 of Definition 2.2.2, it holds that *(9)* R is α-maximal in ∂_s, *(10)* if $\zeta = \mathtt{t}$ then $\mathsf{r}^{-1}(p) \subseteq \mathrm{R}$, and *(11)* $\alpha \equiv_2 p = \mathsf{bep}^{\overline{\alpha}}_{\partial_s}(\mathrm{R})$. First note that, by Points (4) and (6), it holds that *(12)* $\mathsf{r}^\star = \mathsf{r}^{(>p)}[\mathrm{R}^\star \mapsto p]$ with $\mathrm{R}^\star \triangleq \mathrm{R} \cup \mathsf{r}^{-1}(p)$, and *(13)* $\mathsf{min}(\mathsf{rng}(\mathsf{r}^\star)) = p$. It is easy to see that *(14)* $\mathrm{R}^\star$ is a saturated α-maximal α-region in $\partial^{\leq p}_{\mathsf{r}}$ with $\mathsf{bep}^{\overline{\alpha}}_{\partial^{\leq p}_{\mathsf{r}}}(\mathrm{R}^\star) = p$. Indeed, if $\zeta = \mathtt{t}$ or $p \notin \mathsf{rng}(\mathsf{r})$ then $\mathrm{R}^\star = \mathrm{R}$, so, its maximality is due to Point (9) and the value of best escape priority to Point (11). Moreover, the saturation follows from Points (2) and (11), if $p \notin \mathsf{rng}(\mathsf{r})$, and from Points (3) and (10), if $\zeta = \mathtt{t}$. In case $\zeta = \mathtt{f}$ and $p \in \mathsf{rng}(\mathsf{r})$, instead, the required properties follow from Proposition 2.2.1, whose hypothesis are derived by Points (1), (3), (9), and (11). Consequently, by the definition of region function and Points (1), (3), (11), (12), and (14), we have that *(15)* $\mathsf{r}^\star$ is a maximal and saturated region function in ∂. Now, it is obvious that $\partial_s \setminus \mathrm{R}$ is a subgame of ∂_s, which implies that $p^\star = \mathsf{pr}(\partial_s \backslash \mathrm{R}) \leq \mathsf{pr}(\partial_s) = p$, due to Points (2) and (8) and the monotonicity of the priority function on games. Thus, by Points (7) and (13), we have that $\mathsf{r}^\star = \mathsf{r}^{\star(\geq p^\star(\zeta^\star))}$ and $p^\star = \mathsf{pr}(\partial_{s^\star})$, from which, together with Point (15), we derive that $s^\star \in \mathrm{S}^{\mathrm{PP}}_{\partial}$. At this point, it remains just to show that $(\mathsf{r}^\star, \zeta^\star, p^\star) \prec^{\mathrm{PP}}_{\partial} (\mathsf{r}, \zeta, p)$. If $\mathsf{r}^{-1}(p) \subset \mathrm{R}^\star$, the thesis follows from Item 3.a of Definition 2.2.1, where the priority q is set to p. On the contrary, because of Point (5), if $\mathrm{R}^\star = \mathsf{r}^{-1}(p)$ and, so, $\mathsf{r}^\star = \mathsf{r}$, we necessarily have $\zeta = \mathtt{t}$. Therefore, the thesis is derived from Item 3.b of the same definition.

On the other hand, suppose that R is α-closed in ∂_s, *i.e.*, $(\mathrm{R}, \alpha) \notin \mathrm{Rg}^{-}_{\partial_s}$. By Lines 1, 5-7 of Algorithm 6, we have that *(16)* $p^\star = \mathsf{brp}^{\overline{\alpha}}_{\partial}\big(\mathrm{R}, \mathsf{r}^{(\geq p(\zeta))}\big)$, *(17)* $\mathsf{r}^\star = \mathsf{r}^{(\geq p^\star)}[\mathrm{R} \mapsto p^\star]$, and *(18)* $\zeta^\star = \mathtt{t}$. By the last two points, it is immediate to see that Items 3.b and 3.d of Definition 2.2.1 hold. Moreover, due to the first point, we have that $p(\zeta) \leq p^\star$. Consequently, by Points (1), (17), and (18), it follows that $\mathsf{r}^{\star(\geq p^\star(\zeta^\star))}$ is maximal, since $p^\star < p^\star(\zeta^\star)$. Thus, Item 3.a of the same definition holds as well. Finally, to prove that $s^\star \in \mathrm{S}^{\mathrm{PP}}_{\partial}$, we only need to show that $\mathsf{r}^\star$ is a region function, which also satisfies Item 3.c, *i.e.*, it is saturated. To do this, first observe that $\mathsf{r}^\star = \mathsf{r}^{(>p^\star)}[\mathrm{R}^\star \mapsto p^\star]$ with $\mathrm{R}^\star \triangleq \mathrm{R} \cup \mathsf{r}^{-1}(p^\star)$. Now, by Proposition 2.2.1, we have $\mathrm{R}^\star$ is a saturated α-region in $\partial^{\leq p^\star}_{\mathsf{r}}$ with $\mathsf{bep}^{\overline{\alpha}}_{\partial^{\leq p^\star}_{\mathsf{r}}}(\mathrm{R}^\star) \in \{p^\star, \infty\}$. Therefore, since $\mathsf{r}^{(>p^\star)}$ is a saturated region function, the thesis immediately follows. At this point, as for the previous case, it remains to show that $(\mathsf{r}^\star, \zeta^\star, p^\star) \prec^{\mathrm{PP}}_{\partial} (\mathsf{r}, \zeta, p)$. This fact easily follows from Item 3.a of Definition 2.2.1, where the priority q is set to $p^\star$, since, by Point (5), we have that $\mathsf{r}^{-1}(p) \subset \mathrm{R}^\star$. □

To conclude, we estimate the size and depth of region space $\mathcal{R}^{\mathrm{PP}}_{\partial}$. This provides upper bounds

on both the time and space needed by the search procedure $\mathsf{win}_{\mathcal{R}^{\mathrm{PP}}_{\mathfrak{D}}}$ computing winning regions. By looking at the definition of state space $\mathcal{S}^{\mathrm{PP}}_{\mathfrak{D}}$, it is immediate to see that, for a game $\mathfrak{D}$ with n positions and k priorities, the number of states is bounded by $2k^{n+1}$. Indeed, there are at most k^{n+1} partial functions $\mathsf{r} : \mathrm{Ps}_{\mathfrak{D}} \rightharpoonup \mathrm{Pr}_{\mathfrak{D}}$ from positions to priorities that can be used as region function of a state. Note that the associated current priority is uniquely determined by the content of the region function and the flag. Measuring the depth is a little trickier. A coarse bound can be obtained by observing that there is an homomorphism from $\mathcal{S}^{\mathrm{PP}}_{\mathfrak{D}}$ to the well-founded partial order in which the region function r of a state is replaced by a partial function $\mathsf{f} : \mathrm{Pr}_{\mathfrak{D}} \to [1, n]$ with the following properties. It assigns to each priority $p \in \mathsf{rng}(\mathsf{r})$ the size $\mathsf{f}(p)$ of the associated region $\mathsf{r}^{-1}(p)$. The order $(\mathsf{f}_1, \zeta_1, p_1) \prec (\mathsf{f}_2, \zeta_2, p_2)$ between two triples is derived from the one on the states, by replacing $\mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$ with $\mathsf{f}_2(q) < \mathsf{f}_1(q)$. This homomorphism ensures that every chain in $\mathcal{S}^{\mathrm{PP}}_{\mathfrak{D}}$ corresponds to a chain in the new partial order. Moreover, there are exactly $\binom{n+k}{k}$ partial functions f such that $\sum_{p \in \mathsf{dom}(\mathsf{f})} \mathsf{f}(p) \leq n$. Consequently, every chain cannot be longer than $2\binom{n+k}{k} \leq 2\left(\mathsf{e}(\frac{n}{k} + 1)\right)^k$, where e is the Euler constant. By further exploiting the structure of the space, one can obtain a recurrence relation expressing a slightly better upper bound, whose explicit solution is $3 \cdot \sum_{i=0}^{k-2} \binom{n-2}{i}$. Then, by applying a standard approximation via geometric series based on the inequality $\left(\frac{h-i}{n-h+i}\right)^i \leq \binom{m}{h-i} / \binom{m}{h} \leq \left(\frac{h}{n-h+1}\right)^i$, we derive the asymptotic bound stated in the following theorem.

Theorem 2.2.2 (PP Size & Depth Upper Bounds)**.** *The size of a* PP *region space* $\mathcal{R}^{\mathrm{PP}}$ *with* $n \in \mathbb{N}_+$ *positions and* $k \in [1, n]$ *priorities is bounded by* $2k^{n+1}$*. Moreover, if* $2 \leq k$*, its depth is bounded by* $3 \cdot \sum_{i=0}^{k-2} \binom{n-2}{i}$*, which is less than* $3\frac{n-k+1}{n-2k+3}\left(\mathsf{e}\frac{n-2}{k-2}\right)^{k-2}$*, if* $k < n/2$*, and less than* $3(2^{n-2} - c(\frac{n-2}{k-2})^{k-2})$*, for a constant* $c > 0$*, otherwise.*

Unfortunately, due to the promotion revocation policy, an exponential worst-case can actually be built. Indeed, consider the game $\mathfrak{D}^{\mathrm{PP}}_{m,h}$ having all positions ruled by player 0 and containing h chains of length $2m + 1$ that converge into a single position of priority 0 with a self loop. The i-th chain has a head of priority $4k - i$ and a body composed of m blocks of two positions having priority $2i - 1$ and $2i$, respectively. The first position in each block also has a self loop. An instance of this game with $m = 2$ and $h = 4$ is depicted in Figure 2.3. The labels of the positions correspond to the associated priorities and the highlighted area at the bottom of the figure groups together the last blocks of the chains. Intuitively, the execution depth of the PP region space for this game is exponential, since the consecutive promotion operations performed on each chain can simulate the increments of a counter up to m. Also, the priorities are chosen in such a way that, when the i-th counter is incremented, all the j-th counters with $j \in]i, h]$ are reset. Therefore, the whole game simulates a counter with h digits taking

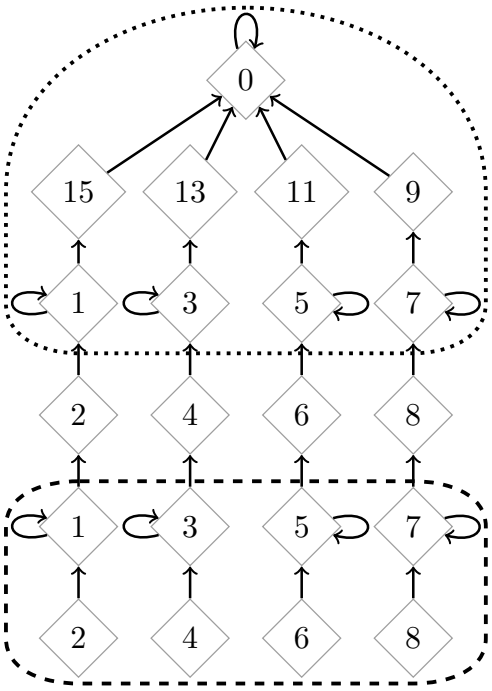


Figure 2.3: The $\mathfrak{I}^{\mathrm{PP}}_{2,4}$ game.

values from 0 to m. Hence, that the overall number of performed promotions is $(m+1)^h$. The search procedure on $\mathfrak{I}^{\mathrm{PP}}_{2,4}$ starts by building the four open 1-regions $\{15\}$, $\{13\}$, $\{11\}$, and $\{9\}$ and the open 0-region $\{8', 7'', 8''\}$, where we use apices to distinguish different positions with the same priority. This state represents the configuration of the counter, where all four digits are set to 0. The closed 1-region $\{7'\}$ is then found and promoted to 9. Consequently, the previously computed 0-region with `bep` equal to 8 is destroyed and the new region is maximized to obtain the open 1-region $\{9, 7', 8'\}$. Now, the counter is set to 0001. After that, the open 0-region $\{8''\}$ and the closed 1-region $\{7''\}$ are computed. The latter one is promoted to 9 and maximized to attract position $8''$. This completes the 1-region containing the entire chain ending in 9. The value of the counter is now 0002. At this point, immediately after the construction of the open 0-region $\{6', 5'', 6''\}$, the closed 1-region $\{5'\}$ is found, promoted to 11, and maximized to absorb position $6'$. Due to the revocation policy, the 1-region with `bep` equal to 9 is also deleted and all the work done to build it gets lost. This last operation represents the reset of the least significant digit of the counter, caused by the increment of the second one, *i.e.*, the counter displays 0010. Following similar steps, the process carries on until each chain is grouped under a unique region. The

corresponding state represents the configuration of the counter in which all digits are set to m. Thus, after an exponential number promotions, the closed 0-region $\{0\}$ is eventually obtained as solution of the search.

Theorem 2.2.3 (PP Execution-Depth Lower Bounds)**.** *For all numbers $h \in \mathbb{N}$, there exists a* PP *region space $\mathcal{R}_h^{\mathrm{PP}}$ with $k = 2h+1$ positions and priorities, whose execution depth is $3 \cdot 2^h - 2 = \Theta\big(2^{k/2}\big)$. Moreover, for all numbers $m \in \mathbb{N}_+$, there exists a* PP *region space $\mathcal{R}_{m,h}^{\mathrm{PP}}$ with $n = (2m+1) \cdot h + 1$ positions and $k = 3h+1$ priorities, whose execution depth is $((3m+1) \cdot (m+1)^h - 1)/m - 2 = \mathrm{O}\Big((3n/(2(k-1)))^{k/3}\Big)$.*

Observe that, in the above theorem we provide two different exponential lower bounds. The general one, with $k/3$ as exponent and a parametric base, is the result of the game $\mathfrak{I}_{m,h}^{\mathrm{PP}}$ described in the previous paragraph, where $k = 3h + 1$. Its proof easily follows after observing that the execution depth $\mathsf{d}(m, h)$ of the associated PP region spaces $\mathcal{R}_{m,h}^{\mathrm{PP}}$ satisfies the following recurrence relation: $\mathsf{d}(m, 0) = 1$ and $\mathsf{d}(m, h+1) = (m+1) \cdot (\mathsf{d}(m, h) + 2) - 1$. Intuitively, each chain added to a game requires a number of operations that is almost $m + 1$ times that of the original game plus 2. The other bound, instead, has a base fixed to 2, but the worse exponent $k/2$. It can be obtained by analysing the PP region space $\mathcal{R}_h^{\mathrm{PP}}$ associated with the game $\mathfrak{I}_h^{\mathrm{PP}}$. This is derived from $\mathfrak{I}_{m,h}^{\mathrm{PP}}$ by contracting each chain of index i to its head and the first position of the body with priority $2i - 1$. The top most area surrounded by the dotted line in Figure 2.3 graphically represents the instance with $h = 4$. In this case, the execution depth $\mathsf{d}(h)$ satisfies the simpler recurrence relation $\mathsf{d}(h+1) = 2(\mathsf{d}(h) + 1)$ with $\mathsf{d}(0) = 1$. We conjecture that the given upper bound could be improved to match the exponent $k/2$ of this lower bound. In this way, we would obtain an algorithm with an asymptotic behaviour comparable with the one exhibited by the small-progress measure procedure [Jur00].

2.2.2 Parity Memoization

As we have shown in the previous subsection, revocation of promotions and, more generally, the deletion of regions cannot be avoided in general. This leads, in some cases, to the destruction and rebuilding of the same regions several times, as it happens for 0-region $\{\mathsf{c}, \mathsf{e}\}$ of the game depicted in Figure 2.3. However, not all such destructions are actually necessary. In fact, consider the situation where a promotion to an α-region of priority $p^\star$ is required. Once the promotion is performed, the maximization of the new region ensures that all the attracted positions are added to it. Obviously, some of these may be part of a previously computed region of lower priority having parity $\overline{\alpha}$. Thus, to ensure the correctness of the procedure, we need to destroy it, since the remaining part may not satisfy the definitional constraints of $\overline{\alpha}$-regions. The situation is different if the positions are taken from a region of the same parity α. In

this case, indeed, the remaining part, if non-empty, still forms an α-region. In fact, the original strategy restricted to the non-attracted positions still satisfies the constraints required by Definition 2.1.2. In addition, when open, it preserves the same bep of the original region. This crucial property is formalized by the following proposition.

Proposition 2.2.4 (Region Splitting)**.** *Let* $\partial \in \mathrm{PG}$ *be a game,* $(\mathrm{R}, \alpha) \in \mathrm{Rg}_{\partial}$ *a region pair, and* $\mathrm{R}^\star \subseteq \mathrm{Ps}_{\partial}$ *an* α*-maximal* α*-region such that* $\mathrm{R}^\natural \triangleq \mathrm{R} \setminus \mathrm{R}^\star \neq \emptyset$*. Then,* $\mathrm{R}^\natural$ *is an* α*-region in* $\partial \setminus \mathrm{R}^\star$ *with* $\mathsf{bep}^{\overline{\alpha}}_{\partial \setminus \mathrm{R}^\star}(\mathrm{R}^\natural) \in \{\mathsf{bep}^{\overline{\alpha}}_{\partial}(\mathrm{R}), \infty\}$.

The above observation allows us to restrict promotion revocations only to the $\overline{\alpha}$-regions whose priorities are lower than $p^\star$. In this way, we obtain a variant of the successor algorithm exploiting a form of memoization and, therefore, called *parity-memoization mechanism* (PM, for short). With more details, the region function of the successor state always retains, besides the regions with priorities greater than or equal to $p^\star$, all those regions below $p^\star$ with priorities of the same parity as $p^\star$. To formalize this concept, Line 6 of Algorithm 6 needs to be replaced by "$\mathsf{r}^\star \leftarrow \mathsf{r}^{(\geq p^\star) \vee (\equiv_2 p^\star)}[\mathrm{R} \mapsto p^\star]$", where $\mathsf{r}^{(\equiv_2 p^\star)}$ denotes the region function obtained by removing all the regions except for those with priorities congruent to $p^\star$. In addition, to determine the next priority to process, we need to change Line 3 into "$p^\star \leftarrow \mathsf{max}\{\mathsf{pr}(\partial_s \setminus \mathrm{R}), \mathsf{max}(\mathsf{rng}(\mathsf{r}^{(<p)}))\}$". In this way, we take into account the maximal priority in both the subgame and preserved part of the region function. This also implies that Line 4 need to be replaced by "$\zeta^\star \leftarrow p^\star < p \wedge p^\star \in \mathsf{rng}(\mathsf{r}^\star)$" to ensure that, once we move to a lower priority assigned to a preserved region, such a region is reconsidered.

	1	2	3	4	5
6	$\mathtt{a}\downarrow$	$\cdots$	$\cdots$	$\cdots$	$\mathtt{a},\mathtt{b},\mathtt{g},\mathtt{h}\downarrow$
5	$\mathtt{b},\mathtt{e},\mathtt{f}\downarrow$	$\cdots$	$\mathtt{b},\mathtt{e},\mathtt{f},\mathtt{g}\downarrow$	$\cdots$	
4	$\mathtt{c}\downarrow$	$\mathtt{c},\mathtt{d}\downarrow$	$\mathtt{c}\downarrow$	$\mathtt{c},\mathtt{d}\downarrow$	$\mathtt{c},\mathtt{d}\uparrow_6$
2	$\mathtt{d}\uparrow_4$		$\mathtt{d}\uparrow_4$		
1		$\mathtt{g}\uparrow_5$			
0				$\mathtt{h}\uparrow_6$	

Table 2.2: PM simulation.

To exemplify the idea, Table 2.2 shows a simulation of the resulting procedure on the game of Figure 2.2. The simulation runs exactly as in Table 2.1 up until Column 5. When the promotion of

0-region $\{\mathtt{i}\}$ to priority 6 is performed, however, 0-region $\{\mathtt{c},\mathtt{e}\}$ with priority 4 is not removed, leading directly to the configuration in Column 6. Then, being $\{\mathtt{c},\mathtt{e}\}$ closed in the corresponding subgame, it is promoted to priority 6. This results in the closed 0-region $\{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e},\mathtt{g},\mathtt{i}\}$, which is winning for player 0. In this case, to obtain a winning region, even though different from the one obtained by the previous algorithm, one less promotion and two less region constructions are necessary. The main difference between PP and PM is that, when a promotion is performed, the former removes from the region function all the regions below priority $p^\star$, to which the promotion was performed, while the latter retains all the regions with priorities congruent to $p^\star$. The soundness of this operation relies on the observation that future open regions of the opponent, later obtained by the query function applied to the corresponding subgames, cannot attract positions already contained in the retained regions. This additional property of the region function is formalized by the notion of *memoization consistency*. We say that a region function r is *memoization consistent w.r.t.* a priority p iff, for all open region-pairs $(\mathrm{R},\alpha) \in \mathrm{Rg}^{-}_{\eth_{\mathsf{r}}^{\leq p}}$ and priorities $q \in \mathsf{rng}(\mathsf{r})$ with $q \not\equiv_2 \alpha$ and $q < \mathsf{bep}^{\overline{\alpha}}_{\eth_{\mathsf{r}}^{\leq p}}(\mathrm{R})$, it holds that $\mathsf{r}^{-1}(q) \cap \mathrm{R} = \emptyset$.

The differences between PP and PM is, thus, accounted for in the definition of the state space by the following modifications. We substitute the stronger condition 1.d in Definition 2.2.1 with the weaker one requiring memoization consistency of the region function *w.r.t.* the current priority of the state. As a consequence of this change, the ordering relation, specifically condition b.ii, has to be adjusted. This modification accounts for the fact that the query function may need to extract the next region directly from r, when the priority of a retained region is processed. If, however, that region is already maximal, as no additional position can be attracted, the operation would leave the region function unchanged in the next state. Hence, to ensure that also in this case the resulting state is smaller that the original one, we require that the priority of the former be smaller than that of the latter. Finally, since now r may contain regions below the current priority p retained after a promotion, the saturation requirement for r in PP needs to be limited to the portion of r up to p itself. These observations results in the following definition of state space for PM.

Definition 2.2.3 (State Space for PM)**.** *A* PM state space *is a tuple* $\mathcal{S}^{\mathrm{PM}}_{\eth} \triangleq \langle \mathrm{S}^{\mathrm{PM}}_{\eth}, \top^{\mathrm{PM}}_{\eth}, \prec^{\mathrm{PM}}_{\eth}\rangle$*, where its components are defined as prescribed in the following:*

1. $\mathrm{S}^{\mathrm{PM}}_{\eth} \subseteq \mathbb{R}_{\eth} \times \{\mathtt{t},\mathtt{f}\} \times \mathrm{Pr}_{\eth}$ *is the set of all triples* $s \triangleq (\mathsf{r},\zeta,p)$*, called* states*, composed by a region function* $\mathsf{r}\in\mathbb{R}_{\eth}$*, a Boolean flag* $\zeta\in\{\mathtt{t},\mathtt{f}\}$*, and a priority* $p\in\mathrm{Pr}_{\eth}$ *such that* (a) $\mathsf{r}^{(\geq p(\zeta))}$ *is maximal,* (b) *if* $\zeta = \mathtt{f}$ *then* $p = \mathsf{pr}(\eth_s)$*, else* $p \in \mathsf{rng}(\mathsf{r})$*,* (c) $\mathsf{r}^{(\geq p)}$ *is saturated, and* (d) r *is memoization consistent* w.r.t. p*;*

2. $\top^{\mathrm{PM}}_{\eth} \triangleq (\varnothing,\mathtt{f},\mathsf{pr}(\eth))$*;*

3. *for any two states* $s_1 \triangleq (\mathsf{r}_1, \zeta_1, p_1), s_2 \triangleq (\mathsf{r}_2, \zeta_2, p_2) \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{PM}}$*, it holds that* $s_1 \prec_{\mathfrak{D}}^{\mathrm{PM}} s_2$ *iff either* (a) *there exists a priority* $q \in \mathsf{rng}(\mathsf{r}_1)$ *with* $p_1 \leq q$ *such that* (a.i) $\mathsf{r}_1{}^{(>q)} = \mathsf{r}_2{}^{(>q)}$ *and* (a.ii) $\mathsf{r}_2^{-1}(q) \subset \mathsf{r}_1^{-1}(q)$*, or* (b) *both* (b.i) $\mathsf{r}_1 = \mathsf{r}_2$ *and* (b.ii) $p_1 < p_2$ *or* $p_1 = p_2$ *and* $\zeta_1 \neq \zeta_2 = \mathtt{t}$ *hold.*

Algorithm 7 reports the pseudo-code of the successor function.

Algorithm 7: Successor Function.

signature $\downarrow_{\mathfrak{D}} : \succ_{\mathfrak{D}} \to \Delta_{\mathfrak{D}} \times \{\mathtt{t}, \mathtt{f}\} \times \mathrm{Pr}_{\mathfrak{D}}$
function $s \downarrow_{\mathfrak{D}} (\mathrm{R}, \alpha)$
  let $(\mathsf{r}, \_, p) = s$ **in**
1 **if** $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\mathfrak{D}_s}$ **then**
2 $\mathsf{r}^\star \leftarrow \mathsf{r}[\mathrm{R} \mapsto p]$
3 $p^\star \leftarrow \mathsf{max}\{\mathsf{pr}(\mathfrak{D}_s \backslash \mathrm{R}), \mathsf{max}(\mathsf{rng}(\mathsf{r}^{(<p)}))\}$
4 $\zeta^\star \leftarrow p^\star < p \wedge p^\star \in \mathsf{rng}(\mathsf{r}^\star)$
  else
5 $p^\star \leftarrow \mathsf{brp}^{\overline{\alpha}}_{\mathfrak{D}}(\mathrm{R}, \mathsf{r})$
6 $\mathsf{r}^\star \leftarrow \mathsf{r}^{(\geq p^\star) \vee (\equiv_2 p^\star)}[\mathrm{R} \mapsto p^\star]$
7 $\zeta^\star \leftarrow \mathtt{t}$
8 **return** $(\mathsf{r}^\star, \zeta^\star, p^\star)$

The following theorem asserts that the PM state space defined above, together with the same query function as for PP and the successor function of Algorithm 7 is indeed a region space.

