Universitá di Napoli “Federico II”

Dipartimento di Matematica e Applicazioni ”R. Caccioppoli”

Dottorato in Scienze Matematiche e Informatiche

Ciclo XXXI

Reasoning about LTL Synthesis over finite and infinite games

Antonio Di Stasio

A thesis submitted in fulfillment of the degree of

Doctor in Compute Science

Napoli, December 11, 2018

Supervisor: Prof. Ph.D. Aniello Murano

Contents

Abstract

In the last few years, research formal methods for the analysis and the verification of properties of systems has increased greatly. A meaningful contribution in this area has been given by algorithmic methods developed in the context of *synthesis*. The basic idea is simple and appealing: instead of developing a system and verifying that it satisfies its specification, we look for an automated procedure that, given the specification returns a system that is correct by construction. Synthesis of reactive systems is one of the most popular variants of this problem, in which we want to synthesize a system characterized by an ongoing interaction with the environment. In this setting, large effort has been devoted to analyze specifications given as formulas of linear temporal logic, *i.e.*, LTL synthesis.

Traditional approaches to LTL synthesis rely on transforming the LTL specification into parity deterministic automata, and then to parity games, for which a so-called winning region is computed. Computing such an automaton is, in the worst-case, double-exponential in the size of the LTL formula, and this becomes a computational bottleneck in using the synthesis process in practice.

The first part of this thesis is devoted to improve the solution of parity games as they are used in solving LTL synthesis, trying to give efficient techniques, in terms of running time and space consumption, for solving parity games. We start with the study and the implementation of an automata-theoretic technique to solve parity games. More precisely, we consider an algorithm introduced by Kupferman and Vardi that solves a parity game by solving the emptiness problem of a corresponding alternating parity automaton. Our empirical evaluation demonstrates that this

algorithm outperforms other algorithms when the game has a small number of priorities relative to the size of the game. In many concrete applications, we do indeed end up with parity games where the number of priorities is relatively small. This makes the new algorithm quite useful in practice.

We then provide a broad investigation of the symbolic approach for solving parity games. Specifically, we implement in a fresh tool, called `SymPGSolver`, four symbolic algorithms to solve parity games and compare their performances to the corresponding explicit versions for different classes of games. By means of benchmarks, we show that for random games, even for constrained random games, explicit algorithms actually perform better than symbolic algorithms. The situation changes, however, for structured games, where symbolic algorithms seem to have the advantage. This suggests that when evaluating algorithms for parity-game solving, it would be useful to have real benchmarks and not only random benchmarks, as the common practice has been.

LTL synthesis has been largely investigated also in artificial intelligence, and specifically in automated planning. Indeed, LTL synthesis corresponds to fully observable nondeterministic planning in which the domain is given compactly and the goal is an LTL formula, that in turn is related to two-player games with LTL goals. Finding a strategy for these games means to synthesize a plan for the planning problem. The last part of this thesis is then dedicated to investigate LTL synthesis under this different view. In particular, we study a generalized form of planning under partial observability, in which we have multiple, possibly infinitely many, planning domains with the same actions and observations, and goals expressed over observations, which are possibly temporally extended. By building on work on two-player games with imperfect information in the Formal Methods literature, we devise a general technique, generalizing the belief-state construction, to remove partial observability. This reduces the planning problem to a game of perfect information with a tight correspondence between plans and strategies. Then we instantiate the technique and solve some generalized planning problems.

Chapter 1

Introduction

1.1 Overview on LTL Synthesis

In the last decades many different methods have been introduced and deeply investigated for automatically check the reliability of hardware and software systems. A very attractive approach in this field is *model checking*, a framework developed independently by Clarke and Emerson [46] and by Queille and Sifakis [133] in early 80s. The idea behind model checking is simple and appealing: in order to check whether a system is correct with respect to a desired behavior, one formally checks instead whether a mathematical model representing the system, usually a *labelled-state transition system* or a *Kripke* structure, is correct with respect to a formal specification of the required behavior, usually described in terms of a *modal logic* formula, such as LTL [128], CTL [46], CTL* [59], μ-calculus [96], and the like. Model checking has been successfully applied to numerous theoretical and practical problems [3, 28, 42, 76, 79, 109, 120], such as verification of sequential circuit designs, communication protocols, software device drivers, security algorithms, to name a few, and the impact on industrial design practices is increasing. In the last four decades, model checking has been the subject of several books [14, 16, 43, 45, 47, 108] and surveys [44, 57, 139].

However, model checking, at least in the way it has been first conceived, turns out to be appropriate only to verify *closed systems*, which are characterized by the fact that their behavior is completely determined by internal states. These systems are often described as those having only one source of nondeterminism, coming from the system itself. Unfortunately, many systems in real life are *open*, in the sense that they are characterized by an ongoing iteration with an external environment and its behavior fully depends on this interaction. Consequently, all model-checking tools devised to verify the correctness of closed systems are not appropriate when applied to open systems. In fact, an appropriate model checking procedure of open systems should check the correctness of the system with respect to arbitrary environments and should take into account the ongoing interaction with the environment. This problem was first addressed and deeply investigated by Kupferman, Vardi and Wolper who introduced *module checking* [103], a specific framework for the verification of open systems against branching-time temporal-logics such as CTL, CTL*, and the like. Since its introduction, module checking has been a very active field of research and applied in several directions. Among the others we recall applications in the infinite-state recursive setting (i.e., pushdown systems) [25, 65], as well as hierarchical [123], and multi-agent systems [85, 86].

Although model checking has been a successful tool to verify systems, two main problems arise with the application of this method in practice: designing a correct system is hard and expensive; redesigning a system when it does not satisfy a desired behavior is still hard and expansive. Therefore, a more ambitious question is to ask whether we can *synthesize* such systems, that is, whether we can start from a specification and automatically design the system that satisfies such specification. Thus, synthesis goes beyond verification, in which both a specification and an implementation have to be given, by automatically deriving the latter from the former.

The synthesis problem was originally posed by Church [41] in the context of synthesizing switching circuits using the *monadic second-order logic of one successor* (S1S) as a specification language. The problem was solved by Rabin [134] and Büchi and Landweber [27] in the late 1960s showing its decidability. The modern approach to this problem was initiated by Pnueli and Rosner who introduced linear temporal logic *(LTL) synthesis* [129, 130], later extended to handle branching-time specifications, such as μ-calculus [57]. The Pnueli and Rosner idea can be summarized as follows. Given sets Σ_I and Σ_O of inputs and outputs, respectively (usually, $\Sigma_I = 2^I$ and $\Sigma_O = 2^O$, where I is a set of input signals and O is a set of output signals), we can view a system as a strategy $\sigma : \Sigma_I^* \rightarrow \Sigma_O$ that maps a finite sequence of sets of input signals into a set of output signals. When σ interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $\Sigma_I \cup \Sigma_O$. Though the system σ is deterministic, it induces a computation tree. The branches of the tree correspond to external nondeterminism, caused by different possible inputs. Thus, the tree has a fixed branching degree $|\Sigma_I|$, and it embodies all the possible inputs of σ. When we synthesize σ from LTL specification ϕ, we require ϕ to hold in all the paths of σ's computation tree [6]. Synthesis can be viewed as a turn-based game between two players: the environment and the system. In each round the system picks an output from Σ_O and then the environment picks a input from Σ_I, and the next round starts. The play is winning for the system in case the sequence is recognized as a desired behavior. Otherwise, the environment wins.

The classic approach to LTL synthesis [27, 130, 135] consists of the following steps. Given an LTL formula ϕ over $I \cup O$, construct a nondeterministic Büchi automaton[1] (NBA) A_ϕ that accepts all the words that satisfy ϕ. Translate A_ϕ into a deterministic parity automaton (DPA) B_ϕ. Convert B_ϕ into a parity game in which the transitions of B_ϕ are splitted into two parts: the first part refers to the output variables O controlled by the system, the second part to the input variables I controlled by the environment. Finally, check whether the system has a winning strategy in such game. (See Section 1.2 for the definition of parity games).

By looking at Figure 1.1, in reducing an LTL formula to nondeterministic Büchi automaton we may have an exponential blow-up in the size of the formula.

[1] For an introduction on automata on infinite objects and additional useful material we suggest [101, 107, 156].

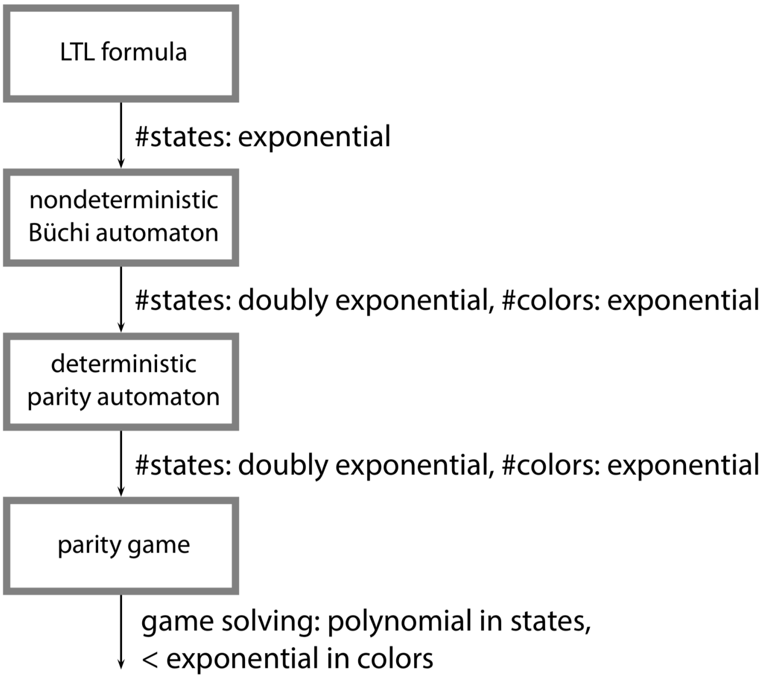

LTL formula
#states: exponential
nondeterministic Büchi automaton
#states: doubly exponential, #colors: exponential
deterministic parity automaton
#states: doubly exponential, #colors: exponential
parity game
game solving: polynomial in states,
< exponential in colors

Figure 1.1: Steps in solving LTL Synthesis.