Theorem 2.2.4 (PM Region Space)**.** *Given a game* $\mathfrak{D}$*, the* PM structure $\mathcal{R}_{\mathfrak{D}}^{\mathrm{PM}} \triangleq \langle \mathfrak{D}, \mathcal{S}_{\mathfrak{D}}^{\mathrm{PM}}, \succ_{\mathfrak{D}}^{\mathrm{PM}}, \Re_{\mathfrak{D}}^{\mathrm{PM}}, \downarrow_{\mathfrak{D}}^{\mathrm{PM}} \rangle$*, where* $\mathcal{S}_{\mathfrak{D}}^{\mathrm{PM}}$ *is the structure of Definition 2.2.3,* $\succ_{\mathfrak{D}}^{\mathrm{PM}}$ *is the relation formalized as in Definition 2.2.2 with* $\mathcal{S}_{\mathfrak{D}}^{\mathrm{PP}}$ *replaced by* $\mathcal{S}_{\mathfrak{D}}^{\mathrm{PM}}$*, and* $\Re_{\mathfrak{D}}^{\mathrm{PM}}$ *and* $\downarrow_{\mathfrak{D}}^{\mathrm{PM}}$ *are the functions computed by the Algorithms 5 and 7 with* $\mathcal{S}_{\mathfrak{D}} \triangleq \mathcal{S}_{\mathfrak{D}}^{\mathrm{PM}}$ *and* $\succ_{\mathfrak{D}} \triangleq \succ_{\mathfrak{D}}^{\mathrm{PM}}$ *is a region space.*

Similarly to the previous subsection, Theorem 2.2.4 is proved by splitting into the Lemmas 2.2.4, 2.2.5, and 2.2.6 the verification of the properties required by Definition 2.1.3 on the three components $\mathcal{S}_{\mathfrak{D}}^{\mathrm{PM}}$, $\Re_{\mathfrak{D}}^{\mathrm{PM}}$, and $\downarrow_{\mathfrak{D}}^{\mathrm{PM}}$ of the structure $\mathcal{R}_{\mathfrak{D}}^{\mathrm{PM}}$.

Lemma 2.2.4 (State Space for PM)**.** *The* PM *state space* $\mathcal{S}_{\mathfrak{D}}^{\mathrm{PM}} = \langle \mathrm{S}_{\mathfrak{D}}^{\mathrm{PM}}, \top_{\mathfrak{D}}^{\mathrm{PM}}, \prec_{\mathfrak{D}}^{\mathrm{PM}} \rangle$ *for a game* $\mathfrak{D} \in \mathrm{PG}$ *is a well-founded partial order* w.r.t. $\prec_{\mathfrak{D}}^{\mathrm{PM}}$ *with maximal element* $\top_{\mathfrak{D}}^{\mathrm{PM}}$*.*

Proof. As for Lemma 2.2.1, since $\mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$ is a finite set, to show that $\prec^{\mathrm{PM}}_{\mathfrak{D}}$ is a well-founded partial order on $\mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$, it is enough to prove that it is an irreflexive and transitive relation on the same set.

For the irreflexive property, by Item 3 of Definition 2.2.3, it is immediate to see that $s \not\prec^{\mathrm{PM}}_{\mathfrak{D}} s$, for all states $s \triangleq (\mathsf{r}, \zeta, \_) \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$, since neither there exists a priority $q \in \mathsf{rng}(\mathsf{r})$ such that $\mathsf{r}^{-1}(q) \subset \mathsf{r}^{-1}(q)$ nor $p < p$ or $\zeta \neq \zeta$.

For the transitive property, instead, consider three states $s_1 \triangleq (\mathsf{r}_1, \zeta_1, p_1), s_2 \triangleq (\mathsf{r}_2, \zeta_2, p_2), s_3 \triangleq (\mathsf{r}_3, \zeta_3, p_3) \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$ such that $s_1 \prec^{\mathrm{PM}}_{\mathfrak{D}} s_2$ and $s_2 \prec^{\mathrm{PM}}_{\mathfrak{D}} s_3$. Due to Items 3.a and 3.b of the same definition, four cases arise, the first three of which are treated as in Lemma 2.2.1. Here, we just discuss the last one. Since Item 3.b holds for both $s_1 \prec^{\mathrm{PM}}_{\mathfrak{D}} s_2$ and $s_2 \prec^{\mathrm{PM}}_{\mathfrak{D}} s_3$, by Item 3.b.i, it follows that $\mathsf{r}_1 = \mathsf{r}_2 = \mathsf{r}_3$. Consequently, to show that $s_1 \prec^{\mathrm{PM}}_{\mathfrak{D}} s_3$, we have only to consider the mutual relations between the three flags ζ_1, ζ_2, and ζ_3 and the three priorities p_1, p_2, and p_3 required by Item 3.b.ii. Since we cannot have both $\zeta_1 \neq \zeta_2 = \mathtt{t}$ and $\zeta_2 \neq \zeta_3 = \mathtt{t}$, it holds that the only possibilities are $p_1 < p_2 < p_3$, $p_1 < p_2 = p_3$, or $p_1 = p_2 < p_3$. However, from all of them we derive that $p_1 < p_3$, which suffices to prove the thesis.

Finally, the fact that $\top^{\mathrm{PM}}_{\mathfrak{D}} \triangleq (\varnothing, \mathtt{f}, \mathsf{pr}(\mathfrak{D}))$ is the maximal element of $\mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$ *w.r.t.* $\prec^{\mathrm{PM}}_{\mathfrak{D}}$ follows by applying the same argument of Lemma 2.2.1. □

Lemma 2.2.5 (Query Function for PM)**.** *The function* $\Re^{\mathrm{PM}}_{\mathfrak{D}}$ *is an query function,* i.e.*, for all states* $s \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$*, it holds that* (1) $\Re^{\mathrm{PM}}_{\mathfrak{D}}(s) \in \mathrm{Rg}_{\mathfrak{D}}$ *and* (2) *if* $\Re^{\mathrm{PM}}_{\mathfrak{D}}(s) \in \mathrm{Rg}^{-}_{\mathfrak{D}}$ *then* $s \succ^{\mathrm{PM}}_{\mathfrak{D}} \Re^{\mathrm{PM}}_{\mathfrak{D}}(s)$.

Proof. The result immediately follows from the Query Function for PP Lemma 2.2.2 and the simple observation that, for all PM states $s \triangleq (\mathsf{r}, \zeta, p) \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$, it holds that: *(1)* $s^{\natural} \triangleq (\mathsf{r}^{(\geq p)}, \zeta, p) \in \mathrm{S}^{\mathrm{PP}}_{\mathfrak{D}}$, *(2)* $\Re^{\mathrm{PM}}_{\mathfrak{D}}(s) = \Re^{\mathrm{PP}}_{\mathfrak{D}}(s^{\natural})$, and *(3)* $s \succ^{\mathrm{PM}}_{\mathfrak{D}} (\mathrm{R}, \alpha)$ iff $s^{\natural} \succ^{\mathrm{PP}}_{\mathfrak{D}} (\mathrm{R}, \alpha)$, for all open region-pairs $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\mathfrak{D}}$. This is due to the fact that both the definitions of the compatibility relation $\succ^{\mathrm{PM}}_{\mathfrak{D}}$ and the function $\Re^{\mathrm{PM}}_{\mathfrak{D}}$ do not make use of the regions $\mathsf{r}^{-1}(q)$ of priorities q strictly less that p. □

Lemma 2.2.6 (Successor Function for PM)**.** *The function* $\downarrow^{\mathrm{PM}}_{\mathfrak{D}}$ *is an successor function,* i.e.*, for all states* $s \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$ *and region pairs* $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\mathfrak{D}}$ *with* $s \succ^{\mathrm{PM}}_{\mathfrak{D}} (\mathrm{R}, \alpha)$*, it holds that* (1) $s \downarrow^{\mathrm{PM}}_{\mathfrak{D}} (\mathrm{R}, \alpha) \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$ *and* (2) $s \downarrow^{\mathrm{PM}}_{\mathfrak{D}} (\mathrm{R}, \alpha) \prec^{\mathrm{PM}}_{\mathfrak{D}} s$.

Proof. Let $s \triangleq (\mathsf{r}, \zeta, p) \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{D}}$ be a state, $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\mathfrak{D}}$ an open region-pair in $\mathfrak{D}$ that is PM compatible with s, and $s^{\star} = (\mathsf{r}^{\star}, \zeta^{\star}, p^{\star}) \triangleq s \downarrow^{\mathrm{PM}}_{\mathfrak{D}} (\mathrm{R}, \alpha)$ the result obtained by computing the function $\downarrow^{\mathrm{PM}}_{\mathfrak{D}}$ on s and (R, α). Due to Item 1 of Definition 2.2.3, we have that *(1)* $\mathsf{r}^{(\geq p(\zeta))}$ is maximal, *(2)* if $\zeta = \mathtt{f}$ then $p = \mathsf{pr}(\mathfrak{D}_s)$ else $p \in \mathsf{rng}(\mathsf{r})$, *(3)* $\mathsf{r}^{(\geq p)}$ is saturated, and *(4)* r is memoization consistent *w.r.t. p*. Moreover, by Item 1 of Definition 2.2.2 with $\mathcal{S}^{\mathrm{PP}}_{\mathfrak{D}}$ replaced by $\mathcal{S}^{\mathrm{PM}}_{\mathfrak{D}}$, it holds that $\mathrm{R} \subseteq \mathrm{Ps}_{\mathfrak{D}_s}$, which implies *(5)* $\mathsf{dom}\big(\mathsf{r}^{(\geq p(\zeta))}\big) \cap \mathrm{R} = \emptyset$.

On the one hand, suppose that R is α-open in ∂_s, *i.e.*, $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\partial_s}$. By Lines 1-4 of Algorithm 7, we have that *(6)* $\mathsf{r}^\star = \mathsf{r}[\mathrm{R} \mapsto p]$, *(7)* $p^\star = \mathsf{max}\{\mathsf{pr}(\partial_s \backslash \mathrm{R}), \mathsf{max}(\mathsf{rng}(\mathsf{r}^{(<p)}))\}$, and *(8)* $\zeta^\star = \mathtt{t}$, if $p^\star < p$ and $p^\star \in \mathsf{rng}(\mathsf{r}^\star)$, and $\zeta^\star = \mathtt{f}$, otherwise. Also, by Item 2 of Definition 2.2.2 with $\mathcal{S}^{\mathrm{PP}}_{\partial}$ replaced by $\mathcal{S}^{\mathrm{PM}}_{\partial}$, it holds that *(9)* R is α-maximal in ∂_s, *(10)* if $\zeta = \mathtt{t}$ then $\mathsf{r}^{-1}(p) \subseteq \mathrm{R}$, and *(11)* $\alpha \equiv_2 p = \mathsf{bep}^{\overline{\alpha}}_{\partial_s}(\mathrm{R})$. First note that, by Point (6), it holds that $\mathsf{r}^\star = \mathsf{r}^{(>p)}[\mathrm{R}^\star \mapsto p] \cup (\mathsf{r}^{(<p)} \setminus \mathrm{R})$ with $\mathrm{R}^\star \triangleq \mathrm{R} \cup \mathsf{r}^{-1}(p)$. However, by Point (4), we can reduce the previous expression to *(12)* $\mathsf{r}^\star = \mathsf{r}^{(>p)}[\mathrm{R}^\star \mapsto p] \cup (\mathsf{r}^{(<p)\wedge(\equiv_2 p)} \setminus \mathrm{R}) \cup \mathsf{r}^{(<p)\wedge(\not\equiv_2 p)}$, since $\mathsf{dom}\big(\mathsf{r}^{(<p)\wedge(\not\equiv_2 p)}\big) \cap \mathrm{R} = \emptyset$. It is easy to see that *(13)* $\mathrm{R}^\star$ is a saturated α-maximal α-region in $\partial^{\leq p}_{\mathsf{r}}$ with $\mathsf{bep}^{\overline{\alpha}}_{\partial^{\leq p}_{r}}(\mathrm{R}^\star) = p$. Indeed, if $\zeta = \mathtt{t}$ or $p \notin \mathsf{rng}(\mathsf{r})$ then $\mathrm{R}^\star = \mathrm{R}$, so, its maximality is due to Point (9) and the value of best escape priority to Point (11). Moreover, the saturation follows from Points (2) and (11), if $p \notin \mathsf{rng}(\mathsf{r})$, and from Points (3) and (10), if $\zeta = \mathtt{t}$. In case $\zeta = \mathtt{f}$ and $p \in \mathsf{rng}(\mathsf{r})$, instead, the required properties follow from Proposition 2.2.1, whose hypothesis are derived by Points (1), (3), (9), and (11). Now, it is obvious that $\mathsf{max}(\mathsf{rng}(\mathsf{r}^{(<p)})) < p$. Furthermore, $\partial_s \setminus \mathrm{R}$ is a subgame of ∂_s, so, $\mathsf{pr}(\partial_s \backslash \mathrm{R}) \leq \mathsf{pr}(\partial_s) = p$, due to the monotonicity of the priority function on games. Hence, by Points (2) and (7), we have that $p^\star = \mathsf{max}\{\mathsf{pr}(\partial_s \backslash \mathrm{R}), \mathsf{max}(\mathsf{rng}(\mathsf{r}^{(<p)}))\} \leq p$ and, so, there are no priorities $q \in \mathsf{rng}(\mathsf{r})$ such that $p^\star < q < p$. Also, by Point (8), we have that $\mathsf{r}^{\star(\geq p^\star(\zeta^\star))} = \mathsf{r}^{\star(\geq p)}$. Consequently, the following hold: $\mathsf{r}^\star$ is a region function, by Point (12) and Proposition 2.2.4; $\mathsf{r}^{\star(\geq p^\star(\zeta^\star))}$ is maximal, by Points (1) and (13); if $\zeta^\star = \mathtt{t}$ then $p^\star \in \mathsf{rng}(\mathsf{r}^\star)$, by Point (8), else $p^\star = \mathsf{pr}(\partial_{s^\star})$, since $\partial_{s^\star} = \partial^{<p^\star(\zeta^\star)}_{\mathsf{r}^\star} = \partial^{<p}_{\mathsf{r}^\star} = \partial^{<p(\zeta)}_{\mathsf{r}} \setminus \mathrm{R} = \partial_s \setminus \mathrm{R}$; $\mathsf{r}^{\star(\geq p^\star)}$ is saturated, by Points (3) and (13), since $\mathsf{r}^{\star(\geq p^\star)} = \mathsf{r}^{\star(\geq p)}[\mathsf{r}^{-1}(p^\star) \mapsto p^\star]$ and $\mathsf{r}^{-1}(p^\star)$ is saturated due to the choice of the value $p^\star$; $\mathsf{r}^\star$ is memoization consistent by Point (4). Summing up, we derive that $s^\star \in \mathrm{S}^{\mathrm{PM}}_{\partial}$. At this point, it remains just to show that $(\mathsf{r}^\star, \zeta^\star, p^\star) \prec^{\mathrm{PM}}_{\partial} (\mathsf{r}, \zeta, p)$. If $\mathsf{r}^{-1}(p) \subset \mathrm{R}^\star$, the thesis follows from Item 3.a of Definition 2.2.3, where the priority q is set to p. On the contrary, because of Point (5), if $\mathrm{R}^\star = \mathsf{r}^{-1}(p)$, we necessarily have $\zeta = \mathtt{t}$. In addition, $\mathsf{dom}\big(\mathsf{r}^{(<p)\wedge(\equiv_2 p)}\big) \cap \mathrm{R} = \emptyset$, so, $\mathsf{r}^\star = \mathsf{r}$. Also, by Point (8), either $p^\star < p$ or $p^\star = p$ and $\zeta^\star = \mathtt{f}$. Therefore, the thesis is derived from Item 3.b of the same definition.

On the other hand, suppose that R is α-closed in ∂_s, *i.e.*, $(\mathrm{R}, \alpha) \notin \mathrm{Rg}^{-}_{\partial_s}$. By Lines 1, 5-7 of Algorithm 7, we have that *(14)* $p^\star = \mathsf{brp}^{\overline{\alpha}}_{\partial}\big(\mathrm{R}, \mathsf{r}^{(\geq p(\zeta))}\big)$, *(15)* $\mathsf{r}^\star = \mathsf{r}^{(\geq p^\star)\vee(\equiv_2 p^\star)}[\mathrm{R} \mapsto p^\star]$, and *(16)* $\zeta^\star = \mathtt{t}$. By the last two points, it is immediate to see that Items 3.b of Definition 2.2.3 hold. Moreover, due to the first point, we have that $p(\zeta) \leq p^\star$. Consequently, by Points (1), (15), and (16), it follows that $\mathsf{r}^{\star(\geq p^\star(\zeta^\star))}$ is maximal, since $p^\star < p^\star(\zeta^\star)$. Thus, Item 3.a of the same definition holds as well. Finally, to prove that $s^\star \in \mathrm{S}^{\mathrm{PM}}_{\partial}$, we only need to show that $\mathsf{r}^\star$ is a region function, which also satisfies Item 3.c, *i.e.*, $\mathsf{r}^{\star(\geq p^\star)}$ is saturated. To do this, first observe that $\mathsf{r}^{\star(\geq p^\star)} = \mathsf{r}^{(>p^\star)}[\mathrm{R}^\star \mapsto p^\star]$ with $\mathrm{R}^\star \triangleq \mathrm{R} \cup \mathsf{r}^{-1}(p^\star)$. Now, by Proposition 2.2.1, we have $\mathrm{R}^\star$ is a saturated α-region in $\partial^{\leq p^\star}_{\mathsf{r}}$ with $\mathsf{bep}^{\overline{\alpha}}_{\partial^{\leq p^\star}_{r}}(\mathrm{R}^\star) \in \{p^\star, \infty\}$. Therefore, since $\mathsf{r}^{(>p^\star)}$ is a saturated region function, the thesis immediately follows. At this point, as for

the previous case, it remains to show that $(\mathsf{r}^\star, \zeta^\star, p^\star) \prec_\partial^{\mathrm{PM}} (\mathsf{r}, \zeta, p)$. This fact easily follows from Item 3.a of Definition 2.2.3, where the priority q is set to $p^\star$, since, by Point (5), we have that $\mathsf{r}^{-1}(p) \subset \mathrm{R}^\star$. □

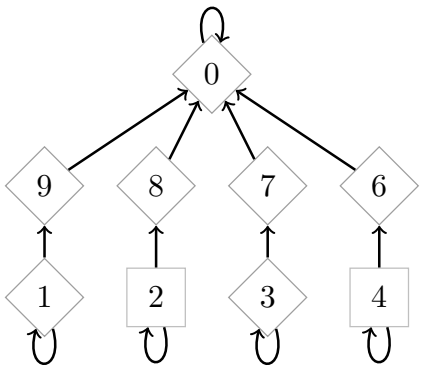


Figure 2.4: The ∂_4^{PM} game.

To get an upper bound on the length of the chains in the this region space, hence an upper bound on the time complexity of PM, it suffices to observe the following. By forgetting the information about regions with priorities lower than the current priority of PM-states, we obtain a homomorphism $\mathsf{h} : (\mathsf{r}, \zeta, p) \in \mathrm{S}_\partial^{\mathrm{PM}} \mapsto (\mathsf{r}^{(\geq p)}, \zeta, p) \in \mathrm{S}_\partial^{\mathrm{PP}}$, mapping each state of the region space for PM to a state in the space for PP. It is not hard to verify that such a mapping does indeed preserve the ordering relation. We can then state the following theorem.

Theorem 2.2.5 (PM Size & Depth Upper Bounds)**.** *The size of a* PM *region space* $\mathcal{R}_\partial^{\mathrm{PM}}$ *for a game* $\partial \in \mathrm{PG}$ *with* $n \in \mathbb{N}_+$ *positions and* $k \in [1, n]$ *priorities is bounded by* $2k^{n+1}$*. Moreover, its depth is not greater then that of the* PP *region space* $\mathcal{R}_\partial^{\mathrm{PP}}$ *for the same game.*

The next theorem gives a precise measure of the running time of the PM algorithm on the worst-case game ∂_h^{PM}. The proof can easily be obtained by observing that the execution depth of the associated region space $\mathcal{R}_h^{\mathrm{PM}}$ satisfies the recurrence relation $\mathsf{d}(h+2) = \mathsf{d}(h+1) + \mathsf{d}(h) + h + 5$ with $\mathsf{d}(0) = 1$ and $\mathsf{d}(1) = 4$.

Theorem 2.2.6 (PM Execution-Depth Lower Bound)**.** *For all numbers* $h \in \mathbb{N}$*, there exists a* PM *region space* $\mathcal{R}_h^{\mathrm{PM}}$ *with* $k = 2h + 1$ *positions and priorities, whose execution depth is* $\mathsf{Fib}(2(h+4))/\mathsf{Fib}(h+4) - (h+6) = \Theta\big(((1+\sqrt{5})/2)^{k/2}\big)$.

Here, we want to briefly discuss the exponential behaviour of the described approach on a family of games similar to the one provided for the previous algorithm. First observe that, by applying the PM mechanism to the PP lower-bound games $\mathfrak{D}_h^{\mathrm{PP}}$ and $\mathfrak{D}_{m,h}^{\mathrm{PP}}$, we are able to find a winning region in a number of steps that is only quadratic in h and linear in m. If, instead, we alternate both the parity of the priorities and the owner of the positions in the body of the chains, as exemplified by the game $\mathfrak{D}_4^{\mathrm{PM}}$ depicted in Figure 2.4, we obtain a new game simulating a different kind of binary counter with h digits. This time, however, the base of the exponential function describing the execution depth of the associated region space $\mathcal{R}_h^{\mathrm{PM}}$ is not a natural number, but the golden ratio $(1+\sqrt{5})/2$. This is due to the fact that, when the i-th digit is incremented, only the j-th ones, with $j \in]i, h]$ and $j \not\equiv_2 i$, gets reset. Also in this case we conjecture that a matching upper bound exists.

2.2.3 Delayed Promotions

It should be clear by now that the exponential behaviour exhibited by the two solutions proposed above is strictly tied to the revocation mechanism applied to regions with opposite priority *w.r.t.* the promotions performed. Therefore, to further reduce such revocations, we need a more clever promotion policy. We propose here a third procedure based on delaying promotions that require revocations, called *blocked promotions*, until a complete knowledge of the current search phase is reached. Then, we choose the highest priority $p^\star$ of the promotion available and perform all the postponed ones of the same parity as $p^\star$ altogether. The target priorities of promotions actually performed during the search, called *instant promotions*, are recorded in a supplementary set P. To keep track of the *delayed promotions*, *i.e.*, the postponed ones, a supplementary priority function $\widetilde{\mathsf{r}}$ is used. The new procedure evolves as the PM algorithm, as long as open regions are discovered. Two cases may arise, instead, when a region R, closed in the current subgame, is provided by the query function. If the corresponding promotion is not blocked, the destination priority is recorded in the set P and the instant promotion is performed similarly to the case of PM. Otherwise, the promotion it is not performed. Instead, it is recorded in the supplementary function $\widetilde{\mathsf{r}}$, by assigning to R in $\widetilde{\mathsf{r}}$ the priority to which it should be promoted and in r the current priority reached in the search. Then, the positions in R are removed from the subgame and the search proceeds at the highest remaining priority. In case the region function r together with R covers the entire game, all priorities available in the game have been processed and the empty subgame is reached. At this point, the delayed promotion to the highest priority $p^\star$ recorded in $\widetilde{\mathsf{r}}$ is selected and all promotions of the same parity are applied at once. This is done by first moving all regions from $\widetilde{\mathsf{r}}$ into r and then removing from r the regions of opposite parity *w.r.t.* $p^\star$, exactly as done by PM. The search, then, resumes at priority

$p^\star$. A promotion is considered as blocked if its target priority is either *(a)* greater than some priority in P of opposite parity, which would be otherwise revoked, or *(b)* lower than the target of some previously delayed promotion recorded in $\widetilde{\mathsf{r}}$, but greater than the corresponding priority set in r.

	1	2	3
6	$\mathtt{a}\downarrow$	$\cdots$	$\mathtt{a},\mathtt{b},\mathtt{g},\mathtt{h}\downarrow$
5	$\mathtt{b},\mathtt{e},\mathtt{f}\downarrow$	$\cdots$	
4	$\mathtt{c}\downarrow$	$\mathtt{c},\mathtt{d}\downarrow$	$\mathtt{c},\mathtt{d}\uparrow_6$
2	$\mathtt{d}\uparrow_4$		
1		$\mathtt{g}\not\uparrow_5$	
0		$\mathtt{h}\uparrow_6$	

Table 2.3: DP simulation.