Additionally, there is a double-exponential blow-up in the size of the formula to translating NBA into a deterministic parity automaton, and then to parity games. Finally, solving parity games witn n nodes and c colors is polynomial in n and exponential in c. This process provides an overall 2EXPTIME procedure to solve LTL synthesis [130]. Therefore, even though LTL synthesis is an appealing problem, despite extensive research, it remains a difficult problem. To mitigate this problem several approaches have been devised. On the one hand, researchers have looked at synthesis techniques for fragments of LTL that cover interesting classes of specifications, but for which the synthesis problem is easier. There exist several interesting cases where the synthesis problem can be solved in polynomial time, by using simpler automata or partial fragments [2, 75, 87, 162]. Representative cases are the work in [8] which presents an efficient quadratic solution to games, and hence synthesis problems, where the acceptance condition is one of the LTL formulas $\mathsf{G}\, p$ (*i.e.*, p is always true), $\mathsf{F}\, q$ (*i.e.*, q will become true at some point in the future), $\mathsf{G}\,\mathsf{F}\,p$ (*i.e.*, p is true infinitely often), or $\mathsf{F}\,\mathsf{G}\,q$ (*i.e.*, p will be always true from some point in the future). The work in [2] presents efficient synthesis approaches for various LTL fragments. In [20] is studied a generalization of the results of [8] and [2] into the wider class of Generalized Reactivity(1) formulas, or (GR(1)). *i.e.* formulas of the form:

$$(\mathsf{G}\,\mathsf{F}\,p_1 \wedge \cdots \mathsf{G}\,\mathsf{F}\,p_m) \rightarrow (\mathsf{G}\,\mathsf{F}\,q_1 \wedge \cdots \mathsf{G}\,\mathsf{F}\,p_n)$$

Thus, any synthesis problem whose specification is a GR(1) formula can be solved with complexity $O(mnN^2)$, where N is the size of the state space. On the same line, the work in [166] focuses on the attention on the *Safety LTL fragment*, which corresponds to the fragment of LTL consisting of Until-free formulas in Negation Normal Form. Such formulas express safety properties, meaning that every violating trace has a finite bad prefix that falsifies the formula [105]. For this strict subset of LTL, the synthesis problem can be reduced to safety games, which are far easier to be solved [110].

On the other hand, researchers have looked at more efficient algorithms for determination and solving parity games. Notable examples are the ones used to avoiding determination through alternate constructions [67, 68, 78, 102, 106]. Another approach proposed in [148] is to derive a deterministic parity automaton for each property in the specification, thus avoiding in most cases the application of the determination to large automata. This leads, however, from the solution of a parity game to a more complex conjunctive generalized parity game [38], *i.e.* a game with multiple parity winning conditions. In solving a generalized parity game, the time required to solve it depends on the number of components of a conjunctive generalized winning condition, then one may conjoin, in a heuristic way, some properties before translating them to deterministic automata as long as no blow up occurs. Or, using a different approach developed in [147] where safety and persistence properties can be dealt with before the rest of the properties without affecting the algorithm's completeness.

A recent line of work has given a more effort in trying to improve the classical approach for solving LTL synthesis bypassing the determinization [127, 141], and then producing different types of automata, such as DPA [62, 100], deterministic Rabin automata (DRA) and deterministic generalized Rabin automata (DGRA) [61,97,98], and limit-deterministic Büchi automata (LDBA) [48,95,146], without the intermediate step through nondeterministic automata [160]. To better understand this approach in Figure 1.2 we report a diagram taken from [99] that highlights the entire process in solving LTL synthesis based on the automata-theoretic approach.

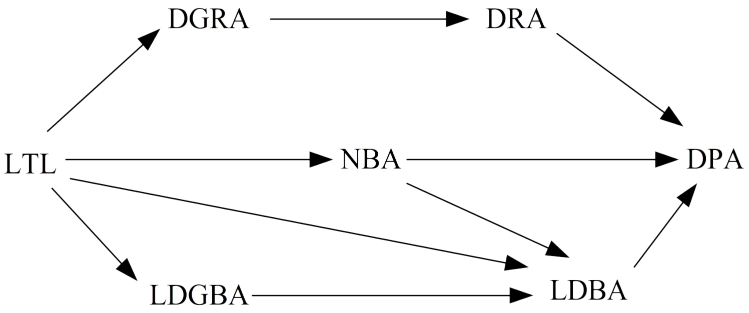


Figure 1.2: LTL translations to different types of automata.

1.2 Parity Games

Parity games [60,167] are abstract infinite-duration games that represent a powerful mathematical framework to address fundamental questions in computer science. They are intimately related to other infinite-round games, such as *mean* and *discounted* payoff, *stochastic*, and *multi-agent* games [5, 19, 31, 33, 37, 39, 116].

In the basic setting, parity games are two-player, turn-based, played on directed graphs whose nodes are labeled with priorities (also called, *colors*) and players have perfect information about the adversary moves. The two players, Player 0 and Player 1, take turns moving a token along the edges of the graph starting from a designated initial node. Thus, a play induces an infinite path and Player 0 wins the play if the smallest priority visited infinitely often is even; otherwise, Player 1 wins the play. The problem of deciding if Player 1 has a winning strategy (i.e., can induce a winning play) in a given parity game is known to be in UPTime ∩ CoUPTime [88]; whether a polynomial time solution exists is a long-standing open question [165].

Several algorithms for solving parity games have been proposed in the last two decades, aiming to tighten the known complexity bounds for the problem, as well as come out with solutions that work well in practice. Among the latter, we recall the recursive algorithm (RE) proposed by Zielonka [167], the Jurdziński's small-progress measures algorithm [89] (SPM), the strategy-improvement algorithm by Jurdziński et al. [161], the (subexponential) algorithm by Jurdzińki et al. [91], the big-step algorithm by Schewe [143], the priority promotion algorithm by Mogavero et al. [15], and the tangle learning algorithm by Tom van Dijk [158]. These algorithms have been implemented in the platforms PGSolver and Oink, and extensively investigated experimentally [70, 159].

This study has also led to a few optimizations in order to speed-up their execution time, such as improving the code implementation along with the use of better performing programming languages, or using parallelism [7, 80, 154].

Table 1.1 summarizes the classic algorithms along with their complexity, where n, e, and c denote the number of nodes, edges, and priorities, respectively.

Algorithm	Computational Complexity
Recursive (RE) [167]	$O(e \cdot n^c)$
Small Progress Measures (SP) [89]	$O(c \cdot e \cdot (\frac{n}{c})^{\frac{c}{2}})$
Strategy Improvement (SI) [161]	$O(2^e \cdot n \cdot e)$
Dominion Decomposition (DD) [91]	$O(n^{\sqrt{n}})$
Big Step (BS) [143]	$O(e \cdot n^{\frac{1}{3}c})$

Table 1.1: Parity algorithms along with their computational complexities.

Recently, Calude et al. [30] have given a major breakthrough providing a quasi-polynomial time algorithm for solving parity games that runs in time $O(n^{\lceil log(c)+6 \rceil})$.

Previously, the best known algorithm for parity games was DD which could solve parity games in $O(n^{\sqrt{n}})$, so this new result represents a significant advance in the understanding of parity games.

Their approach is to provide a compact witness that can be used to decide whether Player 0 wins a play. Traditionally, one must store the entire history of a play, so that when the players construct a cycle, we can easily find the largest priority on that cycle. The key observation in [30] is that a witness of poly-logarithmic size can be used instead. This allows them to simulate a parity game on an alternating Turing machine that uses poly-logarithmic space, which leads to a deterministic algorithm that uses quasi-polynomial time and space. This new result has already inspired follow-up works [49, 63, 90, 112]. However, benchmarks in the literature have demonstrated that both on random games and real examples the quasi-polynomial is not the best performing one. For this reason we decided in this thesis to concentrate on (the best performing) existing algorithms and drop the quasi-polynomial algorithm along the benchmarks.

In formal system design [46, 47, 107, 133], parity games arise as a natural evaluation machinery for the automatic synthesis and verification of distributed and reactive systems [3, 103, 157], as they allow to express liveness and safety properties in a very elegant and powerful way [121]. Specifically, the model-checking question, in case the specification is given as a μ-calculus formula [96], can be rephrased, in linear-time, as a parity game [60]. So, a parity game solver can be used as a model checker for a μ-calculus specification (and vice-versa), as well as for fragments such as CTL, CTL*, and the like.

In the automata-theoretic approach to μ-calculus model checking, under a linear-time translation, one can also reduce the verification problem to a question about automata. More precisely, one can take the product of the model and an alternating tree automaton accepting all tree models of the specification. This product can be defined as an alternating parity word automaton over a singleton alphabet, and the system is correct with respect to the specification iff this automaton is nonempty [107]. It has been proved there that the nonemptiness problems for nondeterministic parity tree automata and alternating parity word automata over a singleton alphabet are equivalent and their complexities coincide. For this reason, in the sequel we refer to these two kinds of automata just as parity automata. Hence, algorithms for the solution of the μ-calculus model checking problem, parity games, and the emptiness problem for parity automata can be interchangeably used to solve any of these problems, as they are linear-time equivalent.

Although each of the algorithms reported above uses a different technique to solve parity games, they all rely on algorithms that explicitly represent a game graph using a structure as a list or array that grows in proportion to the number of nodes. As the number of nodes in a graph may grow exponentially in the verification of finite-state systems, the size of the structures is usually the limiting factor in applying these algorithms to the analysis of the realistic systems. Hence for the analysis of large finite-state systems symbolic algorithms are necessary.

Symbolic algorithms are an efficient way to deal with extremely large graphs. These algorithms do not manipulate individual nodes, but, rather, sets of nodes. This is accomplished by representing the edge relation of the graph and sets of nodes as Boolean formulas, and by conducting the search by directly manipulating the symbolic representation. Both the edge relation and sets of nodes are described by Boolean functions and represented by Binary Decision Diagrams (BDDs) [26]. BDDs are widely used in various tools for the design and analysis of systems. Although they do not prevent state explosion, they allow us to verify in practice systems with extremely large state spaces that would be impossible to handle with explicit state. In formal verification, and specifically in model checking, symbolic algorithms have been widely investigated and, it was demonstrated that large regular models with as many as 10^{20} states can be handled [28, 118].

When using BDDs, however, it must been noticed that they are not sufficient in case the algorithms also require real-valued functions. To face this limitation, Algebraic Decision Diagrams (ADDs) [12] have been introduced to extend BDDs by allowing values from any arbitrary finite domain to be associated with the terminal nodes of the diagram.

Coming back to parity games, one can observe that while many explicit algorithms have been introduced and deeply investigated for solving such games, symbolic algorithms have been only marginally explored. In this direction, we just mention a symbolic implementation of `RE` [13, 93], which, however, has been done a for different purposes and no benchmark comparison with the explicit version has been carried out. Other works close to this topic and worth mentioning are [29, 35], where a symbolic version of the small progress measures algorithm has been theoretically studied but not implemented.