This idea is illustrated by Table 2.3 on the example of Figure 2.2. The computation proceeds as for PM, until the promotion to priority 1, shown in Column 2, occurs. This is an instant promotion to 3, since the only other promotion already computed and recorded in P has value 4. Hence, it can be performed and saved in P as well. Starting from priority 3 in Column 3, the closed 1-region $\{\mathtt{d},\mathtt{g}\}$ could be promoted to 5. However, since its target is greater than $4 \in \mathrm{P}$, it is delayed and recorded in $\widetilde{\mathsf{r}}$, where it is assigned priority 5. At priority 0, a delayed promotion of 0-region $\{\mathtt{i}\}$ to priority 6 is encountered and registered, since it would overtake priority $3 \in \mathrm{P}$. Now, the empty subgame is reached. Since the highest available promotion is the one to priority 6 and no other promotion of the same parity was delayed, 0-region $\{\mathtt{i}\}$ is promoted and its target 6 recorded in P. Previously computed 0-region $\{\mathtt{c},\mathtt{e}\}$ has the same parity and, therefore, it is not revoked, while both 1-regions $\{\mathtt{b},\mathtt{f},\mathtt{h}\}$ and $\{\mathtt{d},\mathtt{g}\}$ are removed from r and $\widetilde{\mathsf{r}}$. After maximization of the newly created 0-region $\{\mathtt{a},\mathtt{i}\}$, positions $\mathtt{b}$, $\mathtt{d}$, and $\mathtt{g}$ get attracted as well. This leads to the first cell of Column 4, where 0-region $\{\mathtt{a},\mathtt{b},\mathtt{d},\mathtt{g},\mathtt{i}\}$ is open. The next priority to process is then 4, where the previous 0-region $\{\mathtt{c},\mathtt{e}\}$ is now closed in the corresponding subgame and gets promoted to 6. Finally, the region obtained after the promotion is a winning region for player 0. Note that postponing the promotion of 1-region $\{\mathtt{d},\mathtt{g}\}$ allowed a substantial reduction in the number of operations. Indeed, the redundant extension of 1-region $\{\mathtt{b},\mathtt{f},\mathtt{h}\}$ was avoided.

The delayed promotion mechanism is essentially a refinement of PM. Therefore, they share all the requirements on the corresponding components of a state, on the ordering and compatibility relations. However, it introduces in the state two supplementary elements: a priority function $\widetilde{\mathsf{r}}$, which collects the delayed promotions not performed within r, and a set P, which collects the priorities of the instant

promotions actually performed. Hence, in order to formally define the corresponding region space, we need to provide suitable constraints for these two elements and the relations with the other components of a state.

The first requirement on $\widetilde{\mathsf{r}}$ is the property that we called *alignment* with r. This property captures the intuition that $\widetilde{\mathsf{r}}$ can only contain regions resulting from promotions that are blocked in r. Hence, every priority in $\widetilde{\mathsf{r}}$ must also be in r. Moreover, every region in $\widetilde{\mathsf{r}}$ at some priority q must contain positions already present in r at some strictly lower priorities of the same parity. This intuition is formalized as follows. We say that a priority function $\widetilde{\mathsf{r}}$ is *aligned* with a region function r *w.r.t.* p if, for all priorities $q \in \mathsf{rng}(\widetilde{\mathsf{r}})$, it holds that *(a)* $p < q$, *(b)* $q \in \mathsf{rng}(\mathsf{r})$, and *(c)* $\widetilde{\mathsf{r}}^{-1}(q) \subseteq \mathsf{dom}\big(\mathsf{r}^{(\geq p)\wedge(<q)\wedge(\equiv_2 q)}\big)$.

When a delayed promotion is applied, all the regions of a given parity contained in the supplementary function $\widetilde{\mathsf{r}}$ get transferred into r. This merging operations is formally defined as $\mathsf{r}_1 \uplus \mathsf{r}_2 \triangleq (\mathsf{r}_1 \setminus \mathsf{dom}(\mathsf{r}_2)) \cup \mathsf{r}_2$. Hence, a state for DP is a tuple $((\mathsf{r},\zeta,p),\widetilde{\mathsf{r}},\mathrm{P})$, where (r,ζ,p) is a PM state. In addition, $\widetilde{\mathsf{r}}$ must be such that, when delayed promotions are eventually performed, the priority function resulting from the merging operation is indeed a region function. This property is ensured by requiring that $\mathsf{r} \uplus \widetilde{\mathsf{r}}$ is a region function. A second constraint pertain the alignment between $\widetilde{\mathsf{r}}$ and r up to the current priority p of the state. The last additional requirement is that the set P only contains priorities in r, which are smaller than p whenever they belong to the opponent player *w.r.t.* the one associated with p. The state space for DP is, therefore, defined as follows.

Definition 2.2.4 (State Space for DP)**.** *A* DP state space *is a tuple* $\mathcal{S}^{\mathrm{DP}}_{\mathfrak{I}} \triangleq \langle \mathrm{S}^{\mathrm{DP}}_{\mathfrak{I}}, \top^{\mathrm{DP}}_{\mathfrak{I}}, \prec^{\mathrm{DP}}_{\mathfrak{I}} \rangle$*, where its components are defined as prescribed in the following:*

1. $\mathrm{S}^{\mathrm{DP}}_{\mathfrak{I}} \subseteq \mathrm{S}^{\mathrm{PM}}_{\mathfrak{I}} \times \Delta_{\mathfrak{I}} \times 2^{\mathrm{Pr}_{\mathfrak{I}}}$ *is the set of all triples* $s \triangleq ((\mathsf{r},\zeta,p),\widetilde{\mathsf{r}},\mathrm{P})$*, called* states*, composed by a* PM *state* $(\mathsf{r},\zeta,p) \in \mathrm{S}^{\mathrm{PM}}_{\mathfrak{I}}$*, a priority function* $\widetilde{\mathsf{r}} \in \Delta_{\mathfrak{I}}$*, and a set of priorities* $\mathrm{P} \subseteq \mathrm{Pr}_{\mathfrak{I}}$ *such that* (a) $\mathsf{r} \uplus \widetilde{\mathsf{r}}$ *is a region function,* (b) $\widetilde{\mathsf{r}}$ *is aligned with* r w.r.t. p*,* (c) $\mathrm{P} \subseteq \mathsf{rng}(\mathsf{r})$*, and* (d)*, for all priorities* $q \in \mathrm{P}$*, if* $p \not\equiv_2 q$ *then* $p < q$*;*

2. $\top^{\mathrm{DP}}_{\mathfrak{I}} \triangleq (\top^{\mathrm{PM}}_{\mathfrak{I}}, \varnothing, \emptyset)$*;*

3. *for any two states* $s_1 \triangleq ((\mathsf{r}_1,\zeta_1,p_1),\widetilde{\mathsf{r}}_1,\mathrm{P}_1), s_2 \triangleq ((\mathsf{r}_2,\zeta_2,p_2),\widetilde{\mathsf{r}}_2,\mathrm{P}_2) \in \mathrm{S}^{\mathrm{DP}}_{\mathfrak{I}}$*, it holds that* $s_1 \prec^{\mathrm{DP}}_{\mathfrak{I}} s_2$ *iff* (a) $(\mathsf{r}_1,\zeta_1,p_1) \prec^{\mathrm{PM}}_{\mathfrak{I}} (\mathsf{r}_2,\zeta_2,p_2)$ *and* (b) *if* $\mathsf{r}_1 = \mathsf{r}_2$ *then* (b.i) $\widetilde{\mathsf{r}}_2 \subseteq \widetilde{\mathsf{r}}_1$ *and* (b.ii) $\mathrm{P}_1 = \mathrm{P}_2$ *else* (b.iii) $\widetilde{\mathsf{r}}_2^{(>q)} \subseteq \widetilde{\mathsf{r}}_1^{(>q)}$ *and* (b.iv)*, for all priorities* $l \in \mathrm{P}_2$ *with* $q < l$*, it holds that* $l \in \mathrm{P}_1$*, where* $q \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}_1) : {\mathsf{r}_1}^{(\geq m)} \neq {\mathsf{r}_2}^{(\geq m)}\}$.

Note that the additional requirements in the definition of the ordering relation further refine the one for PM, by taking into account the additional elements contained in the states.

Finally, in order to properly define the compatibility relation connecting the query and successor functions for DP, we first need to formally characterize blocked promotions *w.r.t.* a state. A promotion is blocked if its target priority is either *(a)* greater than some priority in P of opposite parity or *(b)* is lower than the target of some previously delayed promotion recorded in $\widetilde{\mathsf{r}}$, but greater than the corresponding priority set in r. These two conditions are expressed by the following two formulas, respectively, where q is the target priority of the blocked promotion.

$$\phi_{\mathrm{P}}(q, \mathrm{P}) \triangleq \exists l \in \mathrm{P}.\, l \not\equiv_2 q \wedge l < q$$
$$\phi_{\widetilde{\mathsf{r}}}(q, \mathsf{r}, \widetilde{\mathsf{r}}) \triangleq \exists v \in \mathsf{dom}(\widetilde{\mathsf{r}})\,.\,\mathsf{r}(v) < q \leq \widetilde{\mathsf{r}}(v)$$

Then, an α-region R is called *α-locked w.r.t.* a state s if the predicate $\phi_{Blk}(q, s) \triangleq \phi_{\widetilde{\mathsf{r}}}(q, \mathsf{r}, \widetilde{\mathsf{r}}) \vee \phi_{\mathrm{P}}(q, \mathrm{P})$ is satisfied, where $q = \mathsf{brp}^{\overline{\alpha}}_{\supset}(\mathrm{R}, (\mathsf{r} \uplus \widetilde{\mathsf{r}}))$.

The compatibility relation requires, in addition to the compatibility constraints for PM, that any α-locked region, possibly returned by the query function, be maximal and contained in the region $\mathsf{r}^{-1}(p)$ associated to the priority p of the current state.

Definition 2.2.5 (Compatibility Relation for DP)**.** *An open region-pair* $(\mathrm{R}, \alpha) \in \mathrm{Rg}^{-}_{\supset}$ *is* DP compatible *with a state* $s \triangleq ((\mathsf{r}, \zeta, p), \_, \_) \in \mathrm{S}^{\mathrm{DP}}_{\supset}$*, in symbols* $s \succ^{\mathrm{DP}}_{\supset} (\mathrm{R}, \alpha)$*, iff*

(1) $(\mathsf{r}, \zeta, p) \succ^{\mathrm{PM}}_{\supset} (\mathrm{R}, \alpha)$ *and*

(2) *if* R *is α-locked* w.r.t. s *then*

(2.a) R *is α-maximal in* $\supset_s$ *and*

(2.b) *if* $\zeta = \mathtt{t}$ *then* $\mathsf{r}^{-1}(p) \subseteq \mathrm{R}$.

Algorithm 8 implements the successor function for DP. The pseudo-code on the right-hand side consists of three macros, namely *#Assignment*, *#InstantPromotion*, and *#DelayedPromotion*, used by the algorithm. Macro *#Assignment*(ξ) performs the insertion of a new region R into the region function r. In presence of a blocked promotion, *i.e.*, when the parameter ξ is set to $\mathtt{f}$, the region is also recorded in $\widetilde{\mathsf{r}}$ at the target priority of the promotion. Macro *#InstantPromotion* corresponds to the DP version of the standard promotion operation of PM. The only difference is that it must also take care of updating the supplementary elements $\widetilde{\mathsf{r}}$ and P. Macro *#DelayedPromotion*, instead, is responsible for the delayed promotion operation specific to DP.

If the current region R is open, the main algorithm proceeds, as in the other solvers, at assigning to it the current priority p in r. This is done by calling macro *#Assignment* with parameter $\mathtt{t}$. Otherwise,

the region is closed and a promotion should be performed at priority q, the brp of that region. In this case, the algorithm first checks whether such promotion is blocked at Line 8. If it is not, then the promotion is performed as in PM, by executing $\#InstantPromotion$, and the target is kept track of in the set P. If, instead, the promotion to q is blocked, but some portion of the game still has to be processed, the region is assigned to priority p in r and the promotion to q is delayed and stored in $\widetilde{\mathsf{r}}$. This is done by executing $\#Assignment$ with parameter $\mathtt{f}$. Finally, in case the whole game has been processed, the delayed promotion to the highest priority recorded in $\widetilde{\mathsf{r}}$ is selected and applied. Macro $\#DelayedPromotion$ is executed, thus merging r with $\widetilde{\mathsf{r}}$. Function $\widetilde{\mathsf{r}}$ is, then, deleted and P is updated accordingly, by forgetting all the instant promotions of the opponent player previously performed.

Algorithm 8: Successor Function.

signature $\downarrow_{\mathcal{D}} : \succ_{\mathcal{D}} \to (\Delta_{\mathcal{D}} \times \{\mathtt{t},\mathtt{f}\} \times \mathrm{Pr}_{\mathcal{D}}) \times \Delta_{\mathcal{D}} \times 2^{\mathrm{Pr}}$

function $s\downarrow_{\mathcal{D}}(\mathrm{R},\alpha)$

let $((\mathsf{r},\_,p),\widetilde{\mathsf{r}},\mathrm{P}) = s$ **in**

1 **if** $(\mathrm{R},\alpha) \in \mathrm{Rg}^{-}_{\mathcal{D}_s}$ **then**

2-6 $\#Assignment(\mathtt{t})$

else

7 $q \leftarrow \mathsf{brp}^{\overline{\alpha}}_{\mathcal{D}}(\mathrm{R}, \mathsf{r} \uplus \widetilde{\mathsf{r}})$

8 **if** $\phi_{Blk}(q,s)$ **then**

9 $\widehat{\mathsf{r}} \leftarrow \widetilde{\mathsf{r}}[\mathrm{R} \mapsto q]$

10-15 **if** $\mathsf{dom}(\mathsf{r}) \cup \mathrm{R} \subset \mathrm{Ps}_{\mathcal{D}}$ **then** $\#Assignment(\mathtt{f})$

16-20 **else** $\#DelayedPromotion$

else

21-25 $\#InstantPromotion$

26 **return** $((\mathsf{r}^\star,\zeta^\star,p^\star),\widetilde{\mathsf{r}}^\star,\mathrm{P}^\star)$

Algorithm 9: Assignment Macro.

macro $\#Assignment(\xi)$

1 $\mathsf{r}^\star \leftarrow \mathsf{r}[\mathrm{R} \mapsto p]$

2 $p^\star \leftarrow \mathsf{max}\{\mathsf{pr}(\mathcal{D}_s \backslash \mathrm{R}), \mathsf{max}(\mathsf{rng}(\mathsf{r}^{(<p)}))\}$

3 $\zeta^\star \leftarrow p^\star < p \wedge p^\star \in \mathsf{rng}(\mathsf{r}^\star)$

4 $\widetilde{\mathsf{r}}^\star \leftarrow$ **#if** ξ **#then** $\widetilde{\mathsf{r}}$ **#else** $\widehat{\mathsf{r}}$

5 $\mathrm{P}^\star \leftarrow \mathrm{P}$

Algorithm 10: Delayed Promotion Macro.

macro *#DelayedPromotion*

$p^\star \leftarrow \mathsf{max}(\mathsf{rng}(\widehat{\mathsf{r}}))$

$\mathsf{r}^\star \leftarrow (\mathsf{r} \uplus \widehat{\mathsf{r}})^{(\geq p^\star)\vee(\equiv_2 p^\star)}$

$\zeta^\star \leftarrow \mathtt{t}$

$\widetilde{\mathsf{r}}^\star \leftarrow \emptyset$

$\mathrm{P}^\star \leftarrow \mathrm{P} \cap \mathsf{rng}(\mathsf{r}^\star) \cup \mathsf{rng}\big(\widehat{\mathsf{r}}^{(\equiv_2 p^\star)}\big)$

Algorithm 11: Instant Promotion Macro.

macro *#InstantPromotion*

$p^\star \leftarrow q$

$\mathsf{r}^\star \leftarrow \mathsf{r}^{(\geq p^\star)\vee(\equiv_2 p^\star)}[\mathrm{R} \mapsto p^\star]$

$\zeta^\star \leftarrow \mathtt{t}$

$\widetilde{\mathsf{r}}^\star \leftarrow \widetilde{\mathsf{r}}^{(> p^\star)}$

$\mathrm{P}^\star \leftarrow \mathrm{P} \cap \mathsf{rng}(\mathsf{r}^\star) \cup \{p^\star\}$

The following theorem establishes the soundness of the resulting solution procedure.

Theorem 2.2.7 (DP Region Space)**.** *Given a game* $\mathcal{D}$*, the* DP structure $\mathcal{R}_{\mathcal{D}}^{\mathrm{DP}} \triangleq \langle \mathcal{D}, \mathcal{S}_{\mathcal{D}}^{\mathrm{DP}}, \succ_{\mathcal{D}}^{\mathrm{DP}}, \Re_{\mathcal{D}}^{\mathrm{DP}}, \downarrow_{\mathcal{D}}^{\mathrm{DP}} \rangle$*, where* $\mathcal{S}_{\mathcal{D}}^{\mathrm{DP}}$ *is the structure of Definition 2.2.4,* $\succ_{\mathcal{D}}^{\mathrm{DP}}$ *is the relation of Definition 2.2.5, and* $\Re_{\mathcal{D}}^{\mathrm{DP}}$ *and* $\downarrow_{\mathcal{D}}^{\mathrm{DP}}$ *are the functions computed by the Algorithms 5 and 8 with* $\mathcal{S}_{\mathcal{D}} \triangleq \mathcal{S}_{\mathcal{D}}^{\mathrm{DP}}$ *and* $\succ_{\mathcal{D}} \triangleq \succ_{\mathcal{D}}^{\mathrm{DP}}$*, where in the former the assumption "***let** $(\mathsf{r}, \zeta, p) = s$*" is replaced by "***let** $((\mathsf{r}, \zeta, p), \_, \_) = s$*", is a region space.*

As done in the previous two subsections, the proof of Theorem 2.2.7 regarding the structure $\mathcal{R}_{\mathcal{D}}^{\mathrm{DP}}$ is split into the Lemmas 2.2.7, 2.2.8, and 2.2.9, each proving one of the properties required by Definition 2.1.3 on the three components $\mathcal{S}_{\mathcal{D}}^{\mathrm{DP}}$, $\Re_{\mathcal{D}}^{\mathrm{DP}}$, and $\downarrow_{\mathcal{D}}^{\mathrm{DP}}$

Lemma 2.2.7 (State Space for DP)**.** *The* DP *state space* $\mathcal{S}_{\mathcal{D}}^{\mathrm{DP}} = \langle \mathrm{S}_{\mathcal{D}}^{\mathrm{DP}}, \top_{\mathcal{D}}^{\mathrm{DP}}, \prec_{\mathcal{D}}^{\mathrm{DP}} \rangle$ *for a game* $\mathcal{D} \in \mathrm{PG}$ *is a well-founded partial order* w.r.t. $\prec_{\mathcal{D}}^{\mathrm{DP}}$ *with maximal element* $\top_{\mathcal{D}}^{\mathrm{DP}}$*.*

Proof. Because of the finiteness of $\mathrm{S}_{\mathcal{D}}^{\mathrm{PM}}$, to show that $\prec_{\mathcal{D}}^{\mathrm{DP}}$ is a well-founded partial order on $\mathrm{S}_{\mathcal{D}}^{\mathrm{DP}}$, it is enough to prove that it is an irreflexive and transitive relation.

Due to Item 3.a of Definition 2.2.4, $s_1 \prec_{\mathcal{D}}^{\mathrm{DP}} s_2$ implies $(\mathsf{r}_1, \zeta_1, p_1) \prec_{\mathcal{D}}^{\mathrm{PM}} (\mathsf{r}_2, \zeta_2, p_2)$, for every pair of states $s_1 \triangleq ((\mathsf{r}_1, \zeta_1, p_1), \widetilde{\mathsf{r}}_1, \mathrm{P}_1), s_2 \triangleq ((\mathsf{r}_2, \zeta_2, p_2), \widetilde{\mathsf{r}}_2, \mathrm{P}_2) \in \mathrm{S}_{\mathcal{D}}^{\mathrm{DP}}$. Consequently, the irreflexive property of $\prec_{\mathcal{D}}^{\mathrm{DP}}$ immediately follows from the fact that the same holds for $\prec_{\mathcal{D}}^{\mathrm{PM}}$, the latter being a well-founded partial order, as proved in Lemma 2.2.4.

As far as the transitivity property is concerned, let $s_1 \triangleq ((\mathsf{r}_1, \zeta_1, p_1), \widetilde{\mathsf{r}}_1, \mathrm{P}_1), s_2 \triangleq ((\mathsf{r}_2, \zeta_2, p_2), \widetilde{\mathsf{r}}_2, \mathrm{P}_2)$, $s_3 \triangleq ((\mathsf{r}_3, \zeta_3, p_3), \widetilde{\mathsf{r}}_3, \mathrm{P}_3) \in \mathrm{S}_{\mathcal{D}}^{\mathrm{DP}}$ be three states such that $s_1 \prec_{\mathcal{D}}^{\mathrm{DP}} s_2$ and $s_2 \prec_{\mathcal{D}}^{\mathrm{DP}} s_3$. By applying again the previous observation, we have that $(\mathsf{r}_1, \zeta_1, p_1) \prec_{\mathcal{D}}^{\mathrm{PM}} (\mathsf{r}_2, \zeta_2, p_2)$ and $(\mathsf{r}_2, \zeta_2, p_2) \prec_{\mathcal{D}}^{\mathrm{PM}} (\mathsf{r}_3, \zeta_3, p_3)$. Thus, $(\mathsf{r}_1, \zeta_1, p_1) \prec_{\mathcal{D}}^{\mathrm{PM}} (\mathsf{r}_3, \zeta_3, p_3)$ holds. Therefore, to show that $s_1 \prec_{\mathcal{D}}^{\mathrm{DP}} s_3$, it remains only to prove that Item 3.b of the same definition holds as well. Due to the equality conditions on the three region functions r_1, r_2, and r_3, fours cases arise.

- If $\mathsf{r}_1 = \mathsf{r}_2 = \mathsf{r}_3$ then $\widetilde{\mathsf{r}}_3 \subseteq \widetilde{\mathsf{r}}_2 \subseteq \widetilde{\mathsf{r}}_1$ and $\mathrm{P}_1 = \mathrm{P}_2 = \mathrm{P}_3$. Thus, $s_1 \prec_{\mathcal{D}}^{\mathrm{DP}} s_3$.

- If $\mathsf{r}_1 = \mathsf{r}_2 \neq \mathsf{r}_3$ then $\widetilde{\mathsf{r}}_3^{(>q)} \subseteq \widetilde{\mathsf{r}}_2^{(>q)} = \widetilde{\mathsf{r}}_1^{(>q)}$ and, for all priorities $l \in \mathrm{P}_3$ with $q < l$, it holds that $l \in \mathrm{P}_2 = \mathrm{P}_1$, where $q \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}_2) : {\mathsf{r}_2}^{(\geq m)} \neq {\mathsf{r}_3}^{(\geq m)}\} = \{m \in \mathsf{rng}(\mathsf{r}_1) : {\mathsf{r}_1}^{(\geq m)} \neq {\mathsf{r}_3}^{(\geq m)}\}$. Thus, $s_1 \prec_{\mathfrak{D}}^{\mathrm{DP}} s_3$.

- If $\mathsf{r}_1 \neq \mathsf{r}_2 = \mathsf{r}_3$ then $\widetilde{\mathsf{r}}_3^{(>q)} = \widetilde{\mathsf{r}}_2^{(>q)} \subseteq \widetilde{\mathsf{r}}_1^{(>q)}$ and, for all priorities $l \in \mathrm{P}_2 = \mathrm{P}_3$ with $q < l$, it holds that $l \in \mathrm{P}_1$, where $q \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}_1) : {\mathsf{r}_1}^{(\geq m)} \neq {\mathsf{r}_2}^{(\geq m)}\} = \{m \in \mathsf{rng}(\mathsf{r}_1) : {\mathsf{r}_1}^{(\geq m)} \neq {\mathsf{r}_3}^{(\geq m)}\}$. Thus, $s_1 \prec_{\mathfrak{D}}^{\mathrm{DP}} s_3$.

- If $\mathsf{r}_1 \neq \mathsf{r}_2 \neq \mathsf{r}_3$ then $\widetilde{\mathsf{r}}_2^{(>q_1)} \subseteq \widetilde{\mathsf{r}}_1^{(>q_1)}$, $\widetilde{\mathsf{r}}_3^{(>q_2)} \subseteq \widetilde{\mathsf{r}}_2^{(>q_2)}$, and, for all priorities $l_1 \in \mathrm{P}_2$ and $l_2 \in \mathrm{P}_3$ with $q_1 < l_1$ and $q_2 < l_2$, it holds that $l_1 \in \mathrm{P}_1$ and $l_2 \in \mathrm{P}_2$, where $q_1 \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}_1) : {\mathsf{r}_1}^{(\geq m)} \neq {\mathsf{r}_2}^{(\geq m)}\}$ and $q_2 \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}_2) : {\mathsf{r}_2}^{(\geq m)} \neq {\mathsf{r}_3}^{(\geq m)}\}$. Now, let $q \triangleq \mathsf{max}\{q_1, q_2\}$. It is easy to see that $\widetilde{\mathsf{r}}_3^{(>q)} = (\widetilde{\mathsf{r}}_3^{(>q_2)})^{(>q)} \subseteq (\widetilde{\mathsf{r}}_2^{(>q_2)})^{(>q)} = \widetilde{\mathsf{r}}_2^{(>q)} = (\widetilde{\mathsf{r}}_2^{(>q_1)})^{(>q)} \subseteq (\widetilde{\mathsf{r}}_1^{(>q_1)})^{(>q)} = \widetilde{\mathsf{r}}_1^{(>q)}$. Moreover, for all priorities $l \in \mathrm{P}_3$ with $q < l$, it holds that $l \in \mathrm{P}_2$, since $q_2 < l$, and, so, $l \in \mathrm{P}_1$, since $q_1 < l$. Therefore, it remains to show that $q_1 \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}_1) : {\mathsf{r}_1}^{(\geq m)} \neq {\mathsf{r}_3}^{(\geq m)}\}$. Due to the maximality in the definition of q, it holds that ${\mathsf{r}_1}^{(>q)} = {\mathsf{r}_2}^{(>q)} = {\mathsf{r}_3}^{(>q)}$. If $q = q_1 > q_2$ then ${\mathsf{r}_1}^{(\geq q)} \neq {\mathsf{r}_2}^{(\geq q)} = {\mathsf{r}_3}^{(\geq q)}$. If $q = q_2 > q_1$ then ${\mathsf{r}_1}^{(\geq q)} = {\mathsf{r}_2}^{(\geq q)} \neq {\mathsf{r}_3}^{(\geq q)}$. Finally, if $q = q_2 = q_1$ then ${\mathsf{r}_3}^{(\geq q)} \subset {\mathsf{r}_2}^{(\geq q)} \subset {\mathsf{r}_1}^{(\geq q)}$. The latter is due to the fact that $(\mathsf{r}_1, \zeta_1, p_1) \prec_{\mathfrak{D}}^{\mathrm{PM}} (\mathsf{r}_2, \zeta_2, p_2)$ and $(\mathsf{r}_2, \zeta_2, p_2) \prec_{\mathfrak{D}}^{\mathrm{PM}} (\mathsf{r}_3, \zeta_3, p_3)$. So, ${\mathsf{r}_1}^{(\geq q)} \neq {\mathsf{r}_3}^{(\geq q)}$. Hence, the thesis holds.