1.3 Contributions of the thesis

The first part of this thesis is devoted to improve the solution of parity games as they are used in solving LTL synthesis, trying to give efficient techniques, in terms of running time and space consumption, for solving parity games. We start with the study and the implementation of an algorithm, which we call `APT`, introduced by Kupferman and Vardi in [104], for solving parity games via emptiness checking of alternating parity automata, and evaluate its performance over the `PGSolver` platform. This algorithm has been sketched in [104], but not spelled out in detail and without a correctness proof, two major gaps that we fill here. The core idea of the `APT` algorithm is an efficient translation to *weak alternating automata* [122]. These are a special case of Büchi automata in which the set of states is partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that the automaton either stays at the same set or moves to a smaller set in the partial order. Thus, each run of a weak automaton eventually gets trapped in some set in the partition. The special structure of weak automata is reflected in their attractive computational properties. In particular, the nonemptiness problem for weak automata can be solved in linear time [107], while the best known upper bound for the nonemptiness problem for Büchi automata is quadratic [36]. Given an alternating parity word automaton with n states and c colors, the `APT` algorithm checks the emptiness of an equivalent weak alternating word automaton with O(n^c) states. The construction goes through a sequence of c intermediate automata. Each automaton in the sequence refines the state space of its predecessor and has one less color to check in its parity condition. Since one can check in linear time the emptiness of such an automaton, we get an $O(n^c)$ overall complexity for the addressed problem. `APT` does not construct the equivalent weak automaton directly, but applies the emptiness test directly, constructing the equivalent weak automaton on the fly. Recently, it has been provided a translation of alternating parity word automata into weak automata that involves only a quasi-polynomial size blow-up [21].

We then provide the first broad investigation of the symbolic approach for solving parity-games. We implement four symbolic algorithms and compare their performances to the corresponding explicit versions (implemented in Oink) for different classes of parity games. Specifically, we implement in a new tool, called `SymPGSolver` the symbolic versions of the Recursive [167], `APT` [104, 153], and the small progress measures algorithm presented in [29] and in [38]. The tool also has a collection of procedures to generate random games, as well as compare the performance of different symbolic algorithms. The main result we obtain from our comparisons is that for random games, and even for constrained random games explicit algorithms actually perform better than symbolic algorithms, most likely because BDDs do not offer any compression for random sets. The situation changes, however, for structured games, where the symbolic algorithms outperform the explicit ones. We take this as an important development because it suggests a methodological weakness in this field of investigation, due to the excessive reliance on random benchmarks. We believe that, in evaluating algorithms for parity-game solving, it would be useful to have real benchmarks and not only random benchmarks, as the common practice has been. This would lead to a deeper understanding of the relative merits of parity-game solving algorithms, both explicit and symbolic.

Planning and LTL Synthesis. LTL synthesis has been largely investigated also in Artificial Intelligence, and specifically in automated planning. Indeed, LTL synthesis corresponds to fully observable nondeterministic planning in which the domain is given compactly and the goal is an LTL formula, that in turn is related to two-players game with LTL goals. Finding a strategy for these games means to synthesize a plan for the planning problem. The last part of this thesis is then dedicated to investigate LTL synthesis under this different view.

Automated planning is a fundamental problem in Artificial Intelligence (AI). Given a deterministic dynamic system with a single known initial state and a goal condition, automated planning consists of finding a sequences of actions (the plan) to be performed by agents in order to achieve the goal [115]. The application of this notion in real-dynamic worlds is limited, in many situations, by three facts: i) the number of objects is neither small nor predetermined, ii) the agent is limited by its observations, iii) the agent wants to realize a goal that extends over time. For example, a preprogrammed driverless car cannot know in advance the number of obstacles it will enter in a road, or the positions of the other cars not in its view, though it wants to realize, among other goals, that every time it sees an obstacle it avoids it. This has inspired research in recent years on *generalized forms of planning* including conditional planning in partially observable domains [114, 138], planning with incomplete information for temporally extended goals [18, 53] and generalized planning for multiple domains or infinite domains [24, 64, 81, 82, 113, 150–152].

We use the following running example, taken from [82], to illustrate a generalized form of planning: the goal is to chop down a tree, and store the axe. The number of chops needed to fell the tree is unknown, but a look-action checks whether the tree is up or down. Intuitively, a plan solving this problem alternates looking and chopping until the tree is seen to be down, and then stores the axe. This scenario can be formalized as a partially-observable planning problem on a single infinite domain, or on the disjoint union of infinitely many finite domains.

The standard approach to solve planning under partial observability for *finite* domains is to reduce them to planning under complete observability. This is done by using the *belief-state construction* that removes partial observability and passes to the *belief-space* [17, 73]. The motivating problem of this work is to solve planning problems on infinite domains, and thus we are naturally lead to the problem of removing partial-observability from infinite domains.

Our technical contribution (Theorem 6.4.2) is a general sound and complete mathematical technique for removing imperfect information from a possibly infinite game **G** to get a game $\mathbf{G}^\beta$, possibly infinite, of perfect information. Our method builds upon the classic belief-state construction [73, 136, 137], also adopted in POMDPs [92, 111].[2] The classic belief-state construction fails for certain infinite games. We introduce a new component to the classic belief-state construction that isolates only those plays in the belief-space that correspond to plays in **G**. This new component is necessary and sufficient to solve the general case and capture also all infinite games **G**.

We apply our technique to game solving, i.e., deciding if a player has a winning strategy (this corresponds to deciding if there is a plan for a given planning instance). We remark that we consider strategies and plans that may depend on the history of the observations, not just the last observation. Besides showing how to solve the running Tree Chopping example, we report two cases. The first case is planning under partial observability for temporally extended goals expressed in LTL in finite domains (or a finite set of infinite domains sharing the same observations). This case generalizes well-known results in the AI literature [17, 53, 64, 81, 138]. The second case involves infinite domains. Note that because game solving is undecidable for computable infinite games (simply code the configuration space of a Turing Machine), solving games with infinite domains requires further computability assumptions. We focus on games generated by pushdown automata; these are infinite games that recently attracted the interest of the AI community [40, 124]. In particular these games have been solved assuming perfect information. By applying our technique, we extend their results to deal with imperfect information under the assumption that the stack remains observable (it is known that making the stack unobservable leads to undecidability [9]).

[2] However, our work considers nondeterminism rather than probability, and qualitative objectives rather than quantitative objects.

Table 1.2 summarizes the publications during my PhD and the concerned parts in the thesis.

References	Concerned part
Solving Parity Games Using An Automata-Based Algorithm [153]	Chapter 4
Solving Parity Games: Explicit vs Symbolic [54]	Chapter 5
Imperfect-Information Games and Generalized Planning [52]	Chapter 6

Table 1.2: Publications during my PhD.

Chapter 2

Infinite Games

In this chapter we introduce infinite games on directed graph. We will define what is a game, a strategy, how to win a game, under which conditions, etc.

2.1 Games

A game is played in an *arena* and the winner is determined by a winning condition. We will start by defining an arena.

An **arena** is a tuple $\mathcal{A} = (\mathrm{P}_0, \mathrm{P}_1, Mv)$ where,

- P_0 and P_1 are sets of nodes, where $\mathrm{P}_0 \cup \mathrm{P}_1 = \mathrm{P}$ and $\mathrm{P}_0 \cap \mathrm{P}_1 = \emptyset$;
- $Mv \subseteq \mathrm{P} \times \mathrm{P}$ is the left-total binary relation of moves. The set of successors of a node $v \in \mathrm{P}$ is defined by $Mv(q) \triangleq \{q' \in \mathrm{P} : (q, q') \in Mv\}$.

The games we are considering are played by two players, called Player 0 and Player 1. The two players take turns moving a token along the edges of the graph by means of the relation Mv starting from a designated initial node. Specifically, if the token is in a node belonging to Player 0, he moves, otherwise Player 1 moves. This is repeated infinitely often. We formally define a **play** over $\mathcal{A}$ as being an infinite sequence $\pi = q_1 \cdot q_2 \cdot \ldots \in \mathrm{Pth} \subseteq \mathrm{P}^\omega$ of nodes that agree with Mv, *i.e.*, $(\pi_i, \pi_{i+1}) \in Mv$, for each natural number $i \in \mathbb{N}$, and a **hystory** a finite sequence

$\pi = q_1 \cdot \ldots \cdot q_n \in \mathrm{Hst} \subseteq \mathrm{P}^*$ of nodes that agree with Mv, *i.e.*, $(\pi_i, \pi_{i+1}) \in Mv$ for each natural number $i \in [1, n-1]$. For a given history $\pi = q_1 \cdot \ldots \cdot q_n$, by $\mathsf{fst}(\pi) \triangleq q_1$ and $\mathsf{lst}(\pi) \triangleq q_n$ we denote the first and last node occurring in π, respectively. Finally, by Hst_0 (*resp.,* Hst_1) we denote the set of histories π such that $\mathsf{lst}(\pi) \in \mathrm{P}_0$ (*resp.,* $\mathsf{lst}(\pi) \in \mathrm{P}_1$).

For a path $\pi = q_0 \cdot q_1 \cdots \in \mathrm{P}^\omega$, we define the set $\mathsf{Inf}(\pi)$ of nodes visited infinitely often, *i.e.*,

$$\mathsf{Inf}(\pi) = \{q \in \mathrm{P} | \forall i \exists j > i,\ \pi(j) = q\}$$

Let $\mathcal{A}$ be an area and $W \subseteq \mathrm{P}^\omega$, a **game** is defined by the pair $\mathcal{G} = (\mathcal{A}, W)$, where W is a winning set of $\mathcal{G}$. Player 0 is the **winner** of a play π iff $\pi \in W$.

The winning sets we will consider are expressed on color sequences, that is, sequences where every node is associated to a color. Let $\mathcal{A}$ be an arena, we define a **coloring function** by $\mathsf{p} : \mathrm{P} \to C$. For a given play $\pi = q_1 \cdot q_2 \cdot \ldots$, by $\mathsf{p}(\pi) = \mathsf{p}(q_1) \cdot \mathsf{p}(q_2) \cdot \ldots \in \mathbb{N}^\omega$ we denote the associated color sequence. Moreover, we need to introduce an acceptance component α in the specification of games, which will arise in different formats. Therefore, we will write $W_{\mathsf{p}}(\alpha)$ for the winning set consisting of all infinite plays π where $\mathsf{p}(\pi)$ is accepted according to α.

Among the acceptance conditions studied in literature we consider the following ones:

- *Büchi* condition, where $\alpha \subseteq C = \mathrm{P}$, $\pi \in W_{\mathsf{p}}(\alpha)$ iff $\mathsf{Inf}(\mathsf{p}(\pi)) \cap \alpha \neq \emptyset$.