To conclude the proof, observe that, due to Items 3.b and 3.c of Definition 2.2.4, $\top_{\mathfrak{D}}^{\mathrm{DP}} \triangleq (\top_{\mathfrak{D}}^{\mathrm{PM}}, \varnothing, \emptyset)$ is the only state in $\mathrm{S}_{\mathfrak{D}}^{\mathrm{DP}}$ having $\top_{\mathfrak{D}}^{\mathrm{PM}}$ as first component. Now, being $\top_{\mathfrak{D}}^{\mathrm{PM}}$ the maximal element of $\mathrm{S}_{\mathfrak{D}}^{\mathrm{PM}}$ *w.r.t.* $\prec_{\mathfrak{D}}^{\mathrm{PM}}$, there cannot be a state $s \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{DP}}$ such that $\top_{\mathfrak{D}}^{\mathrm{DP}} \prec_{\mathfrak{D}}^{\mathrm{DP}} s$. Moreover, $s \prec_{\mathfrak{D}}^{\mathrm{DP}} \top_{\mathfrak{D}}^{\mathrm{DP}}$ holds, for all states $s \triangleq ((\mathsf{r}, \zeta, p), \widetilde{\mathsf{r}}, \mathrm{P}) \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{DP}}$ with $s \neq \top_{\mathfrak{D}}^{\mathrm{DP}}$. Indeed, $(\mathsf{r}, \zeta, p) \prec_{\mathfrak{D}}^{\mathrm{PM}} \top_{\mathfrak{D}}^{\mathrm{PM}}$, since $(\mathsf{r}, \zeta, p) \neq \top_{\mathfrak{D}}^{\mathrm{PM}}$. So, Item 3.a is verified. Furthermore, as already shown in the proof of Lemma 2.2.1, we have that $\mathsf{r} \neq \varnothing$. Hence, Item 3.b holds as well, since Items 3.b.iii and 3.b.iv are immediately satisfied. □

Lemma 2.2.8 (Query Function for DP)**.** *The function* $\Re_{\mathfrak{D}}^{\mathrm{DP}}$ *is an query function,* i.e.*, for all states* $s \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{DP}}$*, it holds that* (1) $\Re_{\mathfrak{D}}^{\mathrm{DP}}(s) \in \mathrm{Rg}_{\mathfrak{D}}$ *and* (2) *if* $\Re_{\mathfrak{D}}^{\mathrm{DP}}(s) \in \mathrm{Rg}_{\mathfrak{D}}^{-}$ *then* $s \succ_{\mathfrak{D}}^{\mathrm{DP}} \Re_{\mathfrak{D}}^{\mathrm{DP}}(s)$.

Proof. The result immediately follows from the Query Function for PM Lemma 2.2.5 and the simple observation that, for all DP states $s \triangleq ((\mathsf{r}, \zeta, p), \_, \_) \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{DP}}$, it holds that: *(1)* $s^{\natural} \triangleq (\mathsf{r}, \zeta, p) \in \mathrm{S}_{\mathfrak{D}}^{\mathrm{PM}}$, *(2)* $\Re_{\mathfrak{D}}^{\mathrm{DP}}(s) = \Re_{\mathfrak{D}}^{\mathrm{PM}}(s^{\natural})$, and *(3)* $s \succ_{\mathfrak{D}}^{\mathrm{DP}} (\mathrm{R}, \alpha)$ iff $s^{\natural} \succ_{\mathfrak{D}}^{\mathrm{PM}} (\mathrm{R}, \alpha)$, for all open region-pairs $(\mathrm{R}, \alpha) \in \mathrm{Rg}_{\mathfrak{D}}^{-}$. This is due to the fact that both the definitions of the compatibility relation $\succ_{\mathfrak{D}}^{\mathrm{DP}}$ and the function $\Re_{\mathfrak{D}}^{\mathrm{DP}}$ do not depend on the two accessory components of the states. □

Lemma 2.2.9 (Successor Function for DP)**.** *The function* $\downarrow_{\ni}^{\mathrm{DP}}$ *is an successor function,* i.e.*, for all states* $s \in \mathrm{S}_{\ni}^{\mathrm{DP}}$ *and region pairs* $(\mathrm{R}, \alpha) \in \mathrm{Rg}_{\ni}^{-}$ *with* $s \succ_{\ni}^{\mathrm{DP}} (\mathrm{R}, \alpha)$*, it holds that* (1) $s \downarrow_{\ni}^{\mathrm{DP}} (\mathrm{R}, \alpha) \in \mathrm{S}_{\ni}^{\mathrm{DP}}$ *and* (2) $s \downarrow_{\ni}^{\mathrm{DP}} (\mathrm{R}, \alpha) \prec_{\ni}^{\mathrm{DP}} s$*.*

Proof. Let $s \triangleq ((\mathsf{r}, \zeta, p), \widetilde{\mathsf{r}}, \mathrm{P}) \in \mathrm{S}_{\ni}^{\mathrm{DP}}$ be a state, $(\mathrm{R}, \alpha) \in \mathrm{Rg}_{\ni}^{-}$ an open region-pair in $\ni$ that is DP compatible with s, and $s^\star = ((\mathsf{r}^\star, \zeta^\star, p^\star), \widetilde{\mathsf{r}}^\star, \mathrm{P}^\star) \triangleq s \downarrow_{\ni}^{\mathrm{DP}} (\mathrm{R}, \alpha)$ the result obtained by computing the function $\downarrow_{\ni}^{\mathrm{DP}}$ on s and (R, α).

It is quite easy to see that $(\mathsf{r}^\star, \zeta^\star, p^\star) \in \mathrm{S}_{\ni}^{\mathrm{PM}}$ and $(\mathsf{r}^\star, \zeta^\star, p^\star) \prec_{\ni}^{\mathrm{DP}} (\mathsf{r}, \zeta, p)$. Indeed, in case one of the two macros $\#Assignment(\xi)$ and $\#InstantPromotion$ of Algorithm 8 is executed, one can derive the thesis by applying the reasoning used in the proof of Lemma 2.2.6, since the instructions concerning the three components $\mathsf{r}^\star$, $\zeta^\star$, and $p^\star$ are exactly the same of the ones in Algorithm 7. If the $\#DelayedPromotion$ macro is considered, due to Item 1.a of Definition 2.2.4 the thesis follows by applying the reasoning used for the $\#InstantPromotion$ macro.

At this point, to conclude the proof, we need to show that Items 1 and 3.b of the same definition hold on $s^\star$. Item 1.c is trivially verified, since, whenever a priority z is added to P in order to build $\mathrm{P}^\star$, it is necessarily associated with a region in $\mathsf{r}^\star$, so, it is contained into the range of $\mathsf{r}^\star$. Also Item 1.d is easy to check, since $\mathsf{r}^\star$ does not contains priorities less than z having opposite parity, so, those present in P are eliminated due to the intersection with the range of $\mathsf{r}^\star$. In case of a delayed promotion, Items 1.a and 1.b are immediate, since $\mathsf{r}^\star$ is a region function and $\widetilde{\mathsf{r}}^\star$ is empty. Similarly, in an instant promotion or in an assignment with symbolic parameter ξ set to $\mathtt{f}$, these items hold because $\widetilde{\mathsf{r}}$, together with r, already satisfies the two items and either $\widetilde{\mathsf{r}}^\star$ is a restriction of $\widetilde{\mathsf{r}}$ or they are equal. If an assignment with symbolic parameter ξ set to $\mathtt{t}$ is performed, the α-region R is added to both r and $\widetilde{\mathsf{r}}$ with priorities p and $p^\star > p$, respectively, so, Item 1.b is verified. Item 1.a, instead, follows by applying Proposition 2.2.1 to the regions $\mathrm{R}^\natural \triangleq \mathsf{r}^{-1}(p^\star) \cup \widetilde{\mathsf{r}}^{-1}(p^\star)$ and R.

Finally, we have to show that, if $\mathsf{r}^\star = \mathsf{r}$, then $\widetilde{\mathsf{r}} \subseteq \widetilde{\mathsf{r}}^\star$ and $\mathrm{P}^\star = \mathrm{P}$ else $\widetilde{\mathsf{r}}^{(>q)} \subseteq \widetilde{\mathsf{r}}^{\star(>q)}$ and, for all priorities $l \in \mathrm{P}$ with $q < l$, it holds that $l \in \mathrm{P}^\star$, where $q \triangleq \mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}^\star) : \mathsf{r}^{\star(\geq m)} \neq \mathsf{r}^{(\geq m)}\}$. In case the macro $\#Assignment(\xi)$ is executed, the required property immediately follows by Lines 4 and 5, since $\widetilde{\mathsf{r}} \subseteq \widetilde{\mathsf{r}}^\star$ and $\mathrm{P}^\star = \mathrm{P}$, independently of the fact that $\mathsf{r}^\star = \mathsf{r}$ holds or not. Instead, if one of the two promotion macros is performed, we necessarily have that $\mathsf{r}^\star \neq \mathsf{r}$. Now, first note that $\mathsf{max}\{m \in \mathsf{rng}(\mathsf{r}^\star) : \mathsf{r}^{\star(\geq m)} \neq \mathsf{r}^{(\geq m)}\} = p^\star$. Since $\mathsf{r}^\star$ preserves all regions having priorities greater than $p^\star$, we have that $\mathrm{P}^\star$ contains all such priorities that already occur in P. Moreover, by Line 4 of the instant promotion, we have that $\widetilde{\mathsf{r}}^{\star(>q)} = \widetilde{\mathsf{r}}^{(>q)}$. The same holds by Lines 1 and 4 of the delayed promotion, since $\widetilde{\mathsf{r}}^{(>q)} = \emptyset$ Consequently, the thesis follows. □

Also in this case, we can observe that the following mapping $\mathsf{h} : ((\mathsf{r}, \zeta, p), \_, \_) \in \mathrm{S}_{\mathcal{D}}^{\mathrm{DP}} \mapsto (\mathsf{r}, \zeta, p) \in \mathrm{S}_{\mathcal{D}}^{\mathrm{PM}}$, which takes DP states to PM states by simply forgetting the additional elements $\widetilde{\mathsf{r}}$ and P, is indeed a homomorphism. This leads to the following theorem.

Theorem 2.2.8 (DP Size & Depth Upper Bound)**.** *The size of a* DP *region space* $\mathcal{R}_{\mathcal{D}}^{\mathrm{DP}}$ *for a game* $\mathcal{D} \in \mathrm{PG}$ *with* $n \in \mathbb{N}_+$ *positions and* $k \in [1, n]$ *priorities is bounded by* $2^k(2k)^{n+2}$*. Moreover, its depth is not greater then that of the* PM *region space* $\mathcal{R}_{\mathcal{D}}^{\mathrm{PM}}$ *for the same game.*

It is worth noting that this procedure only requires a quadratic number of promotions on both the lower bound games for PP and PM. It also appears to be much more robust, in terms of preventing promotion revocations, than the PM technique alone, to the point that it does not seem obvious whether an exponential lower bound even exists. Further investigation is, however, needed for a definite answer on its actual time complexity.

Chapter 3

Implementation

In this chapter we shall describe how the priority promotion mechanism described in the previous chapter can be implemented in a framework called PGSolver, which provided implementations of most of the parity games solvers proposed in the literature. This will allow us to conduct an experimental assessment of newly proposed solutions in Chapter 4.

Then after presenting the PGSolver framework, the main data structures and the basic algorithms used by the solvers, a detailed description of the implementation of the *query* and the *successor* function of the Priority Promotion is provided. The complete source code for all the algorithms described in Chapter 2 can be found in the appendix.

3.1 Parity Game Solver

PGSolver is a program written in Objective Categorical Abstract Machine Language (OCAML) language, a highly used functional programming language, developed by the French INRIA and distributed as an open source project. The OCAML comes from the ML family and is a multi paradigm programming language, which also incorporate imperative and object oriented extensions.

3.1.1 Features

The framework PGsolver is composed of five modules: **pgsolver**, **generators**, **tools**, **solvers** and paritygame based on a library, that is tcslib. Three of these modules, namely **pgsolver**, **generators** and **tools**, can be accessed independently via command line. The module **pgsolver** contain the program

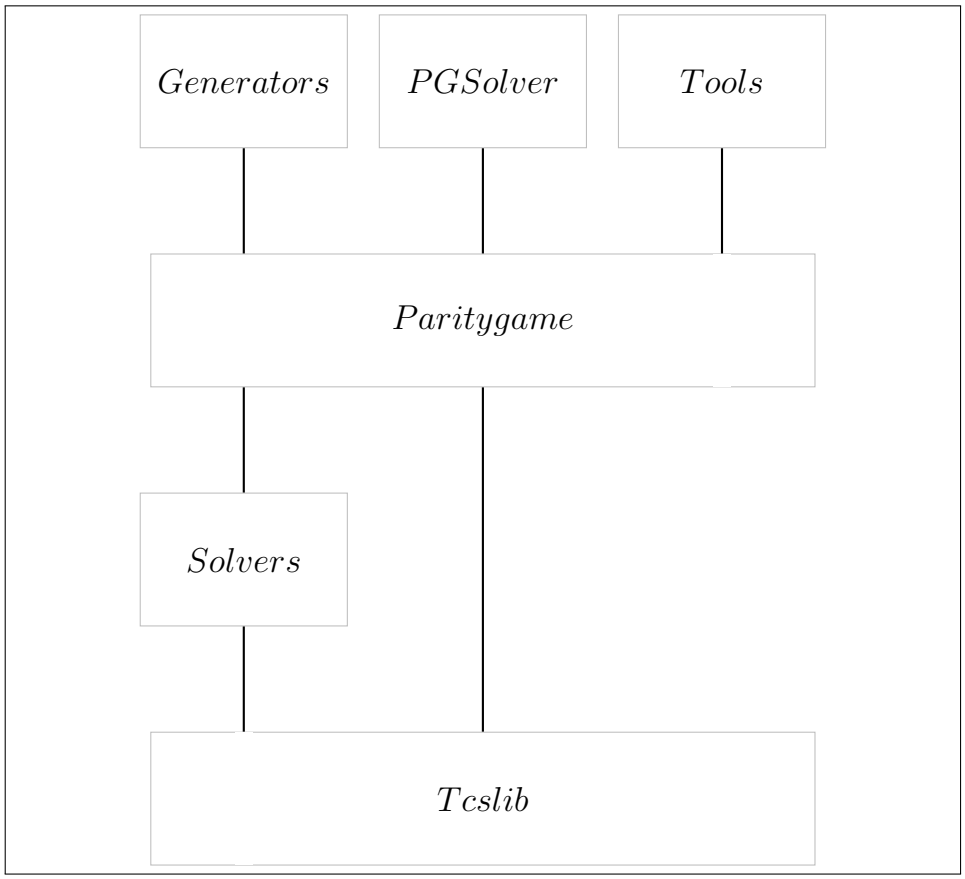


Figure 3.1: The PGSolver architecture.

of interface between the user and the implemented parity game solvers. In order to solve a parity game using a specific solver the following command is used:

```
> pgsolver -global <solver> <gamename>
```

Where the *global* keyword specify that the game must be solver globally, that is for all $v \in V$ as starting position.

Beyond the optional parameters the two input of pgsolver are the name of solver and the name of the file containing the parity game to solve. Such file must begin with the keyword *parity* followed by the maximum identified of vertex, the others line of the file contain the list of vertices. The content of the file of the game of Figure 1.2 is:

```
parity 5;
0 5 1 1 "a";
1 6 0 2,3 "b";
2 3 1 3 "c";
3 0 0 4,5 "d";
4 3 1 0,1 "e";
5 7 1 3 "f";
```

The grammar to which game files must conform is reported below in EBNF format.

⟨*parity game*⟩ ::= parity ⟨*node_id*⟩; ⟨*arena*⟩

⟨*arena*⟩ ::= { ⟨*node_id*⟩ ⟨*priority*⟩ ⟨*player*⟩ [⟨*adjacency_list*⟩] ⟨*name*⟩; }

⟨*adjacency_list*⟩ ::= ⟨*node_id*⟩ { , ⟨*node_id*⟩ }

⟨*name*⟩ ::= ⟨*quoted_string*⟩ | none

⟨*player*⟩ ::= 0 | 1

where the non-terminals ⟨*node_id*⟩ and ⟨*priority*⟩ are natural numbers, while ⟨*quoted_string*⟩ is a string enclosed within quotes, like, *e.g.*, `"0"`. The first element of type *arena* must have as *node_id* the value 0.

The module **generators** contains programs that generate various families of parity games. These families will be discussed in details in Chapter 4, where the experimental evaluation is presented.

The module **tools** provides a banckmarking tool, textsfbenchmark, wich allows to perform experiments with different solvers, together with various programs for manipulating parity games: obfuscator, compressor, combinator and transformer

Given a list of games and a list of solvers, the benchmark tool allows to compare the solutions time of those solvers on the list of games. It provides the average, the best and the worse computation times on an input number for the solution attempts for each solver-game pair.

Given a game, the obfuscator permutes the identifier of the vertices, in order to prevent solvers from taking advantage of their order, especially analyzing their successors. To this end, random permutation are applied.

The compressor module modifies identifiers and priorities of a game by applying the compaction technique described in Section 1.2.6. A similar approach is applied to the vertices identifiers as well. It also tries to reduce the number of priorities, by pushing priorities along edges of the game graph and overriding the smaller ones.

The combinator module merges two or more different games, by computing the disjoint union of the underlying graphs, without adding any new edges between them.

Given a parity game, the transformer generates an somehow equivalent parity game enforcing some additional properties such: as strict alternation between players turns; congruence of the parity of the player and priority of each vertex; presence of zero priority vertices between two vertices connected by an edge; uniqueness of priorities in the game; presence of winning cycles for each player reachable from vertices of the opponent; presence of a vertex for each different priority.

The module **paritygame** is the library that contains the parity game definitions and all the functions needed to handle parity games. It contains functions to access, order, remove, add and collect vertices, as well as functions to create subgames, compute SCCs and attractors.

The module **solvers** contains among the others the following solver implementations: recursive (that is the Zielonka's algorithm), strategy improvement, small progress measure, dominion decomposition and big step.

The library **tcslib** provides definitions of basic data structure such as arrays, lists, queues and sets. It also provides dedicated data structures for manipulating automata and formulas.

3.1.2 Representation of basics operation on parity games

The main data structure is clearly the game. Parity games are stored as arrays of vertices. Each vertices has four attributes, the priority, the player, the array of successors and the name. The definition is:

```
type paritygame = (int * int * int array * string option) array
```

The declaration of the game of Figure 1.2 is:

```
[|(5,1,[|1|],Some("a")),(6,0,[|2;3|],Some("b")),(3,1,[|3|],Some("c"))
  ,(0,0,[|4;5|],Some("d")),(3,1,[|0;1|],Some("e")),(7,1,[|3|],Some("f"))
  |]
```

Given a parity game, a solver algorithm must return the winning set and the winning strategy for each player in the game, in symbols the pairs (W_0, σ_0) and (W_1, σ_1). The solution of the game consists of a partition of V, stored within a single array. Each index of the solution array contains the winning player for the vertex identified by the index. Under this interpretation, the solution is a function $sol : V \to Pl$, with $Pl \in \{0, 1\}$, which associates a player with each vertex.

$$sol[v] = \begin{cases} 0 & \text{if } v \in W_0 \\ 1 & \text{if } v \in W_1. \end{cases}$$

The strategy is stored as a single array as well. Each index of the strategy array contains the correct move computed by the solver algorithm. Clearly, the algorithm returns a strategy for v if and only if $v \in V_\alpha \cap W_\alpha$, with $\alpha \in \{0, 1\}$. The strategy is a partial function that associate a vertex to another vertex. Formally, $str : V \to V$ is defined as follows:

$$str[v] = \begin{cases} \sigma_0(v) & \text{if } v \in W_0 \cap V_0 \\ \sigma_1(v) & \text{if } v \in W_1 \cap V_1 \\ -1 & \text{if } v \in (W_0 \cap V_1) \cup (W_1 \cap V_0) \end{cases}$$

The algorithms that solve parity games by computing winning sets maximize the current winning set using the attractor before iterating the process on subgames. In fact, given a winning set W_α for game $\mathcal{D}$, the set $W_\alpha' = \mathsf{atr}^\alpha_{\mathcal{D}}(W_\alpha)$ is still winning, since the attractor collects all the positions from where α can force the game to reach W_α. Since W_α is an $\overline{\alpha}$-trap, once the game enters in W_α, player $\overline{\alpha}$ is not able to exit.

Algorithm 12: The Attractor Operator.

```
   signature atr : (PG × 2^Ps_D × α) →_D 2^Ps_D
   function atr^α_D(N)
       let (V_0, V_1, E) = D in
1          attr ← ∅
2          E^T ← trasp(E)
3          Q ← ⋃_{v∈N} E^T(v)
4          while Q ≠ ∅ do
5              v ← extract(Q)
6              if v ∉ attr then
7                  ws ← E(v)
8                  if (v ∈ V_α ∧ (ws ∩ attr) ≠ ∅) ∨ (v ∈ V_ᾱ ∧ ws ⊆ attr) then
9                      attr ← attr ∪ {v}
10                     Q ← Q ∪ E^T(v)
11         return attr
```

The parameters of the attractor (Algorithm 12) are the game $\mathcal{D}$, the player α and the set of vertices N. The data structure used to store the attractor is the array *attr* at line 1. At line 3 the transposed graph is computed. Line 4 loads into queue Q the predecessors of the vertices in N. A while loop, then,

extracts a vertex v from Q (line 5) and checks if it belong to the attractor (line 6). If not, the successors of v are collected in the array ws (line 7). Then v is added to the attractor (line 9) if either $v \in V_\alpha$ and at least one of its successor is in the attractor already or $v \in V_{\overline{\alpha}}$ and all its successors belong to the attractor. Finally the predecessor of v are added into Q. The process is iterated until Q becomes empty.

Algorithms based on decomposing the game require to compute as many subgames as the needed iterations. Given a parity game $\mathfrak{d}$ and a set of vertices N, the subgame obtained by the removal of N is denoted by $\mathfrak{d}' \triangleq \mathfrak{d} \setminus \mathrm{N}$. Since on big parity games often N is small, the subgame construction modifies the current game by removing N as well as the edges connected to them.

Algorithm 13: The Subgame Function.

```
   signature subgame : (PG × 2^Ps_∋) →_∋ PG
   function subgame(∋, N)
       let (V_0, V_1, E) = ∋ in
1          V_0 ← V_0 \ N
2          V_1 ← V_1 \ N
3          for v ∈ V_0 ∪ V_1 do
4              E(v) ← E(v) ∩ V_0 ∪ V_1
5      return (V_0, V_1, E)
```

The function that generates subgames (Algorithm 13) has two parameters: the game $\mathfrak{d} = (V_0, V_1, E)$ and a list of vertices N. The commands of line 1 and 2 remove from V_0 and V_1 the vertices in N. At line 4 from the adjacency list of each vertex are examined and only those not in N are retained. Note that every algorithm uses as set N an α-trap with $\alpha \in \{0, 1\}$ player, that is an $\overline{\alpha}$-attractor. In fact, due the properties of attractor, the removal of the attractor from a parity game ensures that no sink will be present in the subgame.

3.2 Our new algorithm

In this section we explain why, unlike others parity game solvers, the new idea needs sets operation that can not be efficiently performed with the structures such as arrays and lists. Then, we explain the supplementary data structure we adopted in addition. Finally we explain the implementation of the basic Priority Promotion Algorithm.

At every step of our algorithm, we need to check if the extracted region is open. This is done by computing the α-escape. The operations needed to implement the α-escape are the α-predecessor,

the intersection and the difference on sets of positions. The best structure we found to perform these operations is the *bitset* provided in the *extlib* library. A Bitset is an array of boolean values (bits) and the operations provided by the library include functions to perform extraction of set elements, set difference, union and intersection, etc. Being implemented as an array, random access to its elements is guaranteed. *Bitset* stores bits, therefore, it requires 1/8 of the memory used by conventional arrays.

Recall that the states defined for the Parity Promotion algorithm (Definition 2.2.1) have the form $s \triangleq (\mathsf{r}, \zeta, p)$s. The region function r is stored in an array of colours indexed by positions, the boolean variable *is_up* in a boolean variable and the priority p in an integer number. Whenever we get data form the array of the region function, we store it into *bitset* and, when these data are not involved in set operations, we extract the positions set into an *enumetarion*, which is another data structure provided by the *extlib* library. *Enumerations* are disposable abstract collections of elements, in this case positions, on which iterators can be used.