- *co-Büchi* condition, where $\alpha \subseteq C = \mathrm{P}$, $\pi \in W_{\mathsf{p}}(\alpha)$ iff $\mathsf{Inf}(\mathsf{p}(\pi) \cap \alpha = \emptyset$.

- *parity* condition, where C is a finite subset of integers $\{0, \ldots, c-1\}$. A play $\pi \in W_{\mathsf{p}}(\alpha)$ iff $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even. We also refer to *co-parity* condition $\sim \alpha$. A play $\pi \in W_{\mathsf{p}}(\sim \alpha)$ iff $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is odd.

- *Rabin* condition, where $\alpha = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ with $E_i, F_i \subseteq C = \mathrm{P}$, for $1 \leq i \leq k$. A play $\pi \in W_{\mathsf{p}}(\alpha)$ iff there exists an index i for which $\mathsf{Inf}(\mathsf{p}(\pi) \cap E_i = \emptyset$ and $\mathsf{Inf}(\mathsf{p}(\pi) \cap F_i \neq \emptyset$.

- *Street* condition, where $\alpha = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ with $E_i, F_i \subseteq C = \mathrm{P}$, for $1 \leq i \leq k$. A play $\pi \in W_{\mathsf{p}}(\alpha)$ iff for all $i \in [1 \ldots k]$ we have $\mathsf{Inf}(\mathsf{p}(\pi) \cap F_i \neq \emptyset \Rightarrow \mathsf{Inf}(\mathsf{p}(\pi) \cap E_i \neq \emptyset$.

Depending on the actual acceptance condition we will speak of **Büchi**, **parity**, **Rabin**, and **Street** games. For simplicity, in the rest of this thesis we will just write $(\mathcal{A}, \mathsf{p}, \alpha)$, instead of $(\mathcal{A}, W_{\mathsf{p}}(\alpha))$ to indicate a game with a certain winning condition.

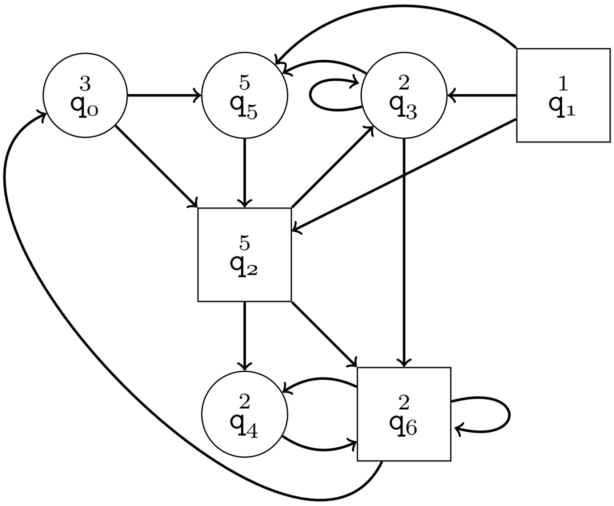


Figure 2.1: A parity game.

As a running example, consider the game depicted in Figure 2.1. The set of players's nodes is $\mathrm{P}_0 = \{q_0, q_3, q_4, q_5\}$ and $\mathrm{P}_1 = \{q_1, q_2, q_6\}$; we use circles to denote nodes belonging to Player 0 and squares for those belonging to Player 1. *Mv* is described by arrows. Finally, the priority function p is given by $\mathsf{p}(q_1) = 1$, $\mathsf{p}(q_3) = \mathsf{p}(q_4) = \mathsf{p}(q_6) = 2$, $\mathsf{p}(q_0) = 3$, and $\mathsf{p}(q_2) = \mathsf{p}(q_5) = 5$.

A possible play is $\overline{\pi} = q_1 \cdot q_5 \cdot q_2 \cdot (q_3)^{\omega}$, while a possible history is given by $\overline{\pi} = q_1 {\cdot} q_5 {\cdot} q_2 {\cdot} q_3$. The associated priority sequence to $\overline{\pi}$ is given by $\mathsf{p}(\overline{\pi}) = 1{\cdot}5{\cdot}5{\cdot}(2)^{\omega}$. Moreover, we have that $\mathsf{Inf}(\overline{\pi}) = \{q_3\}$ and $\mathsf{Inf}(\mathsf{p}(\overline{\pi})) = \{2\}$. If we consider as winning condition the parity acceptance condition, the play π is winning for Player 0.

2.2 Strategies and Determinacy

A **strategy**, informally, is a method that Player 0 (*resp.,* Player 1) applies to determine the next action to take, at any stage of the game, depending on the history of play up to that stage. Formally, a Player 0 (*resp.,* Player 1) strategy is a function $\mathsf{str}_0 : \mathrm{Hst}_0 \rightarrow \mathrm{P}$ (*resp.,* $\mathsf{str}_1 : \mathrm{Hst}_1 \rightarrow \mathrm{P}$) such that, for all $\pi \in \mathrm{Hst}_0$ (*resp.,* $\pi \in \mathrm{Hst}_1$), it holds that $(\mathsf{lst}(\pi), \mathsf{str}_0(\pi)) \in Mv$ (*resp.,* $\mathsf{lst}(\pi), \mathsf{str}_1(\pi)) \in Mv$).

Given a node q, Player 0 and a Player 1 strategies str_0 and str_1, the play of these two strategies, denoted by $\mathsf{play}(q, \mathsf{str}_0, \mathsf{str}_1)$, is the only play π in the game that starts in q and agrees with both Player 0 and Player 1 strategies, *i.e.*, for all $i \in \mathbb{N}$, if $\pi_i \in \mathrm{P}_0$, then $\pi_{i+1} = \mathsf{str}_0(\pi_i)$, and $\pi_{i+1} = \mathsf{str}_1(\pi_i)$, otherwise.

A strategy str_0 (*resp.,* str_1) is *memoryless* if, for all $\pi_1, \pi_2 \in \mathrm{Hst}_0$ (*resp.,* $\pi_1, \pi_2 \in \mathrm{Hst}_1$), with $\mathsf{lst}(\pi_1) = \mathsf{lst}(\pi_2)$, it holds that $\mathsf{str}_0(\pi_1) = \mathsf{str}_0(\pi_2)$ (*resp.,* $\mathsf{str}_0(\pi_1) = \mathsf{str}_1(\pi_2)$). Note that a memoryless strategy can be defined on the set of nodes, instead of the set of histories. Thus we have that they are of the form $\mathsf{str}_0 : \mathrm{P}_0 \rightarrow \mathrm{P}$ and $\mathsf{str}_1 : \mathrm{P}_1 \rightarrow \mathrm{P}$.

We say that Player 0 (*resp.,* Player 1) *wins* the game $\mathcal{G}$ from node q if there exists a Player 0 (*resp.,* Player 1) strategy str_0 (*resp.,* str_1) such that, for all Player 1 (*resp.,* Player 0) strategies str_1 (*resp.,* str_0) it holds that $\mathsf{play}(q, \mathsf{str}_0, \mathsf{str}_1)$ is winning for Player 0 (*resp.,* Player 1).

A node q is *winning* for Player 0 (*resp.,* Player 1) if Player 0 (*resp.,* Player 1) wins the game from q. By $\mathrm{Win}_0(\mathcal{G})$ (*resp.,* $\mathrm{Win}_1(\mathcal{G})$) we denote the set of winning nodes in $\mathcal{G}$ for Player 0 (*resp.,* Player 1).

A game is **determined** if for every node q, either $q \in \mathrm{Win}_0(\mathcal{G})$ or $q \in \mathrm{Win}_1(\mathcal{G})$ [60].

In this thesis we will focus on parity and Rabin games. In the following we report some important results about these games.

Theorem 2.2.1 ([117]). *Parity games are determined.*

Theorem 2.2.2 ([167]). *In every parity game, both players win memoryless.*

Theorem 2.2.3 ([23]). *In every Rabin game, Player 0 has memoryless winning strategy on his winning region. Symmetrically, in every Street game, Player 1 has a memoryless strategy on his winning region.*

2.2.1 Subgames, traps, and dominions

Let $\mathcal{G} = (\mathcal{A}, \mathsf{p}, \alpha)$ be a game and $U \subseteq \mathrm{P}$. The subgraph induced by U, denoted by $\mathcal{G}[U]$, is defined as follows,

$$\mathcal{G}[U] = (\mathrm{P}_0 \setminus U, \, \mathrm{P}_1 \setminus U, Mv \setminus (U \times \mathrm{P} \cup \mathrm{P} \times U), \mathsf{p}_{|\mathrm{P} \setminus U})$$

where $\mathsf{p}_{|\mathrm{P}\setminus U}$ is the restriction of p to U.

$\mathcal{G}[U]$ is a *subgame* of $\mathcal{G}$ if it is a game, *i.e.*, if each node of U has at least one successor in U.

For example, in the game of Figure 2.1, $\mathcal{G}[\{q_4, q_6\}]$ is a subgame of $\mathcal{G}$. Instead, $\mathcal{G}[\{q_0, q_4, q_6\}]$ is not a subgame of $\mathcal{G}$.

Let $\sigma \in \{0, 1\}$, a *σ-trap* in a game $\mathcal{G}$ is any nonempty set U of nodes $\mathcal{G}$ such that

$$\forall v \in U \cap \mathrm{P}_\sigma, \; Mv(q) \subseteq U \text{ and } \forall v \in U \cap \mathrm{P}_{1-\sigma}, \; Mv(q) \neq U$$

If the token is in a σ-trap U, then Player σ can play a strategy consisting in choosing always successors inside of U. On the other hand, since all successors of Player σ nodes in U belong to U, Player σ has no possibility to force the token outside of U.

A set $U \subseteq \mathrm{P}$ is a *σ-dominion* if U is $1 - \sigma$-trap and Player σ has a winning strategy for the winning condition α from all nodes in U in the subgame $\mathcal{G}[U]$.

As a example, in the game of Figure 2.1, the set $U = \{q_4, q_6\}$ is a 0-trap, but it is not a 1-dominion since Player 1 loses from all nodes in U. While the set $\{q_2, q_3, q_4, q_6\}$ is a 1-trap, and a 0-dominion.

2.3 Attractor

We now define the notion of **attractor**, core of some algorithms for solving parity and Rabin games we will describe in later chapters. Intuitively, given a set of nodes $F \subseteq \mathrm{P}$, the attractor of U for a Player $\sigma \in \{0, 1\}$, indicated by $Attr_\sigma(G, F)$, represents those nodes that σ can force the play toward. That is, Player σ can force any play to behave in a certain way, even though this does not mean that Player σ wins the game. Formally, for all $k \in N$:

$$Attr^0_\sigma(G, F) = U$$
$$Attr^{k+1}_\sigma(G, F) = Attr^k_\sigma(G, F) \cup \{v \in \mathrm{P}_\sigma | \exists w \in Attr^k_\sigma(G, F) \ s.t \ (v, w) \in E\}$$

Then, we have that $Attr_\sigma(G, F) = \bigcup_{k \in N} Attr^k_\sigma(G, F)$.

Note that, any attractor on a finite game is necessarily finite, and we can easily see that in a finite area with n nodes and m edges the attractor can be computed in time $\mathcal{O}(m)$. In the Figure 2.2 we show a possible example of attractor.

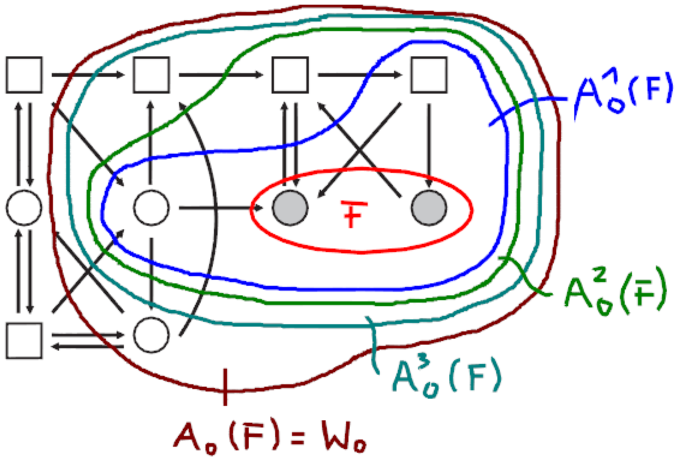


Figure 2.2: Example of attractor

The attractor of a set F for Player σ defines the winning region for Player σ in a reachability game [74], *i.e.*, game in which Player σ wins a play π if some node from

U occurs in π. A memoryless winning strategy $\mathsf{str}_\sigma(\pi)$ is called a corresponding *attractor strategy* for Player σ, and it is defined as follows.

$$\mathsf{str}_\sigma(\pi) = \begin{cases} w, & \exists k > 0 \text{ s.t } \mathsf{lst}(\pi) \in (\mathrm{P}_0 \cap \mathit{Attr}_\sigma^k(\mathcal{G}, F)) \setminus \mathit{Attr}_\sigma^{k-1}(\mathcal{G}, F) \\ & \quad \text{and } w \in A_\sigma^{k-1}(\mathcal{G}, F) \cap \mathit{Mv}(\mathsf{lst}(\pi)) \\ \perp & \text{otherwise} \end{cases}$$

Chapter 3

Algorithms for Solving Parity Games

In this chapter we are going to give a brief overview of the two most studied algorithms for solving parity games, *i.e.*, the recursive algorithm by Zielonka [167] and the small progress measures algorithm by Jurdzinski [88].

3.1 Small Progress Measures Algorithm

We start with an informal description of the *progress measure*, and then provide the formal definition. The progress measure is based on a so-called *lexicographic* ranking function that assigns a rank to each node v, where the rank is a vector of counters that indicates the number of times Player 1 can force a play to visit an odd priority node before a node with lower even priority is reached. If Player 1 can ensure a counter value of at least n, with n number of nodes, then he can ensure that a cycle with lowest priority odd is reached from v and therefore Player 1 can win from v. Conversely, if Player 1 can reach a cycle with lowest priority odd before reaching a lowest even priority, then he can also force a play to visit an odd priority n times before reaching a lower even priority. When a node v is classified as winning for Player 1, it is marked with the rank $\top$.

The small progress measures algorithm (SPM, in short) computes the progress measure by updating the rank of a node according to the ranks of its successors, *i.e.*, computing a least fixed point for all nodes with respect to ranking functions.

First, we briefly recall the basic notions. Let c be the maximum priority in $\mathcal{G}$ and $d \in \mathbb{N}^c$ be a c-tuple of non-negative integers, *i.e.*, $d = (d_0, d_1, \ldots, d_{c-1})$. We use the comparison symbol $<$ to denote the lexicographic ordering applied to tuples of natural numbers and, the comparison symbol $<_i$ with $i \in \mathbb{N}$, to denote the lexicographic ordering on $\mathbb{N}^i$ applied to tuples restricted to their first i components. Moreover, for all $n \in \mathbb{N}$, by $[n]$ we denote the set $\{0, \ldots, n-1\}$.

Let n_i be the number of nodes with priority i for odd i, and let $n_i = 0$ for even i. The progress measure domain is defined by the set $M_G^\top = M_G \cup \{\top\}$ with $M_G = (M_0 \times M_1 \times \ldots \times M_{c-2} \times M_{c-1})$, where $M_i = [n_i + 1]$ for $i \in \{0, \ldots, c-1\}$. The element $\top$ is the biggest element such that $m < \top$ for all $m \in M_G$.

For a tuple $d = (d_0, \ldots, d_{c-1})$, let $\langle d \rangle_l$ be the tuple $(d_0, d_1, \ldots, d_l, 0, \ldots, 0)$ where all elements with index more than l are set to 0. We indicate with $\mathrm{inc}(d)$ the smallest tuple $d' \in M_G^\top$ such that $d < d'$, and with $\mathrm{dec}(d)$ the greatest tuple $d' \in M_G^\top$ such that $d' < d$. These notions can also be restricted to the tuples in $\mathbb{N}^i$ defining $\mathrm{inc}_l(d)$ (*resp.* $\mathrm{dec}_l(d)$) with $l > 0$, to be the smallest (*resp.* the greatest) tuple $d' \in M_G^\top$ such that $d <_l d'$ (*resp.* $d' <_l d$). In particular, for $d = \top$ we have $\mathrm{inc}_l(d) = d$. Otherwise, $\mathrm{inc}_l(d) = \langle d \rangle_l$ if l is even and $\min\{y \in M_G^\top | y >_l d\}$ if l is odd. And, for $d = (0, \ldots, 0)$ we have $\mathrm{dec}_l(0, \ldots, 0) = (0, \ldots, 0)$. Otherwise, if $\langle x \rangle_l > (0, \ldots, 0)$ then $\mathrm{dec}_l(x) = \min\{y \in M_G^\top | x = inc_l(y)\}$.

We now report the definitions of ranking function, the best and the lift operators, and finally the algorithm to compute the small progress measures.

A *ranking function* $\varrho : \mathrm{P} \to M_G^\top$ associates to each node either one of the c dimensional vectors in M_G or the top element $\top$.

The *best* operation defines the edge that leads to the lowest rank and Player i has to choose when he owns the node. Formally, given the ranking function ϱ and a node v, the best function is defined as follows.

$$\mathrm{best}(\varrho, v) = \begin{cases} \min\{\varrho(w) | (v, w) \in Mv\}, & \text{if } v \in \mathrm{P}_0 \\ \max\{\varrho(w) | (v, w) \in Mv\}, & \text{if } v \in \mathrm{P}_1 \end{cases}$$

The *Lift* operator uses the function *best* to define the increment of a node v based on its priority and the ranks of its neighbors. The formal definition follows.

$$\mathrm{Lift}(\varrho, v)(u) = \begin{cases} \mathrm{inc}_{\mathsf{p}(v)}(best(\varrho, v)), & v = u \\ \varrho(u), & \text{otherwise} \end{cases}$$

The Lift(., v)-operators are monotone and the progress measure for a parity game is defined as the least fixed point of all Lift(., v)-operators.

The algorithm to compute a progress measure starts assigning 0 to every node. Then, it applies the lift operator as long as $\mathrm{Lift}(\varrho, v)(u) > \varrho(v)$ for some $v \in \mathrm{P}$. When the algorithm terminates, the least fixed point of all lift operators has been found.

The following lemma implies that to solve a parity game $\mathcal{G}$ it is sufficient to provide an algorithm, reported in Algorithm 1, that computes the least fixed point of all Lift(., v)-operators.

Lemma 1 ([88]). *If ϱ is a progress measure then the set of nodes with $\varrho(v) < \top$ is the set of winning nodes for Player 0.*

Algorithm 1 Progress Measure Lifting

```
$\mu := \lambda v \in \mathrm{P}.(0, \cdots, 0)$
**while** $\mu \sqsubset \mathrm{Lift}(\mu, v)$ for some $v \in \mathrm{P}$ **do**
    $\mu := \mathrm{Lift}(\mu, v)$
```

The algorithm requires $O(c \cdot m \cdot M_G^{\top}) = O\Big(c \cdot m \cdot \left(\frac{n}{\lfloor c/2 \rfloor}\right)^{\lfloor c/2 \rfloor}\Big)$ and it works in space $O(c \cdot n)$.

3.2 Recursive (Zielonka) Algorithm

The recursive algorithm (`RE`, for short), reported in Algorithm 2, is one of the first exponential-time algorithm for solving parity games. It is based on the work of McNaughton [119] and it was explicitly presented as a solver for parity games by Zielonka [167]. The algorithm makes use of a divide and conquer technique and its core subroutine is the *attractor* described in Section 2.3.

Algorithm 2 Recursive Algorithm

```
procedure Solve(G)
    if (P == $\emptyset$) then
        $(W_0, W_1) = (\emptyset, \emptyset)$
    else
        d = maximal priority in $\mathcal{G}$
        $U = \{ v \in V \mid \mathsf{p}(v) = d \}$
        p = d % 2
        j = 1 - p
        $A = Attr_p(U)$
        $(W'_0, W'_1)$ = Solve $(\mathcal{G} \setminus A)$
        if $W'_j == \emptyset$ then
            $W_p = W'_p \cup A$
            $W_j = \emptyset$
        else
            $B = Attr_j(W_1^j)$
            $(W'_0, W'_1)$ = Solve $(\mathcal{G} \setminus B)$
            $W_p = W'_p$
            $W_j = W'_j \cup B$
    return $(W_0, W_1)$
```

At each step, the algorithm removes all nodes with the highest priority p, denoted by U, together with all nodes Player $i = p \ mod\ 2$ can attract to them, denoted by A, and recursively computes the winning sets (W_0', W_1') for Player 0 and Player 1, respectively, on the remaining subgame $\mathcal{G}[A]$ (See Figure 3.1).