3.2.1 Implementation of Priority Promotion

The algorithm for Priority Promotion is implemented as a single iterative monolithic function. The source code of the algorithms and the supplementary functions can by found in the appendix.

Since the three algorithms proposed in this thesis are all evolutions of the basic Priority Promotion solver, in the following we shall focus on the implementation of the basic Algorithm 2.2.1. In particular we shall describe the implementation of the Query and Successor functions.

The *Query Function* (algorithm 5) extracts a region from the current game. The state $s \triangleq (\mathsf{r}, \zeta, p)$ is encoded as follows. The array r stores the region function r and the boolean variable *is_up* stores the boolean flag ζ. Differently from the abstract algorithm, the implementation does not save in a single variable the priority p of the current state s. In fact, it is contained in variable m immediately after a promotion, and in variable n, otherwise. The value α for the player (line 1 in Algorithm 5) is recorded in variable *player* at line 239. The computation of the set N of positions with priority p (line 2 in the algorithm) is performed by lines 229-238, depending on the value of the flag *is_up*. If *is_up* is true, a for loop collects all the indexes of the positions in r with priority m (lines 232-235). Variable m is, in this case, equal to the target priority of the promotion performed. Otherwise, N contains the set of positions with the maximal colour n in the current subgame. The attractor of line 3 is computed at line 241.
The *Successor* function is encoded by the code blocks in Figures 3.3,3.4 and 3.5. Similarly to the Query function, the region function r is stored in array r, the boolean flag ζ in variable *is_up*, the priority p in variable n and the player α in variable *player*. Moreover, the region R is stored in variable *a_en*,

```
229 let target = if !is_up
230   then (let r_m = BitSet.create l in
231         for i = 0 to l - 1 do
232             if r.(i) = m
233             then BitSet.set r_m i
234         done;
235         r_m)
236   else collect_nodes_by_prio game' n l in
237 let player = n mod 2 in
238 let tar_en = BitSet.enum target in
239 let a = attr_bit game' strategy player target
240                 (Enum.clone tar_en) in
```

Figure 3.2: The Query function.

which is implemented as a list of vertices. The code block in Figure 3.3 takes care of the if case of the *Successor* function, while the blocks in Figures 3.4 and 3.5 correspond to the $else$ case. The test whether region R is open (line 1 of Algorithm 6) is computed at line 256, where the escape set $\mathsf{esc}^{\alpha}_{\mathcal{D}}(\mathrm{R})$ of region R is collected. In the implementation, the priority of the current state p is not recorded in a variable, since it can be computed starting for the information contained in the region function. This is done by the code block at lines 221-228. First the value m of the minimal priority currently contained in r, which corresponds to the priority of the last region inserted in the region function, is computed at line 210. Priority p is computed as the maximum priority in the subgame $\mathcal{D}_{\mathsf{r}}^{<m(\zeta)}$ with $m(\zeta) = m$ if $\zeta = false$, $m+1$, otherwise. Finally, the maximal priority in the subgame is computed and stored in n. The update of the region function with the new region R with priority p (line 2 of the algorithm) is computed by means of an iterator over the enumeration a_en, containing R, at line 258. The new value of the flag ζ (line 4 of the algorithm) is set at line 259.

The $else$ case at lines 5-7 of the algorithm begins with computation of $\mathsf{brp}^{\overline{\alpha}}_{\mathcal{D}}((\mathrm{R},\mathrm{r}))$ function. In the implementation this function is computed separately, exactly at lines 267-284. Lines 267-273 collect the positions of α contained into region function r and stores them in the bitset r_pl. At line 274 the maximal priority of the game is saved as temporary $\mathsf{brp}^{\overline{\alpha}}_{\mathcal{D}}((\mathrm{R},\mathrm{r}))$. The successive lines execute an iterator over R stored in enumeration a_en. The iteration search for all $w \in ws$ of the positions $v \in a\_en$ the minimum colour of the position w such as $w \in r\_pl$.

Figure 3.3: The IF case of Priority Promotion successor function.

```
221 let r_dom =
222     if !is_up then bit_dom_array_bound r l m
223      else bit_dom_array_bound r l (-1) in
224        let game' = Array.copy game in
225              let r_dom_en = BitSet.enum r_dom in
226                pg_remove_nodes_bit game' r_dom
227                     (Enum.clone r_dom_en) l;
228                let n = max_prio game' in
229              ...

258 Enum.iter (fun i -> r.(i) <- n;) a_en;
259 is_up := false;
```

Lanes 6 and 7 of the algorithm are coded in the block of lines 287-295. The update of line 6 of the algorithm is computed by the for loop at line 287. At lines 288 and 289, the positions in a, storing R, are coloured with priority min, that store the $\mathsf{brp}^{\overline{\alpha}}_{\mathcal{D}}((\mathrm{R}, \mathrm{r}))$. The position with priority lower than min are deleted from r at line 291. Finally the flag set at line 7 of the algorithm is updated at line 295.

```
267 let r_pl = BitSet.create l in
268 if player = 0 then Enum.iter (fun i -> if r.(i) mod 2 = 0
269                                        then BitSet.set r_pl i)
270                               r_dom_en
271 else Enum.iter (fun i -> if r.(i) mod 2 = 1
272                          then BitSet.set r_pl i)
273                r_dom_en;
274 let min = ref (max_col) in
275  Enum.iter (fun v -> let (_,_,ws,_) = game.(v) in
276    let color = ref (max_col) in
277    Array.iter (fun w -> if BitSet.is_set r_pl w
278                         then (if r.(w) < !color
279                               then (
280                               color := r.(w);
281                               if r.(w) < !min
282                               then min := r.(w); )))
283                ws;)
284   (Enum.clone a_en);
```

Figure 3.4: The brp function.

```
287 for i = 0 to l - 1 do
288   if BitSet.is_set a i
289   then r.(i) <- !min
290   else (if r.(i) < !min
291         then (r.(i) <- -1;
292         ... ); );
293   ...
294 done;
295 is_up := true;
```

Figure 3.5: The ELSE case of Priority Promotion successor function.

Chapter 4

Experimental evaluation

This chapter is dedicated to the experimental comparison of the state-of-the-art solvers proposed in the literature and implemented in PGSolver. In fact, a good theoretical time complexity upper bound is, in general, not enough to ensure good performances in practical applications. Running tests is, therefore, crucial to understand whether an algorithm has some practical value or not. For this reason, we implemented and tested our algorithms against the best solvers proposed to date. As explained in Chapter 3, we used the PGSolver framework and its benchmark tool.

Usually, solvers are encapsulated into the generic solver of PGSolver, called *universal solver*, that first applies the optimizations described in Section 1.2.6 to the games and then runs the specific solver. To measure the performances intrinsic to the considered solution technique, we, therefore, have isolated the code of each solver from the universal one.

We tested the solvers on three classes of games: encodings of real problems coming from the formal verification field, well-know exponential problems for each solving technique, and synthetic randomly generated games. Experiments were carried out on a 64-bit 3.1GHz Intel® quad-core machine, with i5-2400 processor and 8GB of RAM, running Ubuntu 12.04 with Linux kernel version 3.2.0. PGSolver was compiled with OCaml version 2.12.1.

The algorithms considered in the experimentation are the Zielonka algorithm *Rec*, its two dominion decomposition variants, *DomDec* and *BigStp*, the strategy improvement algorithm *StrImp*, the small progress measure algorithm *SmlProg*, and the three algorithms proposed in this thesis, PP, PM and DP.

4.1 Concrete Problems

As we said in the introduction, the interest for parity games comes from their relation with model checking problems. Therefore, the computation time for solving algorithms on real problems of verification is of great importance. PGSolver provides three implementation of verification problems: Tower of Hanoi, Elevator and Language Inclusion.

Tower of Hanoi The Tower of Hanoi puzzle consists of tree spindles and n disks. The disks are arranged in order of diameters: the largest at the base, the smallest on the top. At the beginning of game, all the disks are arranged on the first spindle and the others two are empty. The problem is to move all disks, one at time, from the first spindle to the third, using the second one to temporarily store moved disks. The difficulty of problem is that it it not allowed to place a larger disk on a smaller one.

Its translation into a graph consists in a reachability problem, where we want to verity whether the vertex that represents the correct final state of game is reachable from the vertex that representing the starting state. Figure 4.1 shows the form of the gadgets that compose the graph. Instance 3 of the problem needs 3 gadgets. Instance 4 needs 3×3 gadgets. So the number of vertices triples every time a disk is added to problem.

The game is a single player game, hence $V_0 = V$, and all the vertices have either priority 0 or priority 1. The final states are represented by vertices with a self-loop, and only one of them has priority 0. Since all the vertices belong to player 0, the attractor of this vertex attracts the entire game.

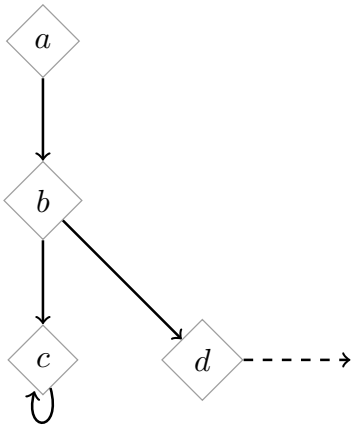


Figure 4.1: The Tower of Hanoi gadget.

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
6	0.8	16.6	0.3	†	**0**	**0**	**0**	**0**
7	5.5	†	2.2	†	**0**	**0**	**0**	**0**
8	40.8	†	15.1	†	**0.1**	**0.1**	**0.1**	**0.1**
9	†	†	106.2	†	0.6	**0.5**	**0.5**	**0.5**
10	†	†	†	†	1.8	1.6	1.6	**1.5**
11	†	†	†	†	5.5	**4.9**	**4.9**	**4.9**

Table 4.1: Execution times in seconds for Tower of Hanoi games. Time out (†) is set to 180 seconds.

Table 4.1 shows the results of all the solvers on the problem. Strategy Improvement exhibits the worst performances on this problem, indeed it is unable to handle even the smallest instances. Similar consideration applies the Dominion Decomposition algorithm. The growth tendency of Small Progress Measure and Big Step is much faster then the tendency of the remaining. Compared to with our solvers, the only competitive algorithm is Zielonka which however is always out performed by the new ones.

Elevator The Elevator problem simulates the management of an elevator. There are n floors and at any time the elevator can be requested from any floor. Repeated requests from the same floor are overridden and it is possible to no request is sent. The elevator can move one floor at a time to reach the next requested floor in the queue. However it stops at the first floor encountered from which a service request was received, regardless of its position in the queue. The state of the problem stores the floor where the elevator is currently positioned, the state of doors (open or closed) and the list of service requests.

The problem is to check whether this elevator model satisfy the following fairness property. If the top floor request service infinitely often, it must be served infinitely many times. All the strategies that do not guarantee player 0 to visit states in which the top floor has requested and received the service are loosing. Therefore, to encode the problem as a parity game its suffices to set the priority of those vertices to be the highest even priority.

Results in Table 4.2 show that Dominion Decomposition algorithm performs badly for this problem, hitting timeout on all the tested instances. Strategy Improvement does not perform much better. Also in this case the best performance is exhibited by the new algorithms.

Language Inclusion Given the automata $A_{n,m}$, defined on the alphabet $\Sigma_n = a_1, a_2, \ldots, a_n$, the language $L(A_{n,m})$ contains all the words with infinitely many occurrences of $(a_i)^m$, for some $i \in [1, n]$. The problem is to check if $L(A_{n,m}) \subseteq L(B_{n,m})$ with $A_{n,m}$ non-deterministic Büchi automaton and $B_{n,m}$

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
2	**0**	†	**0**	0.1	**0**	**0**	**0**	**0**
3	**0**	†	**0**	2.5	**0**	**0**	**0**	**0**
4	0.2	†	0.1	†	**0**	**0**	**0**	**0**
5	2.3	†	0.6	†	0.3	**0.1**	**0.1**	**0.1**
6	26.9	†	6.5	†	2.0	**1.2**	**1.2**	**1.2**
7	†	†	91.11	†	18.45	12.6	**11.0**	11.2

Table 4.2: Execution times in seconds for Elevator games. Time out (†) is set to 180 seconds.

deterministic Büchi automaton for the same language. Both model checking and validity problems can be encoded as language inclusion problems. For instance, the model M of a system and a temporal logic property φ can both be modelled as Büchi automata. As a consequence, M satisfies φ if and only if the language accepted by the automaton modelling M is included in the language of the automaton for φ.

The game $\Im$ generated by the tool is the synchronous product of $A_{n,m}$ and $B_{n,m}$. In $\Im$ each vertex v represent a pair of states, one for each automaton, that is $v =< p, q >$, with p a state of $A_{n,m}$ and q a state $B_{n,m}$. All the vertices in the game are owned by player 1 and the priorities of a vertex $v =< p, q >$ are assigned according to the following rule: if p is accepting then the priority is 2; if p is not accepting but q is, then the priority is 1; otherwise, the priority is 0.

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
100 20	6.5	†	†	†	0.5	**0.3**	**0.3**	**0.3**
200 20	33.2	†	†	†	2.5	**1.0**	**1.0**	**1.0**
300 20	†	†	†	†	6.7	**2.0**	2.1	2.1
400 20	†	†	†	†	14.8	**3.5**	3.7	3.9
500 20	†	†	†	†	27.8	**5.3**	**5.9**	**5.6**
600 20	†	†	†	†	48.0	9.0	**8.3**	**8.3**

Table 4.3: Execution times in seconds for Language Inclusion games. Time out (†) is set to 180 seconds.

As witnessed by the results in Table 4.3 the resulting class of games is very hard for most of the solvers. The only solver, besides our new ones, able to solve all the instances considered is *Rec*. However, its performances are much worse.

4.2 Special Families

The PGSolver framework provides the implementation of several families of games. There is at least one exponential family for each solver. Interestingly enough, on none of these families our new algorithm exhibit an exponential behaviour, thus proving that the solution approach is completely orthogonal to the existing ones.

In the following experiments time-out is set to 600 seconds and memory-out to 7,5 Gb.

Clique Vertices of a clique graphs are connected to each others. A feature of clique games is that they have an high number of edges. Two variant of this problem are possible. The first one allows self cycles, making the graph complete. The second ones prohibits self cycles. This latter variant is is harder to solve, since self cycle are often partial winning strategies and algorithms based of computation of partial strategies can benefit from this feature.

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
1000	†	2.4	†	22.9	3.5	**0.5**	**0.5**	**0.5**
1200	†	3.9	†	37.3	6.0	**0.7**	**0.7**	**0.7**
1400	†	5.9	†	59.7	10.4	**1.0**	**1.0**	**1.0**
1600	†	9.7	†	82.8	16.8	**1.3**	**1.3**	**1.3**
1800	†	12.2	†	143.3	26.8	**1.7**	**1.7**	**1.7**
2000	†	17.7	†	†	30.8	**2.1**	**2.1**	**2.1**

Table 4.4: Execution times in seconds for Clique games. Time out (†) is set to 180 seconds.

As Table 4.4 shows, progress measure base approaches have very bad performances. Also Strategy Improve algorithm struggle to solve these games. Both Dominion Decomposition and the recursive are comparable to each other while the three new algorithms significantly out performed them both.

Ladder The name of this family comes from the ladder like structure of the graph, as shown if Figure 4.2. Ladder games are composed of two chains of vertices, each one belonging to a different player. All the vertices of the chain of player 0 have priority 0, while those of the chain of player 1 have priority 1. Each chain forms a cycle, therefore, each player wins his chain. Furthermore, each vertex has an outgoing edge to a vertex of the opponent chain. The interest of this family lies in the fact that a game of size n admits 2^n strategies for each player. In fact in each vertex the corresponding player can choose to remain in its own chain or swap to the opponent one. Among the 2^n strategies for each player, only

the one choosing to remain in its own chain is winning. The structure of these games makes them hard to solve for algorithm based on guessing an initial strategy.

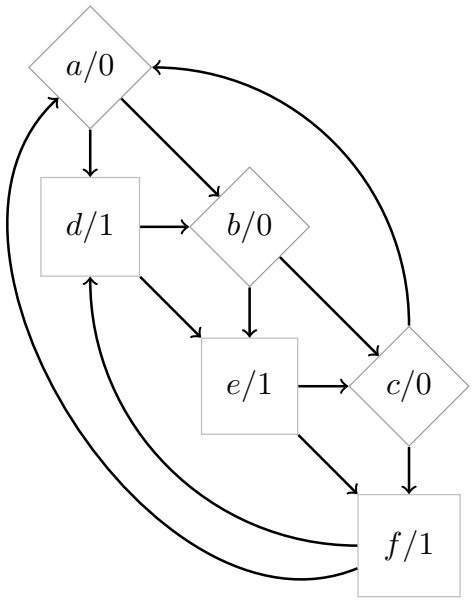


Figure 4.2: Instance of Ladder Game 3.

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
50K	3.1	†	†	†	0.8	**0.5**	**0.5**	**0.5**
60K	3.7	†	†	†	1.2	**0.6**	**0.6**	**0.6**
70K	4.3	†	†	†	1.2	**0.7**	**0.7**	**0.7**
80K	5.0	†	†	†	1.3	**0.8**	**0.8**	**0.8**
90K	5.9	†	†	†	1.5	**0.9**	**0.9**	**0.9**
100K	6.6	†	†	†	1.8	**1.1**	**1.1**	**1.1**

Table 4.5: Execution times in seconds for Ladder games. Time out (†) is set to 180 seconds.

Results of Table 4.5 shows that different algorithms have very different behaviours. In particular the best algorithms are *Rec* and the new algorithms presented in this thesis.

Recursive Ladder The Recursive Ladder is the family exponential for the Zielonka's algorithm. Its structure is composed of gadget, each containing five vertices. The $i+1$ instance of the game encapsulates

the i instance, attaching a new gadget. Figure 4.3 shows the second instance of the recursive ladder family. The left-hand side corresponds to the gadget of the instance 1, while the right-hand side the one for instance 2. The gadgets are connected to each other by three edges.

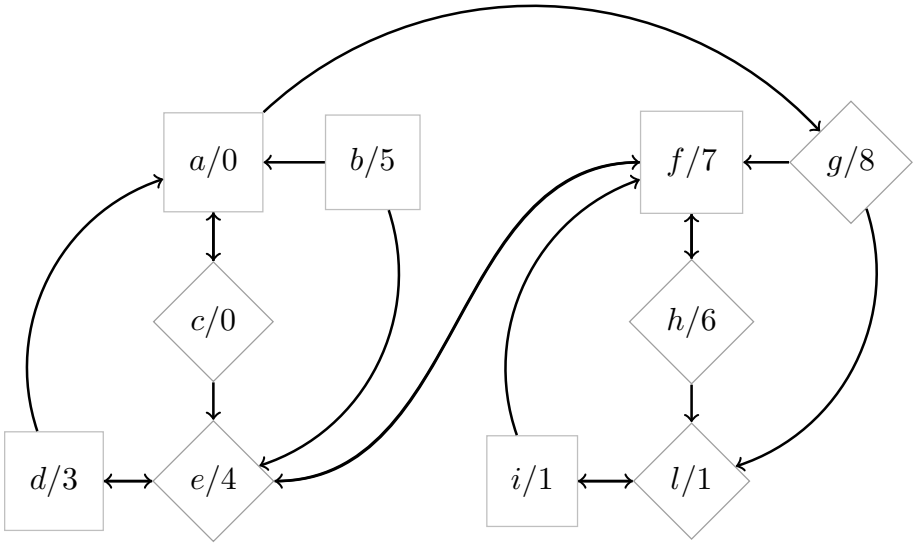


Figure 4.3: Instance of Recursive Ladder Game 2.

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
100	†	1.6	†	**0**	†	0.1	0.1	0.1
200	†	12.4	†	**0**	†	0.2	0.2	0.2
300	†	44.5	†	**0**	†	0.9	0.9	0.9
400	†	110.6	†	**0**	†	2.6	2.6	2.5
500	†	†	†	**0**	†	4.8	4.8	4.8
600	†	†	†	**0.1**	†	9.9	10.4	10.3

Table 4.6: Execution times in seconds for Recursive Ladder games. Time out (†) is set to 180 seconds.

As Table 4.4 shows, our algorithms can solve these games quite easily compared to most of the other solvers, except for Strategy Improvement, which shows surprisingly good performances.

Model Checker Ladder The Model Checker Ladder family is exponential for Model Checking algorithm [SS98]. The game is winning for player 0 but has $V = V_1$, therefore is a single player game

played by the opponent. i-th instance has 2^i cycles, which are all winning for player 0. In Figure 4.4 we show how the gadget, containing vertices a, b, c, for 2-nd instance of the game is attached to the gadget, containing vertices d, e, f, g, for the 1-st instance.

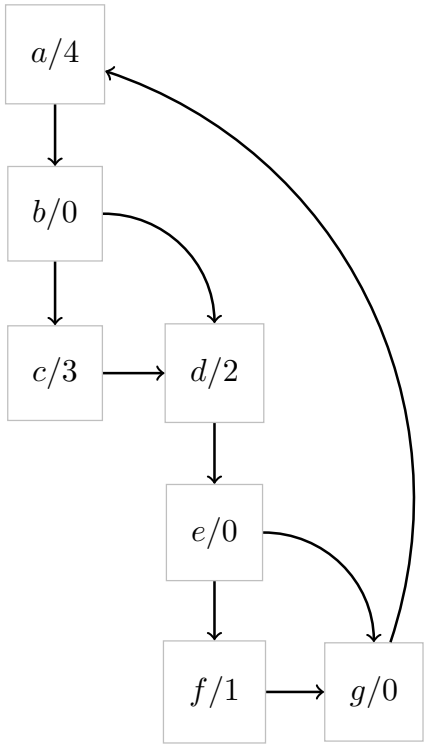


Figure 4.4: Instance of Model Checker Ladder Game 2

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
100	†	†	†	‡	**0.3**	0.4	0.4	0.4
200	†	†	†	‡	**0.4**	0.5	0.5	0.5
300	†	†	†	‡	**0.4**	0.7	0.7	0.6
400	†	†	†	‡	**0.5**	0.8	0.8	0.7
500	†	†	†	‡	**0.6**	0.9	0.9	0.8
600	†	†	†	‡	**0.7**	1.0	1.0	0.9

Table 4.7: Execution times in seconds for Model Checker Ladder games. Time out (†) is set to 180 seconds.

As shown in Table 4.7, most of the solvers cannot solve any on the tested instances, either because

time-out or memory-out. Our solvers and *Rec* are the only one able to solve the instance with comparable running times.

Exponential Strategy Improvement This is the exponential family for the Strategy Improvement algorithm, presented by Friedmann [Fri09].

Instance	SmlProg	DomDec	BigStp	StrImp	Rec	PP	PM	DP
100	†	0.4	†	†	**0.3**	**0.3**	**0.3**	**0.3**
200	†	0.9	†	†	0.7	**0.6**	**0.6**	**0.6**
300	†	2.0	†	†	1.4	**1.2**	**1.2**	**1.2**
400	†	3.4	†	†	2.3	**2.0**	**2.0**	**2.0**
500	†	4.9	†	†	3.6	3.2	**3.1**	**3.1**
600	†	6.9	†	†	5.5	**4.4**	**4.4**	5.5

Table 4.8: Execution times in seconds for Exponential Strategy Improvement games. Time out (†) is set to 180 seconds.

Table 4.5 shows that Strategy Improvement, Big Step and Small Progress Measure all exhibit vary poor performance. The Recursive algorithm and Dominion Decomposition on the other hand show growth rate comparable to ours.

Despite the fact that occasionally on few of the families some competitor solvers may have slightly better running times, the new solvers exhibit the most consistent behaviour overall on these benchmarks.

4.3 Random Games

In order to have a complete picture of the relative performance on the solvers, a number of experiments randomly generated games were performed. We considered games with different numbers of vertices, edges and priorities. Games with 2 outgoing edges per vertex are harder for all the solvers considered, because all solving techniques generate small regions and require more iteration to reach a set of vertices that leads to solution.

With more than 2 outgoing edges per vertex the games become much easier. We, then, chose to generate games with 100 outgoing edges, since games have at least 1000 vertices. For for both classes of random games, we generated games with increasing number of priorities, proportional to the number of vertices.

Since the time complexity of all the algorithms heavily depends on the number of priorities, we

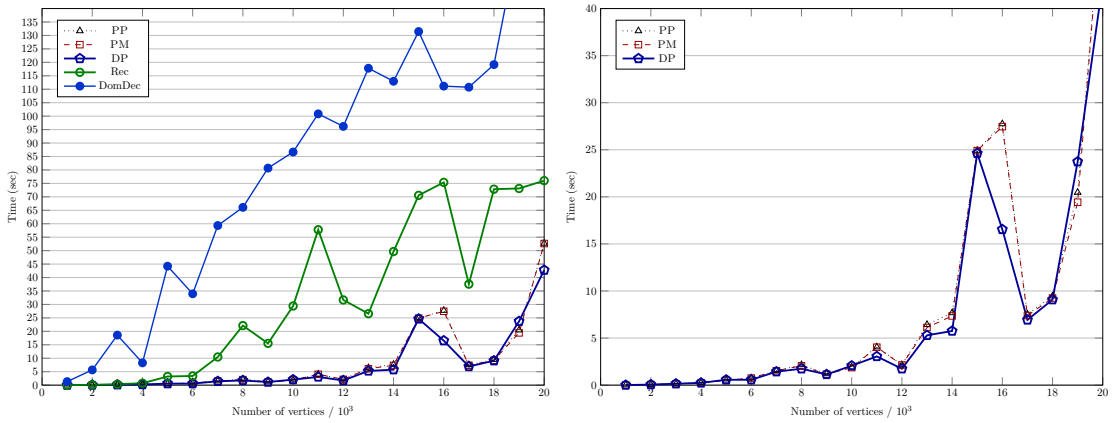


Figure 4.5: Performances on random games with 2 outgoing edges per node and 1/10 of priorities.

also considered random games with a very small number of priority *w.r.t.* the number of vertices. In particular we ran experiments on games with a logarithmic number of priorities *w.r.t.* the number of vertices.