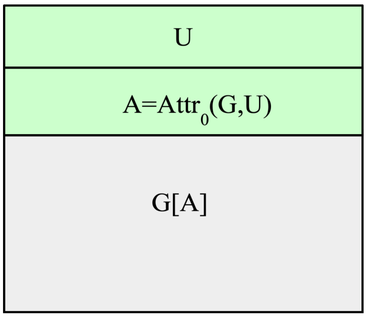


Figure 3.1: Execution of RE, lines 6-9, with p even.

At this point, there are two cases to be considered. First, if Player i wins $\mathcal{G}[A]$, then he also wins the whole game $\mathcal{G}$. Indeed, whenever Player $1-i$ decides to visit A, Player i's winning strategy would be to reach U. Then, every play that visits A infinitely often has p as the highest priority occurring infinitely often, or otherwise it stays eventually in $\mathcal{G}[A]$, and hence is won by i.

Second, if Player i does not win the whole subgame $\mathcal{G}[A]$, *i.e.*, W_{1-i}' is non empty, then Player $1-i$ wins on a subset W_{1-i}' in $\mathcal{G}[A]$. And, since Player i cannot force Player $1-i$ to leave W_{1-i}', we have that Player $1-i$ also wins on W_{1-i}' in the game $\mathcal{G}$. Hence, the algorithm computes the attractor B for Player $1-i$ of W_{1-i}' and recursively solves the subgame $\mathcal{G}[A]$ (See Figure 3.2).

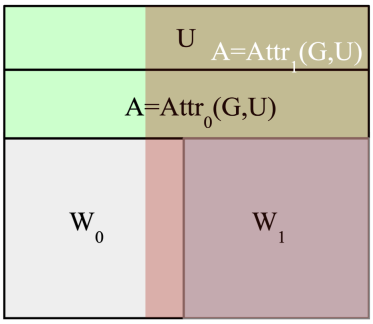


Figure 3.2: Execution of RE, lines 15-16, with p even.

Chapter 4

An Automata Approach for Parity Games

In this chapter, we study and implement an algorithm, which we call `APT`, introduced by Kupferman and Vardi in [104], for solving parity games via emptiness checking of alternating parity automata, and evaluate its performance over the `PGSolver` platform. This algorithm has been sketched in [104], but not spelled out in detail and without a correctness proof, two major gaps that we fill here.

4.1 The `APT` Algorithm

The `APT` algorithm makes use of two special sets of nodes, denoted by V and A, and called set of *Visiting* and *Avoiding*, respectively. Intuitively, a node is declared visiting for a player at the stage in which it is clear that, by reaching that node, he can surely induce a winning play and thus winning the game. Conversely, a node is declared avoiding for a player whenever it is clear that, by reaching that node, he is not able to induce any winning play and thus losing the game. The algorithm, in turns, tries to partition all nodes of the game into these two sets. The introduction of this two sets leads to define a new type of game we call *Extended Parity Game*. The formal definition follows.

Definition 1. *Let $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p}, \alpha)$ be a parity game, an* Extended Parity Game*, (*Epg*, for short) is a tuple $\mathcal{G}_{\mathcal{E}} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p}', V, A, \alpha)$, where P_0, P_1, Mv, and α are defined as in $\mathcal{G}$. The subsets of nodes $\mathrm{V}, \mathrm{A} \subseteq \mathrm{P} = \mathrm{P}_0 \cup \mathrm{P}_1$ are two disjoint sets of* Visiting *and* Avoiding *nodes, respectively. Finally, $\mathsf{p}' : (\mathrm{P} \setminus V \cup A) \rightarrow \mathbb{N}$ is a parity function mapping every non-visiting and non-avoiding set to a color.*

The notions of histories and plays of $\mathcal{G}_{\mathcal{E}}$ are equivalent to the ones given for $\mathcal{G}$. Moreover, as far as the definition of strategies is concerned, we say that a play π that is in $\mathrm{P} \cdot (\mathrm{P} \setminus (\mathrm{V} \cup \mathrm{A}))^* \cdot \mathrm{V} \cdot \mathrm{P}^\omega$ is winning for Player 0, while a play π that is in $\mathrm{P} \cdot (\mathrm{P} \setminus (\mathrm{V} \cup \mathrm{A}))^* \cdot \mathrm{A} \cdot \mathrm{P}^\omega$ is winning for Player 1. For a play π that never hits either V or A, we say that it is winning for Player 0 iff it satisfies the parity condition, *i.e.*, $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even, otherwise it is winning for Player 1.

Clearly, parity games are special cases of Epgs in which $\mathrm{V} = \mathrm{A} = \emptyset$. Conversely, one can transform an Epg into an equivalent parity game with the same winning set by simply replacing every outgoing edge with loop to every node in $\mathrm{V} \cup \mathrm{A}$ and then relabeling each node in V and A with an even and an odd number, respectively.

In order to describe how to solve Epgs, we introduce some notation. By $\mathrm{F}_i = \mathsf{p}^{-1}(i)$ we denote the set of all nodes labeled with i. Doing that, the parity condition can be described as a finite sequence $\alpha = \mathrm{F}_1 \cdot \ldots \cdot \mathrm{F}_k$ of sets, which alternates from sets of nodes with even priorities to sets of nodes with odd priorities and the other way round, forming a partition of the set of nodes, ordered by the priority assigned by the parity function. We call the set of nodes F_i an even (*resp.,* odd) parity set if i is even (*resp.,* odd).

For a given set $\mathrm{X} \subseteq \mathrm{P}$, by $\mathsf{force}_0(\mathrm{X}) = \{q \in \mathrm{P}_0 : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{P}_1 : \mathrm{X} \subseteq Mv(q)\}$ we denote the set of nodes from which Player 0 can force to move in the set X. Analogously, by $\mathsf{force}_1(\mathrm{X}) = \{q \in \mathrm{P}_1 : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{P}_0 : \mathrm{X} \subseteq Mv(q)\}$ we denote the set of nodes from which Player 1 can force to move in the set X. For example, in the parity game in Figure 4.1, $\mathsf{force}_1(\{\mathsf{q}_2\}) = \{\mathsf{q}_1, \mathsf{q}_0\}$.

We now introduce two functions that are co-inductively defined that will be used to compute the winning sets of Player 0 and Player 1, respectively.

For a given Epg $\mathcal{G}_{\mathcal{E}}$ with α being the representation of its parity condition, V its visiting set, and A its avoiding set, we define the functions $\text{Win}_0(\alpha, \text{V}, \text{A})$ and $\text{Win}_1(\alpha, \text{A}, \text{V})$. Informally, $\text{Win}_0(\alpha, \text{V}, \text{A})$ computes the set of nodes from which the player 0 has a strategy that avoids A and either force a visit in V or he wins the parity condition. The definition is symmetric for the function $\text{Win}_1(\alpha, \text{A}, \text{V})$. Formally, we define $\text{Win}_0(\alpha, \text{V}, \text{A})$ and $\text{Win}_1(\alpha, \text{A}, \text{V})$ as follows.

If $\alpha = \varepsilon$ is the empty sequence, then

- $\text{Win}_0(\varepsilon, \text{V}, \text{A}) = \mathsf{force}_0(\text{V})$ and
- $\text{Win}_1(\varepsilon, \text{A}, \text{V}) = \mathsf{force}_1(\text{A})$.

Otherwise, if $\alpha = \text{F} \cdot \alpha'$, for some set F, then

- $\text{Win}_0(\text{F} \cdot \alpha', \text{V}, \text{A}) = \mu \text{Y}(\text{P} \setminus \text{Win}_1(\alpha', \text{A} \cup (\text{F} \setminus \text{Y}), \text{V} \cup (\text{F} \cap \text{Y})))$ and
- $\text{Win}_1(\text{F} \cdot \alpha', \text{A}, \text{V}) = \mu \text{Y}(\text{P} \setminus \text{Win}_0(\alpha', \text{V} \cup (\text{F} \setminus \text{Y}), \text{A} \cup (\text{F} \cap \text{Y})))$,

where μ is the least fixed-point operator[1].

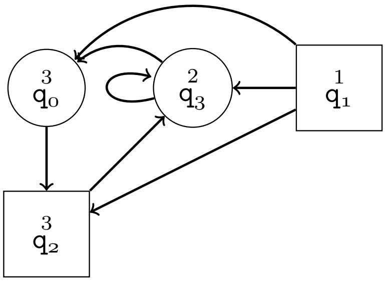


Figure 4.1: A parity game with no gaps.

To better understand how APT solves a parity game we show a simple piece of execution on the example in Fig 4.1. It is easy to see that such parity game is

[1] The unravellings of Win_0 and Win_1 have some anologies with the fixed-point formula introduced in [163] also used to solve parity games. Unlike our work, however, the formula presented there is just a translation of the Zielonka algorithm [167].

won by Player 0 in all the possible starting nodes. Then, the fixpoint returns the entire set P. The parity condition is given by $\alpha = \mathrm{F}_1 \cdot \mathrm{F}_2 \cdot \mathrm{F}_3$, where $\mathrm{F}_1 = \{\mathsf{q}_1\}$, $\mathrm{F}_2 = \{\mathsf{q}_3\}$, $\mathrm{F}_3 = \{\mathsf{q}_0, \mathsf{q}_2\}$. The repeated application of functions $\mathrm{Win}_0(\alpha, \mathrm{V}, \mathrm{A})$ and $\mathrm{Win}_1(\alpha, \mathrm{A}, \mathrm{V})$ returns:

$$\mathrm{Win}_0(\alpha, \emptyset, \emptyset) = \mu \mathrm{Y}^1(\mathrm{P} \setminus \mu \mathrm{Y}^2(\mathrm{P} \setminus \mu \mathrm{Y}^3(\mathrm{P} \setminus \mathsf{force}_1(\mathrm{V}^4))))$$

in which the sets Y^i are the nested fixpoint of the formula, while the set V^4 is obtained by recursively applying the following:

- $\mathrm{V}^1 = \emptyset$, $\mathrm{V}^{i+1} = \mathrm{A}^i \cup (\mathrm{F}_i \setminus \mathrm{Y}^i)$, and
- $\mathrm{A}^1 = \emptyset$, $\mathrm{A}^{i+1} = \mathrm{V}^i \cup (\mathrm{F}_i \cap \mathrm{Y}^i)$.

As a first step of the fixpoint computation, we have that $\mathrm{Y}^1 = \mathrm{Y}^2 = \mathrm{Y}^3 = \emptyset$. Then, by following the two iterations above for the example in Figure 4.1, we obtain that at the second step:

- $\mathrm{V}^2 = \mathrm{A}^1 \cup (\mathrm{F}_1 \setminus \mathrm{Y}^1) = \{\mathsf{q}_1\}$,
- $\mathrm{A}^2 = \mathrm{V}^1 \cup (\mathrm{F}_1 \cap \mathrm{Y}^1) = \emptyset$.

At the third step:

- $\mathrm{V}^3 = \mathrm{A}^2 \cup (\mathrm{F}_2 \setminus \mathrm{Y}^2) = \{\mathsf{q}_3\}$,
- $\mathrm{A}^3 = \mathrm{V}^2 \cup (\mathrm{F}_2 \cap \mathrm{Y}^2) = \{\mathsf{q}_1\}$.

At the fourth step:

- $\mathrm{V}^4 = \mathrm{A}^3 \cup (\mathrm{F}_3 \setminus \mathrm{Y}^3) = \{\mathsf{q}_0, \mathsf{q}_1, \mathsf{q}_2\}$,
- $\mathrm{A}^4 = \mathrm{V}^3 \cup (\mathrm{F}_3 \cap \mathrm{Y}^3) = \{\mathsf{q}_3\}$.

At this point we have that $\mathsf{force}_1(\mathrm{V}^4) = \{\mathsf{q}_0, \mathsf{q}_1, \mathsf{q}_2, \mathsf{q}\} \neq \emptyset = \mathrm{Y}^3$. This means that the fixpoint for Y^3 has not been reached yet. Then, we update the set Y^3 with the new value and compute again V^4. This procedure is repeated up to the point in which $\mathsf{force}_1(V^4) = Y^3$, which means that the fixpoint for Y^3 has been reached. Then we iteratively proceed to compute $Y^2 = P \setminus Y^3$ until a fixpoint for Y^2 is reached. Note that the sets A^i and V^i depends on the Y^i and so they need to be updated step by step. As soon as a fixpoint for Y_1 is reached, the algorithm returns the set $P \setminus Y_1$. As a fundamental observation, note that, due to the fact that the fixpoint operations are nested one to the next, updating the value of Y^i implies that every Y^j, with $j > i$, needs to be reset to the empty set.

We now prove the correctness of this procedure. Note that the algorithm is an adaptation of the one provided by Kupferman and Vardi in [104], for which a proof of correctness has never been shown.

Theorem 4.1.1. *Let* $\mathcal{G}_\mathcal{E} = (P_0, P_1, Mv, \mathsf{p}', V, A, \alpha)$ *be an* EPG *with* α *being the parity sequence condition. Then, the following properties hold.*

1. *If* $\alpha = \varepsilon$ *then* $\mathrm{Win}_0(\mathcal{G}_\mathcal{E}) = \mathrm{Win}_0(\alpha, V, A)$ *and* $\mathrm{Win}_1(\mathcal{G}_\mathcal{E}) = \mathrm{Win}_1(\alpha, V, A)$*;*

2. *If* α *starts with an odd parity set, it holds that* $\mathrm{Win}_0(\mathcal{G}_\mathcal{E}) = \mathrm{Win}_0(\alpha, V, A)$*;*

3. *If* α *starts with an even parity set, it holds that* $\mathrm{Win}_1(\mathcal{G}_\mathcal{E}) = \mathrm{Win}_1(\alpha, V, A)$*.*

Proof. The proof of Item 1 follows immediately by definition, as $\alpha = \epsilon$ forces the two players to reach their respective winning sets in one step.

For Item 2 and 3, we need to find a partition of F into a winning set for Player 0 and a winning set for Player 1 such that the game is invariant *w.r.t.* the winning sets, once they are moved to visiting and avoiding, respectively. We proceed by mutual induction on the length of the sequence α. As base case, assume $\alpha = F$ and F to be an odd parity set. Then, first observe that Player 0 can win only by eventually hitting the set V, as the parity condition is made by only odd numbers. We have that $\mathrm{Win}_0(F, V, A) = \mu Y(P \setminus \mathrm{Win}_1(\varepsilon, A \cup (F \setminus Y), V \cup (F \cap (Y)))) = \mu Y(P \setminus \mathsf{force}_1(A \cup (F \setminus Y)))$ that, by definition, computes the set from which Player 1 cannot avoid a visit to V, hence the winning set for Player 0. In the case the set F is an even parity set the reasoning is symmetric.

As an inductive step, assume that Items 2 and 3 hold for sequences α of length n, we prove that it holds also for sequences of the form $\mathrm{F} \cdot \alpha$ of length $n+1$. Suppose that F is a set of odd priority. Then, we have that, by induction hypothesis, the formula $\mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}))$ computes the winning set for Player 1 for the game in which the nodes in $\mathrm{F} \cap \mathrm{Y}$ are visiting, while the nodes in $\mathrm{F} \setminus \mathrm{Y}$ are avoiding. Thus, its complement $\mathrm{P} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}))$ returns the winning set for Player 0 in the same game. Now, observe that, if a set Y' is bigger than Y, then $\mathrm{P} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}'), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}'))$ is the winning set for Player 0 in which some node in $\mathrm{F} \setminus \mathrm{Y}$ has been moved from avoiding to visiting. Thus we have that $\mathrm{P} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y})) \subseteq \mathrm{P} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}'), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}'))$. Moreover, observe that, if a node $q \in \mathrm{F} \cup \mathrm{A}$ is winning for Player 0, then it can be avoided in all possible winning plays, and so it is winning also in the case q is only in F. It is not hard to see that, after the last iteration of the fixpoint operator, the two sets $\mathrm{F} \setminus \mathrm{Y}$ and $\mathrm{F} \cap \mathrm{Y}$ can be considered in avoiding and winning, respectively, in a way that the winning sets of the game are invariant under this update, which concludes the proof of Item 2.

Also in the inductive case, the reasoning for Item 3 is perfectly symmetric to the one for Item 2. □

4.1.1 Implementation of `APT` in `PGSolver`

In this section we describe the implementation of `APT` in the well-known platform `PGSolver` developed in OCaml by Friedman and Lange [70], which collects the large majority of the algorithms introduced in the literature to solve parity games [77, 89, 91, 143, 144, 161, 167].

We briefly recall the main aspects of this platform. The graph data structure is represented as a fixed length array of tuples. Every tuple has all information that a node needs, such as the owner player, the assigned priority and the adjacency list of nodes. The platform implements a collection of tools to generate and solve parity games, as well as compare the performance of different algorithms. The purpose of

Algorithm 3 The APT Algorithm

```
function WIN($\mathcal{G}$,$i$,$\alpha$,V,A)
    if ($\alpha \neq \epsilon$) then
        return MinFP($i$,$\alpha$,V,A)
    else
        return $\mathsf{force}_i(V)$
    function MinFP($i$,$\alpha$,V,A)
        $Y_1 = Y2 = \emptyset$
        $F$ =head[$\alpha$]
        $\alpha'$=tail[$\alpha$]
        do
            $Y_2 =$ P\WIN($\mathcal{G}$,$1 - i$,$\alpha'$,$V \cup (F \cap Y_1), A \cup (F \setminus Y_1)$)
            $Y_1 = Y_2$
        while $Y_1 \neq Y_2$
        return $Y_2$
```

this platform is not just that of making available an environment to deploy and test a generic solution algorithm, but also to investigate the practical aspects of the different algorithms on the different classes of parity games. Moreover, `PGSolver` implements optimizations that can be applied to all algorithms in order to improve their performance. The most useful optimizations in practice are decomposition into strongly connected components, removal of self-cycles on nodes, and priority compression.

We have added to `PGSolver` an implementation of the `APT` algorithm introduced above. Our procedure applies the fixpoint algorithm to compute the set of winning positions in the game by means of two principal functions that implement the two functions of the algorithm core processes, i.e., function force_i and the recursive function $\mathrm{Win}_i(\alpha, V, A)$. The pseudocode of the `APT` algorithm implementation is reported in Algorithm 3. It takes six parameters: the Player (0 or 1), the game, the set of nodes, the condition α, the set of visiting and avoiding. Moreover, we define the function *min_fp* for the calculation of the fixed point. The whole procedure makes use of Set and List data structures, which are available in the OCaml's standard library, for the manipulation of the sets visiting and avoiding, and the accepting condition α. The tool along with the implementation of the APT algorithm is available for download from https://github.com/antoniodistasio/pgsolver-APT.

For the sake of clarity, we report that in PGSolver it is used the maximal priority to decide who wins a given parity game. Conversely, the APT algorithm uses the minimal priority. However, these two conditions are well known to be equivalent and, in order to compare instances of the same game on different implementations of parity games algorithms in PGSolver, we simply convert the game to the specific algorithm accordingly. For the conversion, we simply use a suitable permutation of the priorities.

4.1.2 Experiments

In this section, we report the experimental results on evaluating the performance for the APT algorithm implemented in PGSolver over the random benchmarks generated in the platform. We have compared the performance of the implementation of APT with those of RE and SPM. We have chosen these two algorithms as they have been proved to be the best-performing in practice [70].

All tests have been run on an AMD Opteron 6308 @2.40GHz, with 224GB of RAM and 128GB of swap running Ubuntu 14.04. We note that APT has been executed without applying any optimization implemented in PGSolver [70], while SPM and RE are run with such optimizations. Applying these optimization on APT is a topic of further research.

We evaluated the performance of the three algorithms over a set of games that are randomly generated by PGSolver, in which it is possible to give the number n of states and the number k of priority as parameters. We have taken 20 different game instances for each set of parameters and used the average time among them returned by the tool. For each game, the generator works as follows. For each node

q in the graph-game, the priority $\mathsf{p}(q)$ is chosen uniformly between 0 and $k-1$, while its ownership is assigned to Player 0 with probability $\frac{1}{2}$, and to Player 1 with probability $\frac{1}{2}$. Then, for each node q, a number d from 1 to n is chosen uniformly and d distinct successors of q are randomly selected.

Experimental results

We ran two experiments. First, we tested games with 2, 3, and 5 priorities, where for each of them we measured runtime performance for different state-space sizes, ranging in $\{2000, 4000, 6000, 8000, 10000, 12000, 14000\}$. The results are in Table 4.1, in which the number of states is reported in column 1, the number of colors is reported in the macro-column 2, 3, and 5, each of them containing the runtime executions, expressed in seconds, for the three algorithms. Second, we evaluated the algorithms on games with an exponential number of nodes *w.r.t.* the number of priorities. More precisely, we ran experiments for $n = 2^k$, $n = e^k$ and $n = 10^k$, where n is the number of states and k is the number of priorities. The experiment results are reported in Table 4.2. By $\texttt{abort}_T$, we denote that the execution has been aborted due to time-out (greater of one hour), while by $\texttt{abort}_M$we denote that the execution has been aborted due to mem-out.