Two outgoing edges We generate 2000 games with exactly 2 outgoing edges per vertex. We made ten clusters of run with 200 of games, incrementing the percentage of priorities related number of vertices. We show the growth rate tendency of the algorithms in each cluster, comparing our three algorithms with Recursive and Strategy Improvement. Algorithms of Big Step and Dominion Decomposition in this category of games have very poor performance, hitting timeout already on the smaller instances.

Figures 4.5-4.14 show the results on games, where the priorities is increase from $\frac{1}{10} \cdot n$ to n where n is the number of vertices. The new algorithms perform very well on all these games, regardless of the number of priorities. In each figure, the right-hand side compares the performances of the three new algorithms alone, showing that DP is often better than the other two.

One hundred outgoing edges A higher number of outgoing edges makes the games easier to solve. A possible explanation of this phenomenon is that the more the number of edges the denser the graphs are. Therefore, graphs with more edges have more chances to contain bigger and fewer SCCs. Recall that solving SCCs in parity games is usually easier than solving games with more intricate topologies. For this reason, in this experiments we generated bigger games with number of vertices ranging from 5000 to 100000. Similarly to the 2-edges case described above, the clusters of generated games have

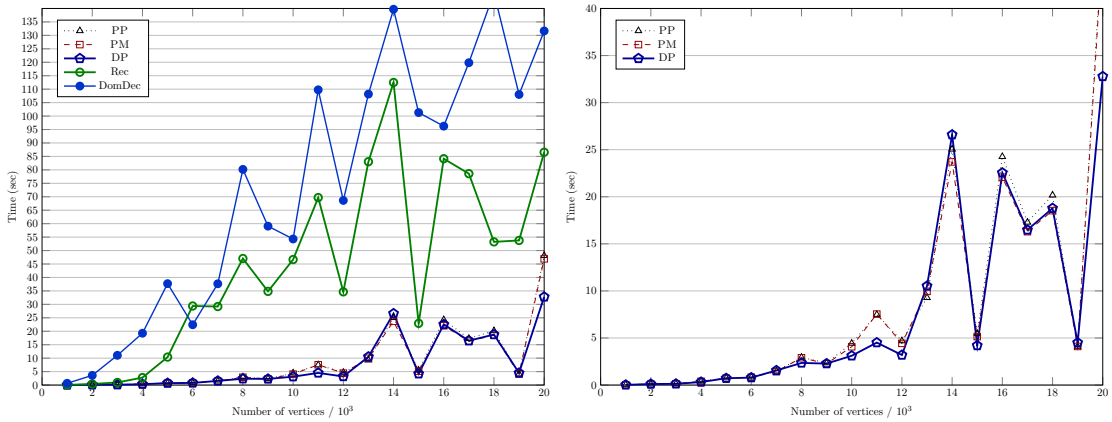


Figure 4.6: Performances on random games with 2 outgoing edges per node and 2/10 of priorities.

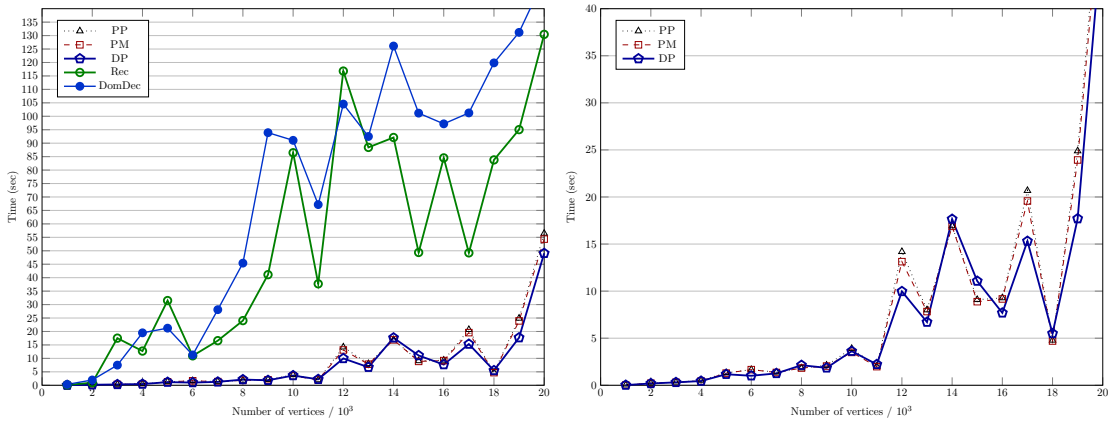


Figure 4.7: Performances on random games with 2 outgoing edges per node and 3/10 of priorities.

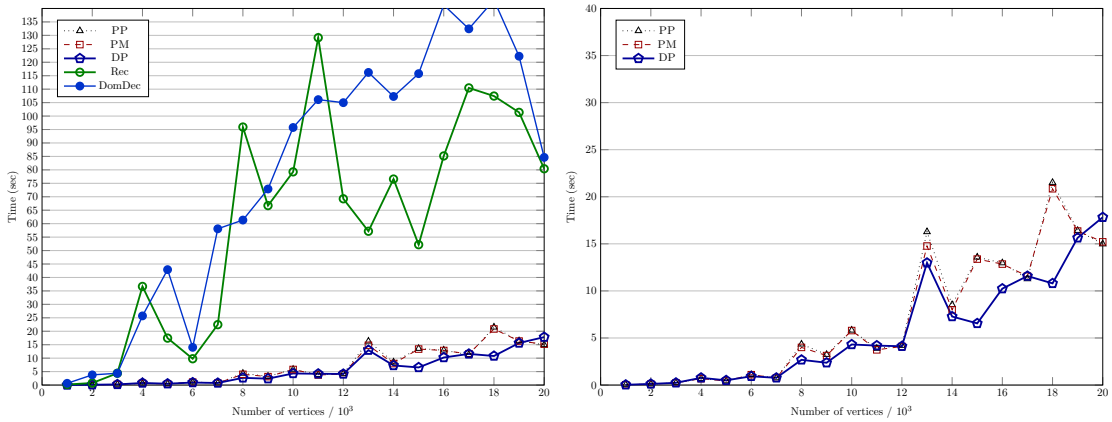


Figure 4.8: Performances on random games with 2 outgoing edges per node and 4/10 of priorities.

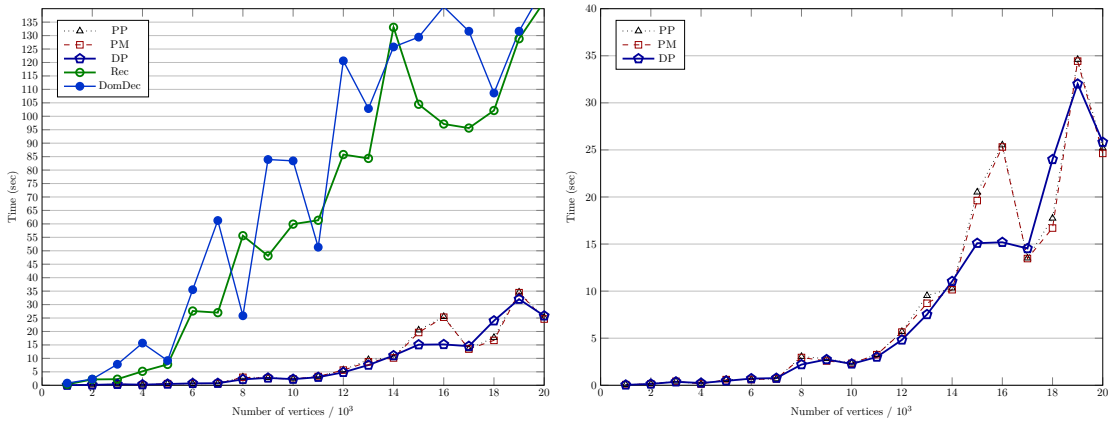


Figure 4.9: Performances on random games with 2 outgoing edges per node and 5/10 of priorities.

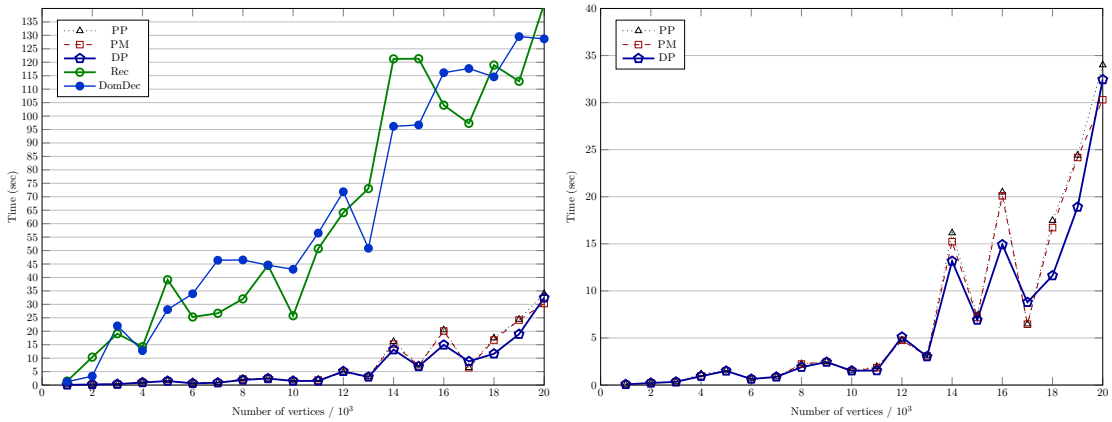


Figure 4.10: Performances on random games with 2 outgoing edges per node and 6/10 of priorities.

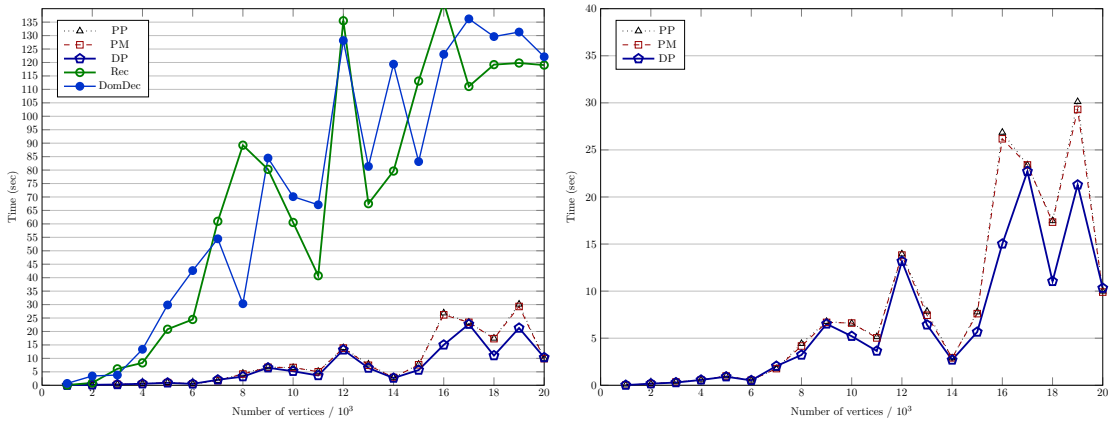


Figure 4.11: Performances on random games with 2 outgoing edges per node and 7/10 of priorities.

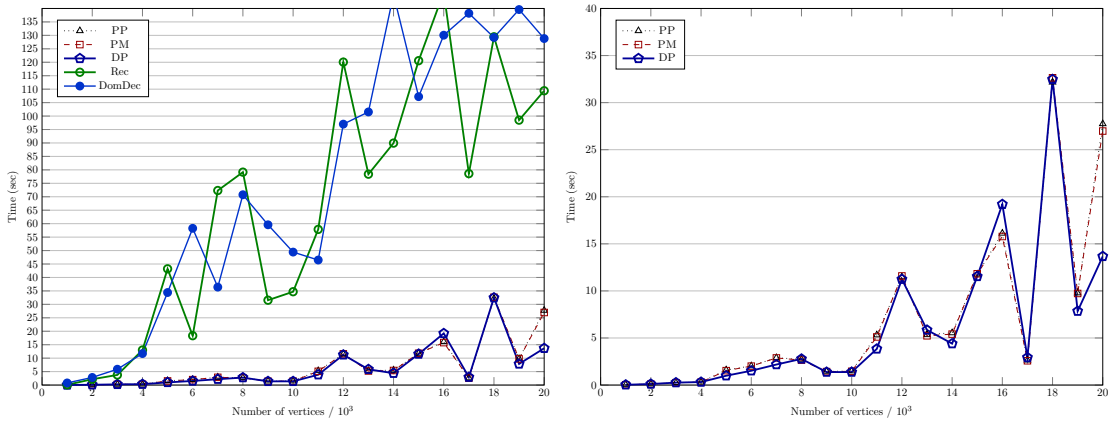


Figure 4.12: Performances on random games with 2 outgoing edges per node and 8/10 of priorities.

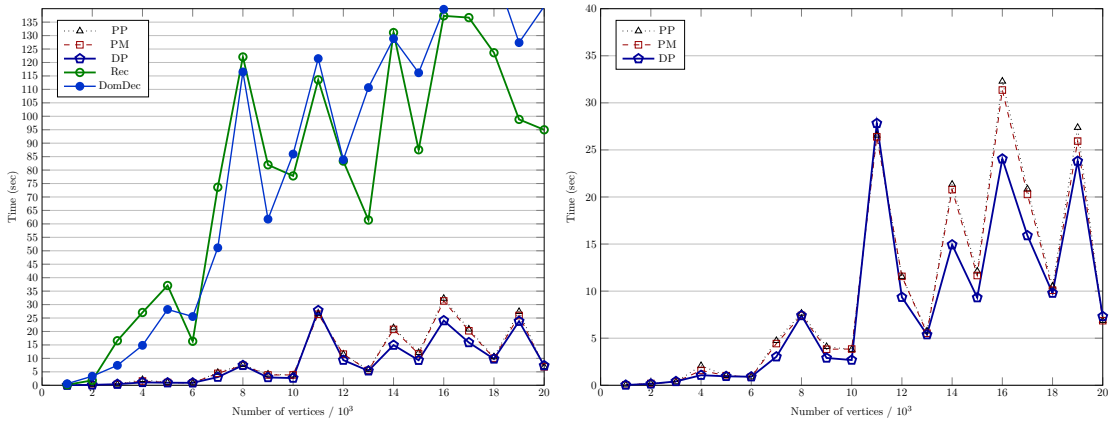


Figure 4.13: Performances on random games with 2 outgoing edges per node and 9/10 of priorities.

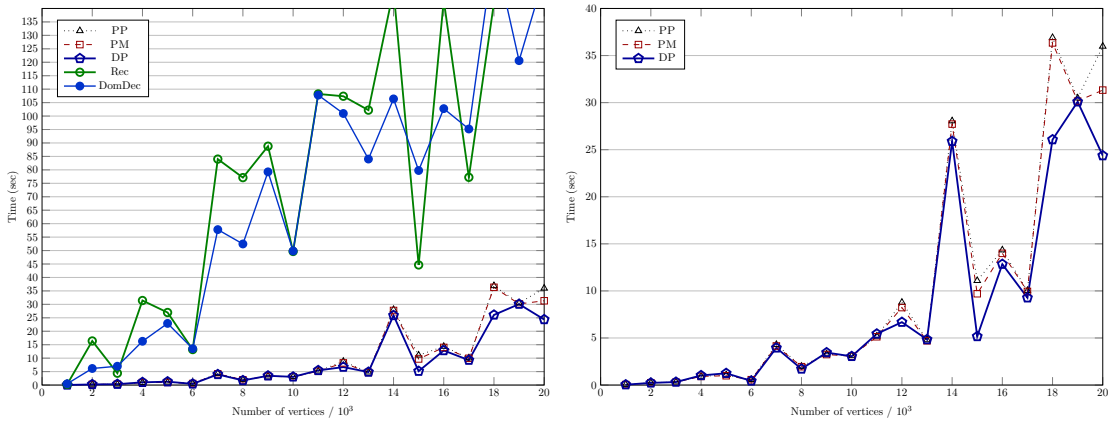


Figure 4.14: Performances on random games with 2 outgoing edges per node and 10/10 of priorities.

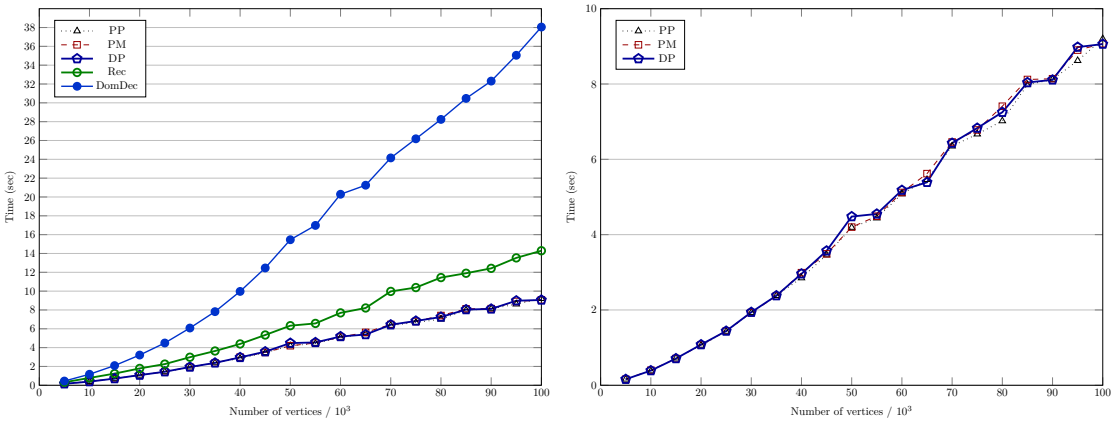


Figure 4.15: Performances on random games with 100 outgoing edges per node and 1/10 of priorities.

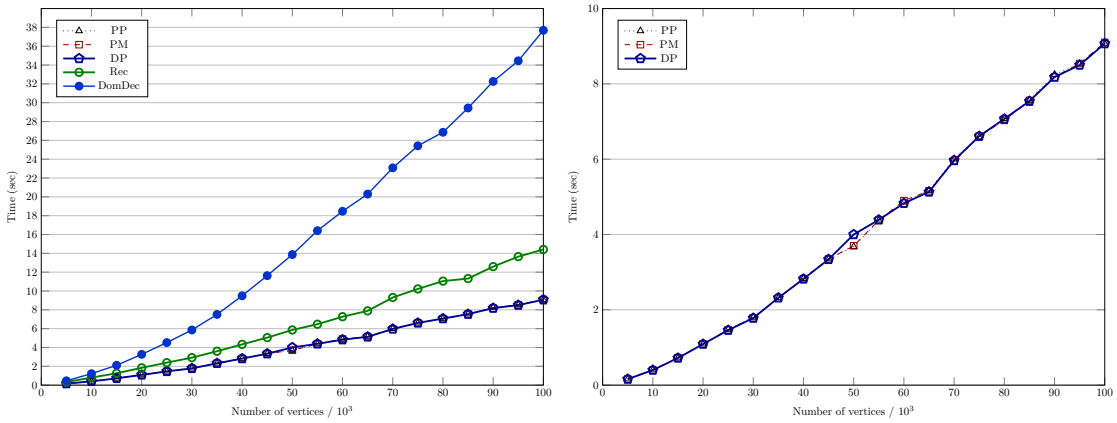


Figure 4.16: Performances on random games with 100 outgoing edges per node and 2/10 of priorities.

varying priorities ranging from $\frac{1}{10} \cdot n$ to n where n is the number of vertices.

Figures 4.15-4.24 depicts the running times of the best solvers on the resulting games. Even though the new algorithms perform better than *Rec* and *DomDec*, the differences in execution time are much lower than in the previous case, due to the fact that the games are very easy to solve. Increasing the number of priorities does not seem to significantly affect the performances either.

It is worth noticing that for none of the 2000 games in this category the new algorithms needed to perform any promotion, which is the main source of complexity of the solution technique. Indeed, when no promotions are performed, the new algorithms only need polynomial time to solve a game.

Logarithmic priorities As expected this class of random games is the easiest to solve among the ones considered in this thesis. The running times on the games with 2 outgoing edges per vertex (left-hand side of Figure 4.25) are very small. Therefore, the fact that *Rec* performs better than our algorithms is not significant. Since the other solvers have much worse running time they have been left out from the picture.

Differently from the previous classes of random games, when the number of priorities is logarithmic of the number of vertices increasing the number of edges makes the resulting games harder to solve. The right-hand side of Figure 4.25 shows the experimental result on games with 150 outgoing edges. Running times are 20 times higher than those for the 2-edges case. On these games *DomDec* and *Rec* performs in a very similar way, while the three new algorithms have slightly better running times than both.

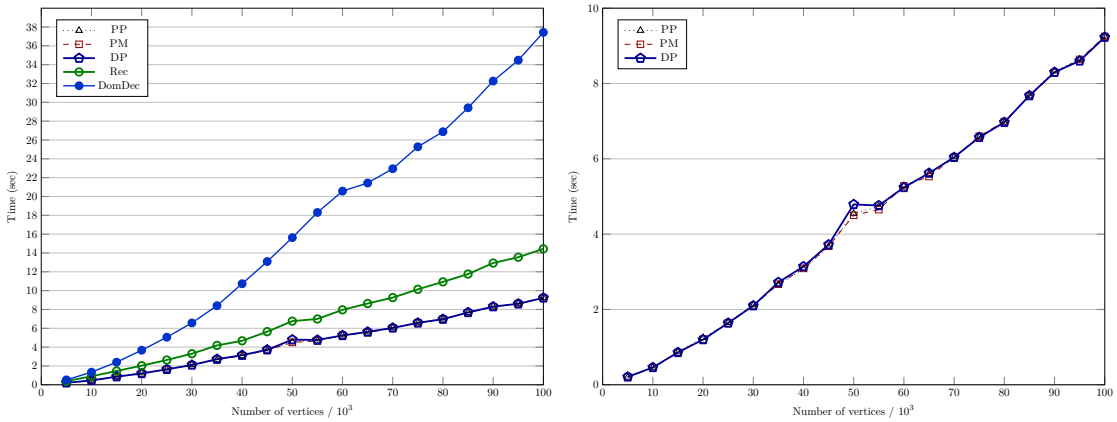


Figure 4.17: Performances on random games with 100 outgoing edges per node and 3/10 of priorities.

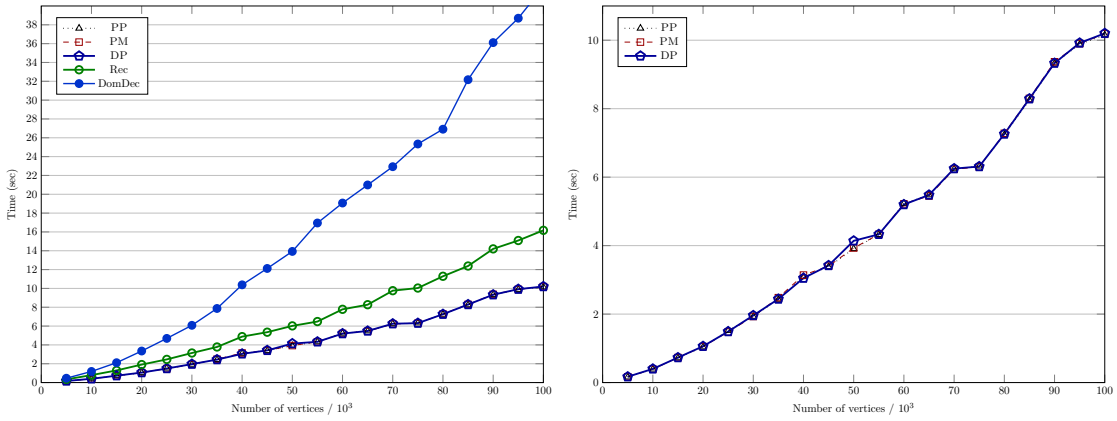


Figure 4.18: Performances on random games with 100 outgoing edges per node and 4/10 of priorities.

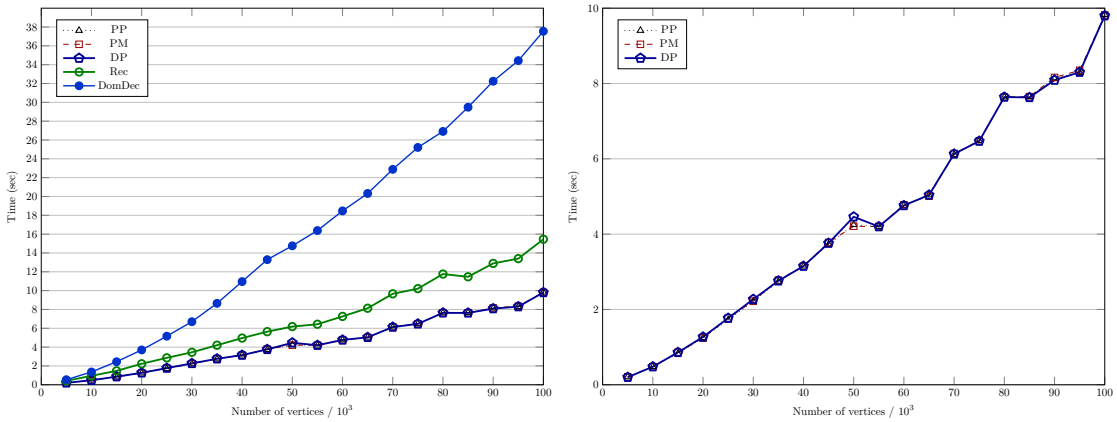


Figure 4.19: Performances on random games with 100 outgoing edges per node and 5/10 of priorities.

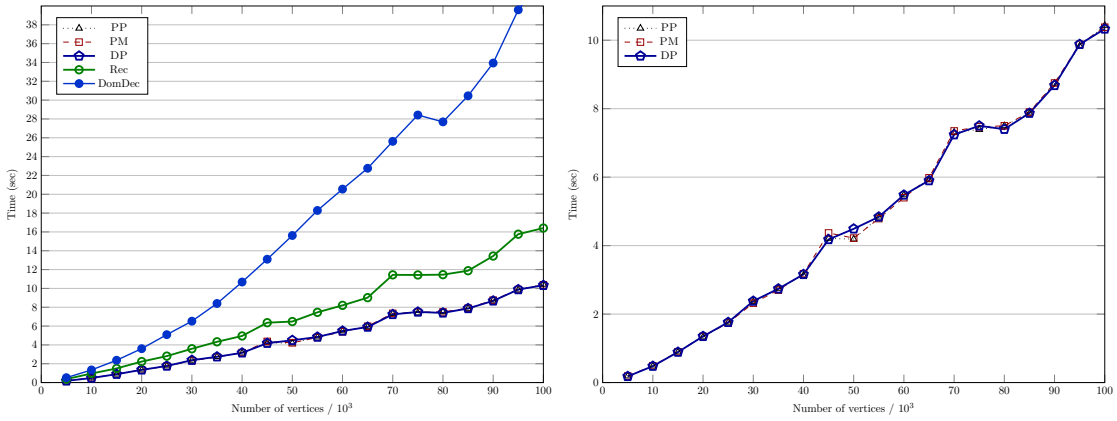


Figure 4.20: Performances on random games with 100 outgoing edges per node and 6/10 of priorities.

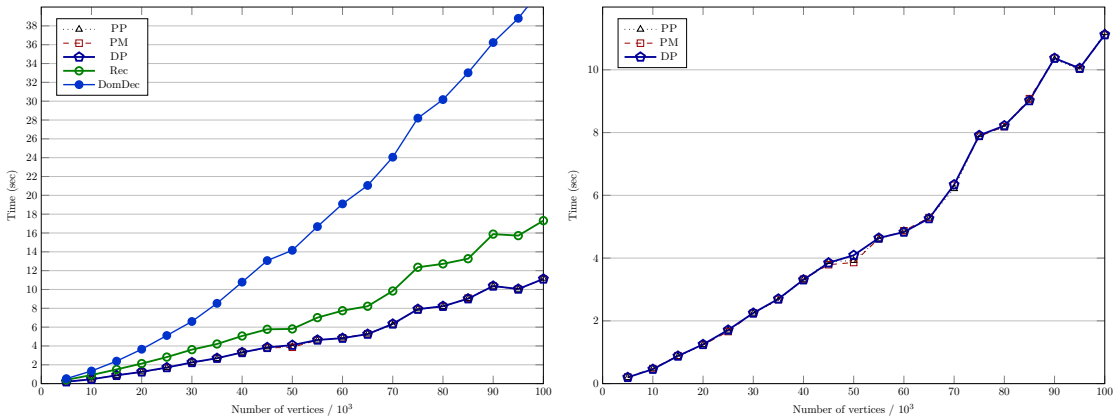


Figure 4.21: Performances on random games with 100 outgoing edges per node and 7/10 of priorities.

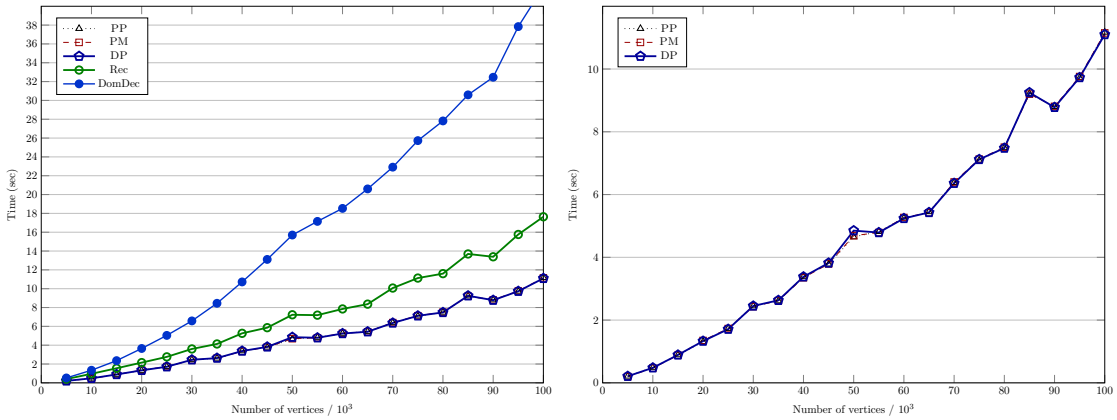


Figure 4.22: Performances on random games with 100 outgoing edges per node and 8/10 of priorities.

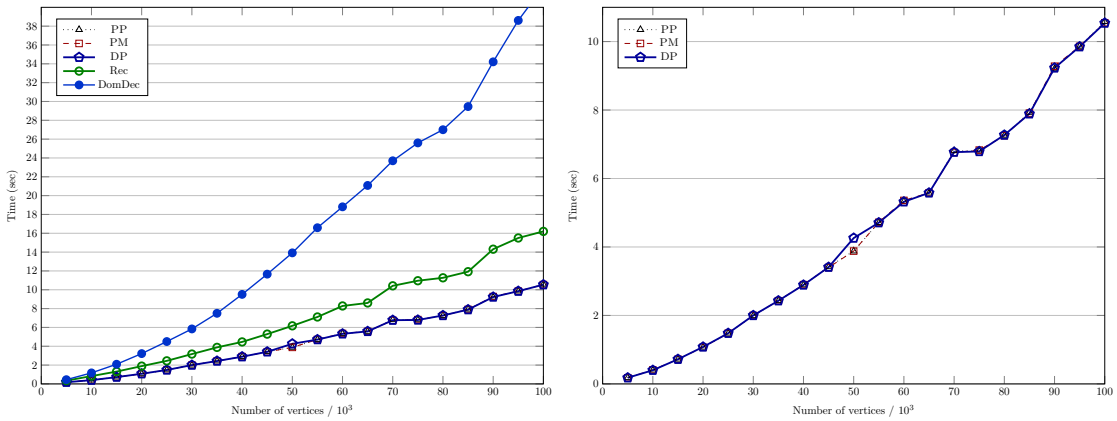


Figure 4.23: Performances on random games with 100 outgoing edges per node and 9/10 of priorities.

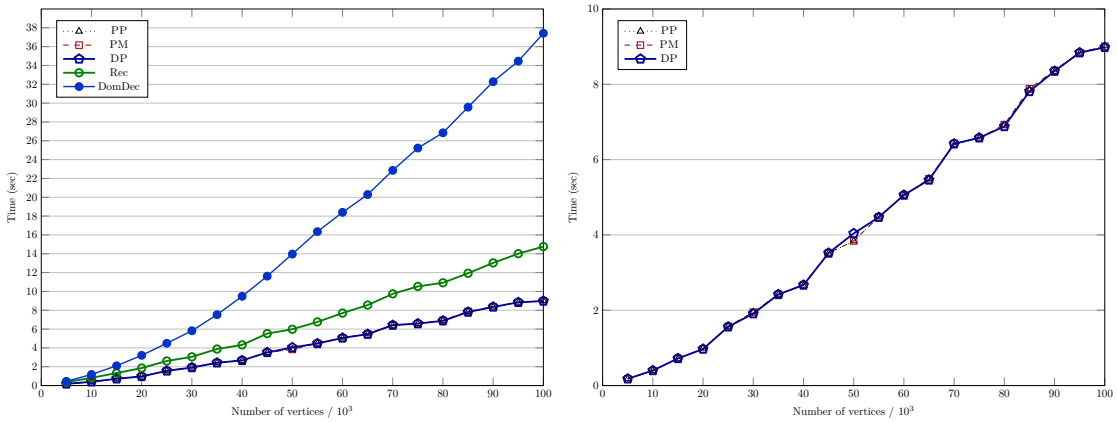


Figure 4.24: Performances on random games with 100 outgoing edges per node and 10/10 of priorities.

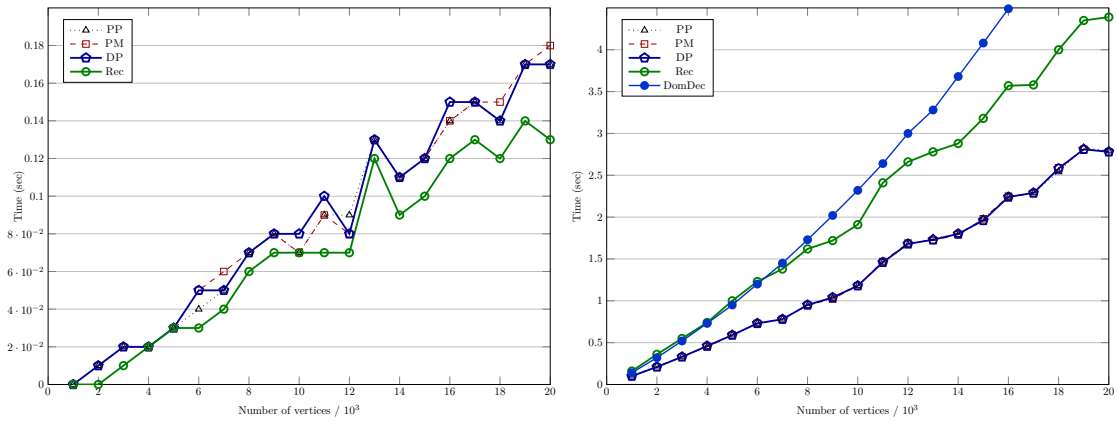


Figure 4.25: Performances on random games with 2 and 150 outgoing edges per node and logarithmic priorities.

Chapter 5

Conclusions

In this thesis we considered the problem of solving parity games, a special form of infinite games over graphs that have relevant applications in various fields of Theoretical Computer Science, most notably in Formal Verification. We proposed a new solving technique based on a Priority Promotion mechanism, which led to a new family of solvers. We developed three different algorithms following the new approach and provided the proofs of their soundness and completeness. We also carried out an accurate analysis of their space and time complexities. While no relevant improvement of the state-of-the-art was obtained on the time complexity, the new family exhibits the best space complexity among the current known algorithms for parity games. In fact, we proved that the maximal additional space needed to solve a parity game is linear in the number of vertices and logarithmic in the number of priorities, thus independent of the number of edges in the graph. This is an important result, in particular considering that practical applications in Formal Verification usually need to deal with the analysis of big computational systems, whose reformulation in terms of parity games results in very big graphs. Therefore, algorithms with low space requirements provide better scalability properties.

As far the time complexity is concerned, an exponential upper bound has been given. For the first two proposed solutions exponential lower bounds for the worst case were also presented in the form of two family of parity games on which the two algorithms exhibit an exponential behaviour. For the third solution algorithm, however, no lower bound is currently known. This leaves still open the possibility of a polynomial-time solution. Nonetheless, the actual time complexity of this algorithm is still an open problem.

The work of the thesis was not merely theoretical. Indeed we also provided an implementation of the three new solvers within the PGSolver framework. This allowed us to experimentally assess

the effectiveness of the new approach against concrete and synthetic problems and to compare their performance with that of the state-of-the-art solvers implemented in the same framework. The results show that the approach proposed in the thesis is extremely effective in practice, performing significantly better than existing algorithms most of the time.

The promising results obtained both at the theoretical and experimental levels suggest that this new approach is worth pursuing further. Therefore, future work include the extension of the family by devising new and clever Priority Promotion policies.

List of Figures

1.1 A coloured arena. 8
1.2 A coloured arena. 10
1.3 A parity game. 13

2.1 An explicative parity game for regions. 21
2.2 Running example. 27
2.3 The $\Im^{\mathrm{PP}}_{2,4}$ game. 36
2.4 The $\Im^{\mathrm{PM}}_{4}$ game. 43

3.1 The PGSolver architecture. 54
3.2 The Query function. 60
3.3 The IF case of Priority Promotion successor function. 61
3.4 The brp function. 62
3.5 The ELSE case of Priority Promotion successor function. 62

4.1 The Tower of Hanoi gadget. 64
4.2 Instance of Ladder Game 3. 68
4.3 Instance of Recursive Ladder Game 2. 69
4.4 Instance of Model Checker Ladder Game 2 70
4.5 Performances on random games with 2 outgoing edges per node and 1/10 of priorities. 72
4.6 Performances on random games with 2 outgoing edges per node and 2/10 of priorities. 73
4.7 Performances on random games with 2 outgoing edges per node and 3/10 of priorities. 73
4.8 Performances on random games with 2 outgoing edges per node and 4/10 of priorities. 74
4.9 Performances on random games with 2 outgoing edges per node and 5/10 of priorities. 74

4.10 Performances on random games with 2 outgoing edges per node and 6/10 of priorities. 75
4.11 Performances on random games with 2 outgoing edges per node and 7/10 of priorities. 75
4.12 Performances on random games with 2 outgoing edges per node and 8/10 of priorities. 76
4.13 Performances on random games with 2 outgoing edges per node and 9/10 of priorities. 76
4.14 Performances on random games with 2 outgoing edges per node and 10/10 of priorities. 77
4.15 Performances on random games with 100 outgoing edges per node and 1/10 of priorities. . . . 77
4.16 Performances on random games with 100 outgoing edges per node and 2/10 of priorities. . . . 78
4.17 Performances on random games with 100 outgoing edges per node and 3/10 of priorities. . . . 79
4.18 Performances on random games with 100 outgoing edges per node and 4/10 of priorities. . . . 79
4.19 Performances on random games with 100 outgoing edges per node and 5/10 of priorities. . . . 80
4.20 Performances on random games with 100 outgoing edges per node and 6/10 of priorities. . . . 80
4.21 Performances on random games with 100 outgoing edges per node and 7/10 of priorities. . . . 81
4.22 Performances on random games with 100 outgoing edges per node and 8/10 of priorities. . . . 81
4.23 Performances on random games with 100 outgoing edges per node and 9/10 of priorities. . . . 82
4.24 Performances on random games with 100 outgoing edges per node and 10/10 of priorities. . . . 82
4.25 Performances on random games with 2 and 150 outgoing edges per node and logarithmic priorities. 83

List of Tables

2.1 PP simulation. 27
2.2 PM simulation. 38
2.3 DP simulation. 45

4.1 Execution times in seconds for Tower of Hanoi games. Time out (†) is set to 180 seconds. 65
4.2 Execution times in seconds for Elevator games. Time out (†) is set to 180 seconds. . . . 66
4.3 Execution times in seconds for Language Inclusion games. Time out (†) is set to 180 seconds. 66
4.4 Execution times in seconds for Clique games. Time out (†) is set to 180 seconds. 67
4.5 Execution times in seconds for Ladder games. Time out (†) is set to 180 seconds. 68
4.6 Execution times in seconds for Recursive Ladder games. Time out (†) is set to 180 seconds. 69
4.7 Execution times in seconds for Model Checker Ladder games. Time out (†) is set to 180 seconds. 70
4.8 Execution times in seconds for Exponential Strategy Improvement games. Time out (†) is set to 180 seconds. 71

List of Algorithms

1 The Recursive Algorithm of Zielonka. 14

2 The Parity Game Solver. 20
3 The Winning Region Searcher. 23
4 The Monolithic Solver. 24
5 Query Function. 30
6 Successor Function. 31
7 Successor Function. 40
8 Successor Function. 48
9 Assignment Macro. 48
10 Delayed Promotion Macro. 49
11 Instant Promotion Macro. 49

12 The Attractor Operator. 57
13 The Subgame Function. 58

Bibliography

[AHK02] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-Time Temporal Logic. *JACM*, 49(5):672–713, 2002.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *AM*, 160(2):781–793, 2004.

[BG01] D. Berwanger and E. Grädel. Games and Model Checking for Guarded Logics. In *LPAR'01*, LNCS 2250, pages 70–84. Springer, 2001.

[BG04] D. Berwanger and E. Grädel. Fixed-Point Logics and Solitaire Games. *TCS*, 37(6):675–694, 2004.

[BMM13] M. Benerecetti, F. Mogavero, and A. Murano. Substructure Temporal Logic. In *LICS'13*, pages 368–377. IEEE Computer Society, 2013.

[BMM15] M. Benerecetti, F. Mogavero, and A. Murano. Reasoning About Substructures and Games. *TOCL*, 16(3):25:1–46, 2015.

[CHP10] K. Chatterjee, T.A. Henzinger, and N. Piterman. Strategy Logic. *IC*, 208(6):677–693, 2010.

[Con92] A. Condon. The Complexity of Stochastic Games. *IC*, 96(2):203–224, 1992.

[EJ91] E.A. Emerson and C.S. Jutla. Tree Automata, muCalculus, and Determinacy. In *FOCS'91*, pages 368–377. IEEE Computer Society, 1991.

[EJS93] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On Model Checking for the muCalculus and its Fragments. In *CAV'93*, LNCS 697, pages 385–396. Springer, 1993.

[EJS01] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On Model Checking for the muCalculus and its Fragments. *TCS*, 258(1-2):491–522, 2001.

[EM79] A. Ehrenfeucht and J. Mycielski. Positional Strategies for Mean Payoff Games. *IJGT*, 8(2), 1979.

[FK92a] M.R. Fellows and N. Koblitz. Self-Witnessing Polynomial-Time Complexity and Prime Factorization. In *CSCT'92*, pages 107–110. IEEE Computer Society, 1992.

[FK92b] M.R. Fellows and N. Koblitz. Self-Witnessing Polynomial-Time Complexity and Prime Factorization. *DCC*, 2(3):231–235, 1992.

[FL09] O. Friedmann and M. Lange. Solving Parity Games in Practice. In *ATVA'09*, LNCS 5799, pages 182–196. Springer, 2009.

[Fri09] Oliver Friedmann. An exponential lower bound for the parity game strategy improvement algorithm as we know it. In *LICS'09*, pages 145–156, 2009.

[GKK90] V.A. Gurvich, A.V. Karzanov, and L.G. Khachivan. Cyclic Games and an Algorithm to Find Minimax Cycle Means in Directed Graphs. *USSRCMMP*, 28(5):85–91, 1990.

[GTW02] E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games: A Guide to Current Research.* LNCS 2500. Springer, 2002.

[GZ05] H. Gimbert and W. Zielonka. Games where you can play optimally without any memory. In *CONCUR'05*, LNCS 3653, pages 428–442. Springer, 2005.

[JPZ06] M. Jurdziński, M. Paterson, and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. In *SODA'06*, pages 117–123. Association for Computing Machinery, 2006.

[JPZ08] M. Jurdziński, M. Paterson, and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SJM*, 38(4):1519–1532, 2008.

[Jur98] M. Jurdziński. Deciding the Winner in Parity Games is in UP ∩ co-Up. *IPL*, 68(3):119–124, 1998.

[Jur00] M. Jurdziński. Small Progress Measures for Solving Parity Games. In *STACS'00*, LNCS 1770, pages 290–301. Springer, 2000.

[KK91] N. Klarlund and D. Kozen. Rabin Measures and Their Applications to Fairness and Automata Theory. In *LICS'91*, pages 256–265. IEEE Computer Society, 1991.

[KV98] O. Kupferman and M.Y. Vardi. Weak Alternating Automata and Tree Automata Emptiness. In *STOC'98*, pages 224–233. Association for Computing Machinery, 1998.

[Mar75] A.D. Martin. Borel Determinacy. *AM*, 102(2):363–371, 1975.

[Mar85] A.D. Martin. A Purely Inductive Proof of Borel Determinacy. In *SPM'82*, Recursion Theory., pages 303–308. American Mathematical Society and Association for Symbolic Logic, 1985.

[McN93] R. McNaughton. Infinite Games Played on Finite Graphs. *APAL*, 65:149–184, 1993.

[MMPV12] F. Mogavero, A. Murano, G. Perelli, and M.Y. Vardi. What Makes ATL* Decidable? A Decidable Fragment of Strategy Logic. In *CONCUR'12*, LNCS 7454, pages 193–208. Springer, 2012.

[MMPV14] F. Mogavero, A. Murano, G. Perelli, and M.Y. Vardi. Reasoning About Strategies: On the Model-Checking Problem. *TOCL*, 15(4):34:1–42, 2014.

[MMV10] F. Mogavero, A. Murano, and M.Y. Vardi. Reasoning About Strategies. In *FSTTCS'10*, LIPIcs 8, pages 133–144. Leibniz-Zentrum fuer Informatik, 2010.

[Mos84] A.W. Mostowski. Regular Expressions for Infinite Trees and a Standard Form of Automata. In *SCT'84*, LNCS 208, pages 157–168. Springer, 1984.

[Mos91] A.W. Mostowski. Games with Forbidden Positions. Technical report, University of Gdańsk, Gdańsk, Poland, 1991.

[Sch07] S. Schewe. Solving Parity Games in Big Steps. In *FSTTCS'07*, LNCS 4855, pages 449–460. Springer, 2007.

[Sch08a] S. Schewe. An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games. In *CSL'08*, LNCS 5213, pages 369–384. Springer, 2008.

[Sch08b] S. Schewe. ATL* Satisfiability is 2ExpTime-Complete. In *ICALP'08*, LNCS 5126, pages 373–385. Springer, 2008.

[SF06] S. Schewe and B. Finkbeiner. Satisfiability and Finite Model Property for the Alternating-Time muCalculus. In *CSL'06*, pages 591–605. Springer, 2006.

[SS98] P Stevens and C Stirling. Practical model-checking using games. In *TACAS'98*, pages 85–101. Springer, 1998.

[Ver13] Maks Verver. Practical improvements to parity game solving. 2013.

[VJ00] J. Vöge and M. Jurdziński. A Discrete Strategy Improvement Algorithm for Solving Parity Games. In *CAV'00*, LNCS 1855, pages 202–215. Springer, 2000.

[Wil01] T. Wilke. Alternating Tree Automata, Parity Games, and Modal muCalculus. *BBMS*, 8(2):359–391, 2001.

[Zie98] W. Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *TCS*, 200(1-2):135–183, 1998.

[ZP96] U. Zwick and M. Paterson. The Complexity of Mean Payoff Games on Graphs. *TCS*, 158(1-2):343–359, 1996.

Appendix A

Source code

```
1
2  let not_none = function
3                  None -> false
4          |       _ -> true;;
5
6  let is_empty_bset bset length =
7          let i = ref 0 in
8          let empty = ref true in
9          while !i < length && !empty do
10           if BitSet.is_set bset !i
11           then empty := false
12           else incr i;
13         done;
14         !empty;;
15
16 let bit_dom_array_bound array length bound =
17         let dom = BitSet.create length in
18         for i = 0 to length - 1 do
19           if array.(i) > bound
20           then BitSet.set dom i
21         done;
22         dom;;
23
24 let max_prio game =
25         let n = ref (-1) in
26         for i = 0 to Array.length game - 1 do
```

```
27            let (p,_,_,_) = game.(i) in
28            if p > !n then n := p
29          done;
30          !n;;
31
32  let array_min array length bound =
33          let min = ref bound in
34          for i = 0 to length - 1 do
35            if array.(i) > (-1) && array.(i) < !min
36            then min := array.(i)
37          done;
38          if !min = bound then min := (-1);
39          !min;;
40
41  let collect_nodes_bit game length =
42          let bset = BitSet.create length in
43          for i = 0 to length - 1 do
44            let (pr, _, _, _) = game.(i) in
45            if (pr >= 0) then BitSet.set bset i
46          done;
47          bset;;
48
49  let pg_remove_nodes_bit game nodes nodes_en length =
50          Enum.iter (fun i -> game.(i) <- (-1,-1,[||],None)) nodes_en;
51          for v = 0 to length - 1 do
52            if not (BitSet.is_set nodes v)
53            then (
54              let (p,pl,ws,ann) = game.(v) in
55              game.(v) <- (p,pl,Array.of_list (List.filter (fun w -> let (p,_,_,_) = game
                    .(w) in
56                                                                    p <> -1)
57                                                  (Array.to_list ws)),ann);
58            )
59          done;;
60
61  let collect_nodes_by_prio game prio length =
62          let bset = BitSet.create length in
63          for i = 0 to length - 1 do
64            let (pr, _, _, _) = game.(i) in
65            if (pr = prio) then BitSet.set bset i
66          done;
```

```
67          bset;;
68
69  let pre_bit game player target length =
70          let pre = BitSet.create length in
71          for i = 0 to length - 1 do
72            let (_,pl,ws,_) = game.(i) in
73            let take = ref false in
74            if pl = player
75            then take := Array.fold_left (fun b -> fun w -> b || (BitSet.is_set target w))
                   false ws
76            else (
77              if pl != (-1)
78              then (
79                take := true;
80                let j = ref 0 in
81                let ws_len = Array.length ws in
82                while !take && (!j < ws_len) do
83                  if not (BitSet.is_set target ws.(!j))
84                    then take := false
85                    else incr j;
86                done;
87              );
88            );
89            if !take then BitSet.set pre i;
90          done;
91          pre;;
92
93  let attr_bit game strategy player region region_en =
94    let attr = BitSet.copy region in
95    let todoQ = Queue.create () in
96    let todoS = Queue.create () in
97    let used = BitSet.copy region in
98    let tgraph = game_to_transposed_graph game in
99    Enum.iter (fun v -> List.iter (fun w -> if not (BitSet.is_set used w)
100                                            then (
101                                              let (_,pl,_,_) = game.(w) in
102                                              BitSet.set used w;
103                                              if pl<>player
104                                              then Queue.add w todoQ
105                                              else (Queue.add w todoS; strategy.(w) <- v
                                                   ;)
```

```
106                                          );
107                                    ) tgraph.(v)) region_en;
108   while not (Queue.is_empty todoS && Queue.is_empty todoQ) do
109     if not (Queue.is_empty todoS)
110     then(
111       let v = Queue.take todoS in
112       BitSet.set attr v;
113       List.iter (fun w -> if not (BitSet.is_set used w)
114                           then (
115                           let (_,pl,_,_) = game.(w) in
116                           BitSet.set used w;
117                           if pl<>player
118                           then Queue.add w todoQ
119                           else (Queue.add w todoS; strategy.(w) <- v;)
120                           );
121                 ) tgraph.(v);
122     )
123     else (
124       let v = Queue.take todoQ in
125         let (_,_,ws,_) = game.(v) in
126         if Array.fold_left (fun b -> fun w -> b && (BitSet.is_set attr w)) true ws
127         then (
128           BitSet.set attr v;
129           List.iter (fun w -> if not (BitSet.is_set used w)
130                           then (
131                             let (_,pl,_,_) = game.(w) in
132                             BitSet.set used w;
133                             if pl<>player
134                             then Queue.add w todoQ
135                             else (Queue.add w todoS; strategy.(w) <- v;)
136                             );
137                  ) tgraph.(v);
138         )
139         else BitSet.unset used v;
140     );
141   done;
142 attr;;
143
144 let attr_bit_c game tgraph strategy player region region_en =
145   let attr = BitSet.copy region in
146   let todoQ = Queue.create () in
```

```
147   let todoS = Queue.create () in
148   let used = BitSet.copy region in
149   Enum.iter (fun v -> List.iter (fun w -> if not (BitSet.is_set used w)
150                                          then (
151                                             let (_,pl,_,_) = game.(w) in
152                                             BitSet.set used w;
153                                             if pl<>player
154                                             then Queue.add w todoQ
155                                             else (Queue.add w todoS; strategy.(w) <- v
                                                    ;)
156                                          );
157                                ) tgraph.(v)) region_en;
158   while not (Queue.is_empty todoS && Queue.is_empty todoQ) do
159     if not (Queue.is_empty todoS)
160     then(
161       let v = Queue.take todoS in
162       BitSet.set attr v;
163       List.iter (fun w -> if not (BitSet.is_set used w)
164                           then (
165                           let (_,pl,_,_) = game.(w) in
166                           BitSet.set used w;
167                           if pl<>player
168                           then Queue.add w todoQ
169                           else (Queue.add w todoS; strategy.(w) <- v;)
170                           );
171                 ) tgraph.(v);
172     )
173     else (
174       let v = Queue.take todoQ in
175         let (_,_,ws,_) = game.(v) in
176         if Array.fold_left (fun b -> fun w -> b && (BitSet.is_set attr w)) true ws
177         then (
178           BitSet.set attr v;
179           List.iter (fun w -> if not (BitSet.is_set used w)
180                           then (
181                             let (_,pl,_,_) = game.(w) in
182                             BitSet.set used w;
183                             if pl<>player
184                             then Queue.add w todoQ
185                             else (Queue.add w todoS; strategy.(w) <- v;)
186                             );
```

```
187                     ) tgraph.(v);
188           )
189           else BitSet.unset used v;
190       );
191     done;
192   attr;;
193
194   let solver1_iter game =
195           let l = Array.length game in
196           let solution = Array.make l (-1) in
197           let str = Array.make l (-1) in
198           let str_out = Array.make l (-1) in
199           let max_col = 1 + max_prio game in
200           let r = Array.make l (-1) in
201           let tgraph_full = game_to_transposed_graph game in
202           let is_up = ref false in
203           let ric = ref 0 in
204           let op = ref 0 in
205           let up = ref 0 in
206           let iterate = ref true in
207           while !iterate do
208             let strategy = Array.make l (-1) in
209             let vr = collect_nodes_bit game l in
210             let m = array_min r l max_col in
211             let i = ref 0 in
212             let empty = ref true in
213             while !i < l && !empty do
214               let (pr,_,_,_) = game.(!i) in
215               if pr <> -1
216               then empty := false
217               else incr i;
218             done;
219             if not !empty
220             then (
221               let r_dom =
222                 if !is_up then bit_dom_array_bound r l m
223                 else bit_dom_array_bound r l (-1) in
224                  let game' = Array.copy game in
225                   let r_dom_en = BitSet.enum r_dom in
226                    pg_remove_nodes_bit game' r_dom
227                                        (Enum.clone r_dom_en) l;
```

```
228                 let n = max_prio game' in
229                  let target = if !is_up
230                    then (let r_m = BitSet.create l in
231                    for i = 0 to l - 1 do
232                      if r.(i) = m
233                      then BitSet.set r_m i
234                    done;
235                   r_m)
236                    else collect_nodes_by_prio game' n l in
237                 let player = n mod 2 in
238                 let tar_en = BitSet.enum target in
239                 let a = attr_bit game' strategy player target
240                       (Enum.clone tar_en) in
241               let a_en = BitSet.enum a in
242               Enum.iter (fun v -> BitSet.unset vr v) (Enum.clone r_dom_en);
243               let p = pre_bit game' (1 - player) (BitSet.diff vr a) l in
244               Enum.iter (fun i -> let (_,pl,ws,_) = game.(i) in
245                                   if pl = player
246                                   then (
247                                     if str.(i) = -1
248                                     then (
249                                       let w = Array.fold_left (fun b -> fun w -> if (b >
                                              -1 || not (BitSet.is_set a w)) then b else w)
                                              (-1) ws in
250                                       if w > -1 then strategy.(i) <- w;
251                                     )
252                                     else strategy.(i) <- str.(i);
253                                   )
254                         ) tar_en;
255               if not (is_empty_bset (BitSet.inter p target) l)
256               then (
257                 Enum.iter (fun i -> r.(i) <- n;) a_en;
258                 is_up := false;
259                 for i = 0 to l - 1 do
260                           if strategy.(i) <> -1
261                           then str.(i) <- strategy.(i);
262                 done;
263                 incr ric;
264                 incr op;
265               )
266               else (
```

```
267                let r_pl = BitSet.create l in
268  if player = 0 then Enum.iter (fun i -> if r.(i) mod 2 = 0
269                                         then BitSet.set r_pl i)
270                                 r_dom_en
271  else Enum.iter (fun i -> if r.(i) mod 2 = 1
272                           then BitSet.set r_pl i)
273                 r_dom_en;
274  let min = ref (max_col) in
275   Enum.iter (fun v -> let (_,_,ws,_) = game.(v) in
276     let color = ref (max_col) in
277     Array.iter (fun w -> if BitSet.is_set r_pl w
278                          then (if r.(w) < !color
279                                then (
280                                color := r.(w);
281                                if r.(w) < !min
282                                then min := r.(w); )))
283                 ws;)
284    (Enum.clone a_en);
285                if !min <> (max_col)
286                then (
287                  for i = 0 to l - 1 do
288                    if BitSet.is_set a i
289                    then r.(i) <- !min
290                    else (
291                      if r.(i) < !min
292                      then (
293                        str.(i) <- -1;
294                        r.(i) <- -1;
295                      );
296                    );
297                    if strategy.(i) <> -1
298                    then str.(i) <- strategy.(i);
299                  done;
300                  incr ric;
301                  incr op;
302                  incr up;
303                  is_up := true;
304                )
305                else (
306                  let a' = if BitSet.count a = pg_node_count game
307                  then a
```

```
308                 else attr_bit_c game tgraph_full strategy player a a_en in
309                 let en_a = BitSet.enum a' in
310                 pg_remove_nodes_bit game a' (Enum.clone en_a) l;
311                 let new_tgraph = game_to_transposed_graph game in
312                 for i = 0 to l - 1 do
313                   r.(i) <- (-1);
314                   tgraph_full.(i) <- new_tgraph.(i);
315                 done;
316                 incr ric;
317                 op := !op + 2;
318                 is_up := false;
319                 if player = 0
320                 then (
321                   Enum.iter (fun i -> solution.(i) <- 0;) en_a;
322                   for i = 0 to l - 1 do
323                     if strategy.(i) <> -1
324                     then str_out.(i) <- strategy.(i)
325                   done;
326                 )
327                 else (
328                   Enum.iter (fun i -> solution.(i) <- 1;) en_a;
329                   for i = 0 to l - 1 do
330                     if strategy.(i) <> -1
331                     then str_out.(i) <- strategy.(i)
332                   done;
333                 );
334                 for i = 0 to l - 1 do
335                   str.(i) <- -1;
336                 done;
337               )
338             )
339           )
340           else (
341             iterate := false;
342           );
343         done;
344         (solution,str_out,!ric,!op,!up);;
345
346 let pla1iter_main game =
347         let l = Array.length game in
348         let game' = Array.copy game in
```

```
349         let (solution,strategy,iteration,operation,upgrade) = solver1_iter game' in
350         print_string "\n";
351         Printf.printf "Recursions: %d, " iteration;
352         Printf.printf "Attractors: %d, " operation;
353         Printf.printf "Upgrades: %d\n" upgrade;
354         (solution,strategy);;
355
356
357 let solve = p1a1iter_main;;
358
359 register_solver solve "parity1a1" "p1a1" "use the new Iterative Parity Art1Alg1
        procedure";;
360
361 let solver2_iter game =
362         let l = Array.length game in
363         let solution = Array.make l (-1) in
364         let str = Array.make l (-1) in
365         let str_out = Array.make l (-1) in
366         let max_col = 1 + max_prio game in
367         let r = Array.make l (-1) in
368         let tgraph_full = game_to_transposed_graph game in
369         let is_up = ref false in
370         let ric = ref 0 in
371         let op = ref 0 in
372         let up = ref 0 in
373         let iterate = ref true in
374         let m = ref 0 in
375         while !iterate do
376           let strategy = Array.make l (-1) in
377           let vr = collect_nodes_bit game l in
378           let i = ref 0 in
379           let empty = ref true in
380           while !i < l && !empty do
381             let (pr,_,_,_) = game.(!i) in
382             if pr <> -1
383             then empty := false
384             else incr i;
385           done;
386           if not !empty
387           then (
388             let r_dom = if !is_up
```

```
389             then bit_dom_array_bound r l !m
390             else bit_dom_array_bound r l (!m - 1) in
391             let game' = Array.copy game in
392             let r_dom_en = BitSet.enum r_dom in
393             pg_remove_nodes_bit game' r_dom (Enum.clone r_dom_en) l;
394             let (n,target) = if !is_up
395             then (
396               let r_m = BitSet.create l in
397               for i = 0 to l - 1 do
398                 if r.(i) = !m
399                 then BitSet.set r_m i
400               done;
401               (!m,r_m)
402             )
403             else (
404               let max_r = ref (-1) in
405               for i = 0 to l - 1 do
406                 if r.(i) > !max_r && r.(i) < !m
407                 then max_r := r.(i);
408               done;
409               let n_g = max_prio game' in
410               if !max_r >= n_g
411               then (
412                 let r_n = BitSet.create l in
413                 for i = 0 to l - 1 do
414                   if r.(i) = !max_r
415                   then BitSet.set r_n i
416                 done;
417                 (!max_r,r_n)
418               )
419               else (n_g,collect_nodes_by_prio game' n_g l)
420             ) in
421             let player = n mod 2 in
422             let tar_en = BitSet.enum target in
423             let a = attr_bit game' strategy player target (Enum.clone tar_en) in
424             let a_en = BitSet.enum a in
425             Enum.iter (fun v -> BitSet.unset vr v) (Enum.clone r_dom_en);
426             let p = pre_bit game' (1 - player) (BitSet.diff vr a) l in
427             Enum.iter (fun i -> let (_,pl,ws,_) = game.(i) in
428                                 if pl = player
429                                 then (
```

```
430                                 if str.(i) = -1
431                                 then (
432                                   let w = Array.fold_left (fun b -> fun w -> if (b >
                                          -1 || not (BitSet.is_set a w)) then b else w)
                                          (-1) ws in
433                                   if w > -1 then strategy.(i) <- w;
434                                 )
435                                 else strategy.(i) <- str.(i);
436                               )
437                     ) tar_en;
438             if not (is_empty_bset (BitSet.inter p target) l)
439             then (
440               Enum.iter (fun i -> r.(i) <- n;) a_en;
441               is_up := false;
442               m := n;
443               for i = 0 to l - 1 do
444                 if strategy.(i) <> -1
445                 then str.(i) <- strategy.(i);
446               done;
447               incr ric;
448               incr op;
449             )
450             else (
451               let r_pl = BitSet.create l in
452               if player = 0
453               then Enum.iter (fun i -> if r.(i) mod 2 = 0 then BitSet.set r_pl i)
                      r_dom_en
454               else Enum.iter (fun i -> if r.(i) mod 2 = 1 then BitSet.set r_pl i)
                      r_dom_en;
455               let min = ref (max_col) in
456               Enum.iter (fun v -> let (_,_,ws,_) = game.(v) in
457                                   let color = ref (max_col) in
458                                   Array.iter (fun w -> if BitSet.is_set r_pl w
459                                                        then (
460                                                          if r.(w) < !color
461                                                          then (
462                                                            color := r.(w);
463                                                            if r.(w) < !min
464                                                            then min := r.(w);
465                                                          )
466                                                        )
```

```
467                                         ) ws;
468                         ) (Enum.clone a_en);
469                 if !min <> (max_col)
470                 then (
471                   for i = 0 to l - 1 do
472                     if BitSet.is_set a i
473                     then r.(i) <- !min
474                     else (
475                       if r.(i) < !min && (r.(i) mod 2 <> player)
476                       then (
477                         str.(i) <- -1;
478                         r.(i) <- -1;
479                       );
480                     );
481                     if strategy.(i) <> -1
482                     then str.(i) <- strategy.(i);
483                   done;
484                   incr ric;
485                   incr op;
486                   incr up;
487                   is_up := true;
488                   m := !min;
489                 )
490                 else (
491                   let a' = if BitSet.count a = pg_node_count game
492                   then a
493                   else attr_bit_c game tgraph_full strategy player a a_en in
494                   let en_a = BitSet.enum a' in
495                   pg_remove_nodes_bit game a' (Enum.clone en_a) l;
496                   let new_tgraph = game_to_transposed_graph game in
497                   for i = 0 to l - 1 do
498                     r.(i) <- (-1);
499                     tgraph_full.(i) <- new_tgraph.(i);
500                   done;
501                   incr ric;
502                   op := !op + 2;
503                   is_up := false;
504                   m := 0;
505                   if player = 0
506                   then (
507                     Enum.iter (fun i -> solution.(i) <- 0;) en_a;
```

```
508                   for i = 0 to l - 1 do
509                     if strategy.(i) <> -1
510                     then str_out.(i) <- strategy.(i)
511                   done;
512                 )
513                 else (
514                   Enum.iter (fun i -> solution.(i) <- 1;) en_a;
515                   for i = 0 to l - 1 do
516                     if strategy.(i) <> -1
517                     then str_out.(i) <- strategy.(i)
518                   done;
519                 );
520                 for i = 0 to l - 1 do
521                   str.(i) <- -1;
522                 done;
523               )
524             )
525           )
526           else iterate := false;
527         done;
528         (solution,str_out,!ric,!op,!up);;
529
530 let p1a2iter_main game =
531         let l = Array.length game in
532         let game' = Array.copy game in
533         let (solution,strategy,iteration,operation,upgrade) = solver2_iter game' in
534         print_string "\n";
535         Printf.printf "Recursions: %d, " iteration;
536         Printf.printf "Attractors: %d, " operation;
537         Printf.printf "Upgrades: %d, " upgrade;
538         (solution,strategy);;
539
540
541 let solve = p1a2iter_main;;
542
543 register_solver solve "parity1a2" "p1a2" "use the new Iterative Parity Art1Alg2
        procedure";;
544
545 let solver3_iter game =
546         let l = Array.length game in
547         let solution = Array.make l (-1) in
```

```
548         let str = Array.make l (-1) in
549         let str_out = Array.make l (-1) in
550         let max_col = 1 + max_prio game in
551         let r = Array.make l (-1) in
552         let tgraph_full = game_to_transposed_graph game in
553         let is_up = ref false in
554         let ric = ref 0 in
555         let op = ref 0 in
556         let up = ref 0 in
557         let iterate = ref true in
558         let m = ref 0 in
559         let max_f = ref 0 in
560         let even_set = ref Base.FSet.empty in
561         let odd_set = ref Base.FSet.empty in
562         let even_f = ref Base.FMap.empty in
563         let odd_f = ref Base.FMap.empty in
564         let f = Array.make l (-1) in
565         while !iterate do
566           let strategy = Array.make l (-1) in
567           let vr = collect_nodes_bit game l in
568           if not (is_empty_game game l)
569           then (
570             let r_dom = if !is_up
571             then bit_dom_array_bound r l !m
572             else bit_dom_array_bound r l (!m - 1) in
573             let game' = Array.copy game in
574             let r_dom_en = BitSet.enum r_dom in
575             pg_remove_nodes_bit game' r_dom (Enum.clone r_dom_en) l;
576             if is_empty_game game' l
577             then (
578               let (dual_pl,pl_f) = if !max_f mod 2 = 0
579               then (1,even_f)
580               else (0,odd_f) in
581               for i = 0 to l - 1 do
582                 if r.(i) < !max_f && r.(i) mod 2 = dual_pl
583                 then (
584                   str.(i) <- -1;
585                   r.(i) <- -1;
586                   f.(i) <- -1;
587                 )
588                 else r.(i) <- f.(i);
```

```
589                 done;
590                 m := !max_f;
591                 is_up := true;
592                 even_set := Base.FSet.empty;
593                 odd_set := Base.FSet.empty;
594                 even_f := Base.FMap.empty;
595                 odd_f := Base.FMap.empty;
596                 max_f := 0;
597                 incr ric;
598                 incr up;
599               )
600               else (
601                 let (n,target) = if !is_up
602                 then (
603                   let r_m = BitSet.create l in
604                   for i = 0 to l - 1 do
605                     if r.(i) = !m
606                     then BitSet.set r_m i
607                   done;
608                   (!m,r_m)
609                 )
610                 else (
611                   let max_r = ref (-1) in
612                   for i = 0 to l - 1 do
613                     if r.(i) > !max_r && r.(i) < !m
614                     then max_r := r.(i);
615                   done;
616                   let n_g = max_prio game' in
617                   if !max_r >= n_g
618                   then (
619                     let r_n = BitSet.create l in
620                     for i = 0 to l - 1 do
621                       if r.(i) = !max_r
622                       then BitSet.set r_n i
623                     done;
624                     (!max_r,r_n)
625                   )
626                   else (n_g,collect_nodes_by_prio game' n_g l)
627                 ) in
628                 let player = n mod 2 in
629                 let tar_en = BitSet.enum target in
```

```
630             let a = attr_bit game' strategy player target (Enum.clone tar_en) in
631             let a_en = BitSet.enum a in
632             Enum.iter (fun v -> BitSet.unset vr v) (Enum.clone r_dom_en);
633             let p = pre_bit game' (1 - player) (BitSet.diff vr a) l in
634             Enum.iter (fun i -> let (_,pl,ws,_) = game.(i) in
635                                 if pl = player
636                                 then (
637                                   if str.(i) = -1
638                                   then (
639                                     let w = Array.fold_left (fun b -> fun w -> if (b >
                                             -1 || not (BitSet.is_set a w)) then b else w)
                                             (-1) ws in
640                                     if w > -1 then strategy.(i) <- w;
641                                   )
642                                   else strategy.(i) <- str.(i);
643                                 )
644                       ) tar_en;
645             if not (is_empty_bset (BitSet.inter p target) l)
646             then (
647                       Enum.iter (fun i -> r.(i) <- n; f.(i) <- n;) a_en;
648                       if !is_up
649                       then (
650                         if player = 0
651                         then even_set := Base.FSet.add n !even_set
652                         else odd_set := Base.FSet.add n !odd_set;
653                       );
654                       is_up := false;
655                       m := n;
656                       for i = 0 to l - 1 do
657                         if strategy.(i) <> -1
658                         then str.(i) <- strategy.(i);
659                       done;
660                       incr ric;
661                       incr op;
662             )
663             else (
664               let r_pl = BitSet.create l in
665               if player = 0
666               then Enum.iter (fun i -> if r.(i) mod 2 = 0 then BitSet.set r_pl i)
                       r_dom_en
667               else Enum.iter (fun i -> if r.(i) mod 2 = 1 then BitSet.set r_pl i)
```

```
                    r_dom_en;
668                 let min = ref (max_col) in
669                 Enum.iter (fun v -> let (_,_,ws,_) = game.(v) in
670                                     let color = ref (max_col) in
671                                     Array.iter (fun w -> if BitSet.is_set r_pl w
672                                                          then (
673                                                            color := f.(w);
674                                                            if !color < !min
675                                                            then min := !color;
676                                                          );
677                                                ) ws;
678                           ) (Enum.clone a_en);
679                 if !min <> (max_col)
680                 then (
681                   let check = ref false in
682                   let over = if player = 0
683                   then Base.FSet.min_elt !odd_set
684                   else Base.FSet.min_elt !even_set in
685                   if over < Some !min && not_none over
686                   then check := true;
687                   if not !check && (not (Base.FMap.is_empty !even_f) || not (Base.FMap.
                        is_empty !odd_f))
688                   then (
689                     Base.FMap.iter (fun a b -> if not !check && !min >= a && !min <= b
                          then check := true) !even_f;
690                     Base.FMap.iter (fun a b -> if not !check && !min >= a && !min <= b
                          then check := true) !odd_f;
691                   );
692                   if !check
693                   then (
694                     for i = 0 to l - 1 do
695                       if BitSet.is_set a i
696                       then (
697                         r.(i) <- n;
698                         f.(i) <- !min;
699                       );
700                       if strategy.(i) <> -1
701                       then str.(i) <- strategy.(i);
702                     done;
703                     if n mod 2 = 0
704                     then even_f := Base.FMap.add n !min !even_f
```

```
705                     else odd_f := Base.FMap.add n !min !odd_f;
706                     if !max_f < !min
707                     then max_f := !min;
708                     incr ric;
709                     incr op;
710                     is_up := false;
711                     m := n;
712                   )
713                   else (
714                     for i = 0 to l - 1 do
715                       if BitSet.is_set a i
716                       then (
717                         r.(i) <- !min;
718                         f.(i) <- !min;
719                       )
720                       else (
721                         if r.(i) < !min && r.(i) mod 2 <> player
722                         then (
723                           str.(i) <- -1;
724                           r.(i) <- -1;
725                           f.(i) <- -1;
726                         );
727                       );
728                       if strategy.(i) <> -1
729                       then str.(i) <- strategy.(i);
730                     done;
731                     if player = 0
732                     then even_set := Base.FSet.remove !min !even_set
733                     else odd_set := Base.FSet.remove !min !odd_set;
734                     if !max_f < !min
735                     then max_f := 0;
736                     incr ric;
737                     incr op;
738                     incr up;
739                     is_up := true;
740                     m := !min;
741                   );
742                 )
743                 else (
744                   let a' = if BitSet.count a = pg_node_count game
745                   then a
```

```
746                 else attr_bit_c game tgraph_full strategy player a a_en in
747                 let en_a = BitSet.enum a' in
748                 pg_remove_nodes_bit game a' (Enum.clone en_a) l;
749                 let new_tgraph = game_to_transposed_graph game in
750                 for i = 0 to l - 1 do
751                   r.(i) <- (-1);
752                   f.(i) <- (-1);
753                   tgraph_full.(i) <- new_tgraph.(i);
754                 done;
755                 incr ric;
756                 op := !op + 2;
757                 is_up := false;
758                 m := 0;
759                 max_f := 0;
760                 even_set := Base.FSet.empty;
761                 odd_set := Base.FSet.empty;
762                 even_f := Base.FMap.empty;
763                 odd_f := Base.FMap.empty;
764                 if player = 0
765                 then (
766                   Enum.iter (fun i -> solution.(i) <- 0;) en_a;
767                   for i = 0 to l - 1 do
768                     if strategy.(i) <> -1
769                     then str_out.(i) <- strategy.(i)
770                   done;
771                 )
772                 else (
773                   Enum.iter (fun i -> solution.(i) <- 1;) en_a;
774                   for i = 0 to l - 1 do
775                     if strategy.(i) <> -1
776                     then str_out.(i) <- strategy.(i)
777                   done;
778                 );
779                 for i = 0 to l - 1 do
780                   str.(i) <- -1;
781                 done;
782               )
783             )
784           )
785         )
786         else iterate := false;
```

```
787         done;
788         (solution,str_out,!ric,!op,!up);;
789
790 let p1a3iter_main game =
791         let game' = Array.copy game in
792         let (solution,strategy,iteration,operation,upgrade) = solver3_iter game' in
793         print_string "\n";
794         Printf.printf "Recursions: %d, " iteration;
795         Printf.printf "Attractors: %d, " operation;
796         Printf.printf "Upgrades: %d\n" upgrade;
797         (solution,strategy);;
798
799
800 let solve = p1a3iter_main;;
801
802 register_solver solve "parity1a3" "p1a3" "use the new Iterative Parity Art1Alg3
        procedure";;
```