	2 Pr			3 Pr			5 Pr		
n	`RE`	`SPM`	`APT`	`RE`	`SPM`	`APT`	`RE`	`SPM`	`APT`
2000	4.94	5.05	0.10	4.85	5.20	0.15	4.47	4.75	0.42
4000	31.91	32.92	0.17	31.63	31.74	0.22	31.13	32.02	0.82
6000	107.06	108.67	0.29	100.61	102.87	0.35	100.81	101.04	1.39
8000	229.70	239.83	0.44	242.24	253.16	0.5	228.48	245.24	2.73
10000	429.24	443.42	0.61	482.27	501.20	0.85	449.26	464.36	3.61
12000	772.60	773.76	0.87	797.07	808.96	0.98	762.89	782.53	6.81
14000	1185.81	1242.56	1.09	1227.34	1245.39	1.15	1256.32	1292.80	10.02

Table 4.1: Runtime executions with fixed priorities 2, 3 and 5

The first experiment shows that with a fixed number of priorities (2, 3, and 5) APT significantly outperforms the other algorithms, showing excellent runtime execution even on fairly large instances. For example, for $n = 14000$, the running time for both RE and SPM is about 20 minutes, while for APT it is less than a minute.

The results of the exponential-scaling experiments, shown in Table 4.2, give more nuanced results.

n	Pr	RE	SPM	APT
		$n = 2^k$		
1024	10	1.25	1.25	8.58
2048	11	7.90	8.21	71.08
4096	12	52.29	52.32	1505.75
8192	13	359.29	372.16	abort$_T$
16384	14	2605.04	2609.29	abort$_T$
32768	15	abort$_T$	abort$_T$	abort$_T$
		$n = e^k$		
21	3	0	0	0
55	4	0	0	0.02
149	5	0.01	0.01	0.08
404	6	0.14	0.14	0.19
1097	7	1.72	1.72	0.62
2981	8	24.71	24.46	7.88
8104	9	413.2.34	414.65	35.78
22027	10	abort$_T$	abort$_T$	311.87
		$n = 10^k$		
10	1	0	0	0
100	2	0	0	0
1000	3	1.3	1.3	0.04
10000	4	738.86	718.24	4.91
100000	5	abort$_M$	abort$_M$	66.4

Table 4.2: Runtime executions with $n = e^k$ and $n = 2^k$ and $n = 10^k$.

Here, APT is the best performing algorithm for $n = e^k$ and $n = 10^k$. For example, when $n = 100000$ and $k = 5$, both RE and SPM memout, while APT completes in just over one minute. That is, the efficiency of APT is notable also in terms of memory usage. At the same APT underperforms for $n = 2^k$. Our conclusion is that APT has superior performance when the number of priorities is logarithmic in the number of game-graph nodes, but the base of the logarithm has to be large enough. As we see experimentally, e is sufficiently large base, but 2 is not. This point deserve further study, which we leave to future work. In Figure 4.2 we just report graphically the benchmarks in the case $n = e^k$.

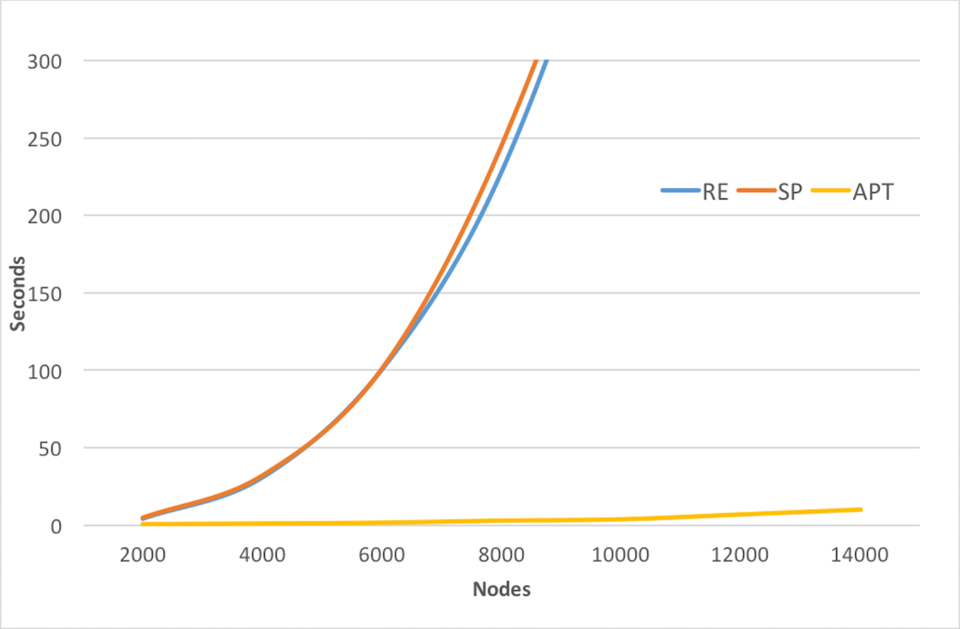


Figure 4.2: Runtime executions with $n = e^k$

4.2 Conclusion and Discussion

The APT algorithm, an automata-theoretic technique to solve parity games, has been designed two decades ago by Kupferman and Vardi [104], but never considered to be useful in practice [69]. In this work, for the first time, we fill missing gaps and implement this algorithm. By means of benchmarks based on random games, we show that it is the best performing algorithm for solving parity games when the number of priorities is very small *w.r.t.* the number of states. We believe that this is a significant result as several applications of parity games to formal verification and synthesis do yield games with a very small number of priorities.

The specific setting of a small number of priorities opens up opportunities for specialized optimization technique, which we aim to investigate in future work. This is closely related to the issue of accelerated algorithms for three-color parity games [50]. We also plan to study why the performance of the `APT` algorithm is so sensitive to the relative number of priorities, as shown in Table 4.2.

Chapter 5

Symbolic Parity Games

In this chapter we provide a broad investigation of the symbolic approach for solving parity game. We implement four symbolic algorithms and compare their performances to the corresponding explicit versions for different classes of parity games. Specifically, we implement in a new tool, called `SymPGSolver`, the symbolic versions of `RE`, `APT`, and two variants of `SPM`. The tool also allows to generate random games, as well as compare the performance of different symbolic algorithms.

5.1 Definition

We start with some notation. In the sequel we use symbols x_i for propositions (variables), l_i for literals, *i.e.*, positive or negative variables, f for a generic Boolean formula, $||f||$ represents the set of interpretations that makes the formula f true, $\lambda(f) \subseteq V$ for the set of variables in f, and $Prime(f)$ for the formula $f[x_1, \ldots, x_n / x'_1, \ldots, x'_n]$, that is, the formula where x_i is replaced with x'_i, for $1 \leq i \leq n$.

Definition 2. *Given a parity game (PG, for short)* $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p}, \alpha)$*, the corresponding* symbolic PG *(SPG, for short) is the tuple* $\mathcal{F} = (\mathcal{X}, \mathcal{X}_M, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}}, \alpha)$ *defined as follows:*

– $\mathcal{X} = \{x_1, \dots, x_n\}$, *with* $n = \lceil log_2(|\mathrm{P}|)\rceil$, *is the set of* propositions *used to encode nodes in* $\mathcal{G}$, *i.e., to each* $v \in \mathrm{P}$ *we associate a Boolean formula* $f_v = l_{v,1} \wedge ... \wedge l_{v,n}$ *where* $l_{v,i}$ *is either* x_i *or* $\overline{x_i}$. *We also associate to* v *the interpretation* $\mathcal{X}_v \in 2^{\mathcal{X}}$, *i.e., the subset of variables appearing positively in* f_v.

– $\mathcal{X}_M = \{x'_1, ..., x'_n\}$, *with* $n = \lceil log_2(|\mathrm{P}|)\rceil$, *is the set of propositions used to encode the successor nodes such that* $\mathcal{X} \cap \mathcal{X}_M = \emptyset$. *We extend to* $\mathcal{X}_M$ *the definitions of* f_v *and* X_v *as used in the previous item.*

– f_{P_i}, *for* $i \in \{0, 1\}$, *is a Boolean formula such that* $||f_{\mathrm{P}_i}||$=P_i.

– f_{Mv} *is a Boolean formula over the propositions* $\mathcal{X} \cup \mathcal{X}_M$ *such that* $||f_{Mv}||$=Mv.

– η_{p} *is the symbolic representation of the priority function* p; *it is formally given by the function* $\eta_{\mathsf{p}} : 2^{\mathcal{X}} \to \mathbb{N}$ *that associates with each interpretation* X_v *a natural number.*

To solve an SPG we compute the Boolean formulas f_{Win_0} over $\mathcal{X}$ that is satisfied by those interpretations that correspond to winning nodes for Player 0.

For technical reasons, we also need the definition of symbolic sub-games.

Definition 3. *Let* $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p}, \alpha)$ *be a PG and* $U \subseteq \mathrm{P}$. *By* $\mathcal{G}[U] = (\mathrm{P}_0 \setminus U, \mathrm{P}_1 \setminus U, Mv \setminus (U \times \mathrm{P} \cup \mathrm{P} \times U), \mathsf{p}_{|\mathrm{P}\setminus U})$ *we denote the PG restricted to nodes* $\mathrm{P} \setminus U$.

Let f_U *be a Boolean formula such that* $||f_U|| = U$ *and* $\mathcal{F} = (\mathcal{X}, \mathcal{X}_M, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}})$ *be the corresponding SPG of the PG* $\mathcal{G}$. *By* $\mathcal{F}_{\mathrm{P}\setminus U} = (\mathcal{X}, \mathcal{X}_M, f'_{\mathrm{P}_0}, f'_{\mathrm{P}_1}, f'_{Mv}, \eta'_{\mathsf{p}})$ *we denote the SPG of* $\mathcal{G}[U]$, *where:*

– $f'_{\mathrm{P}_i} = f_{\mathrm{P}_i} \wedge \neg f_U$, *for* $i \in \{0, 1\}$, *is the Boolean formula for nodes* $v \in \mathrm{P}_i \setminus U$;

– $f_{Mv'} = f_{Mv} \wedge \neg(f_U \vee f'_U)$, *where* $||f'_U|| = U$ *and* $\lambda(f'_U) = \mathcal{X}_M$, *is the Boolean formula representing moves restricted to* $Mv \setminus (U \times \mathrm{P} \cup \mathrm{P} \times U)$;

– $\eta'_{\mathsf{p}} = 2^{\mathcal{X}} \to \mathbb{N}$ *is the symbolic representation of* $\mathsf{p}_{|\mathrm{P}\setminus U}$ *that associates to the interpretations* X_v *satisfying the Boolean formula* $f_{\mathrm{P}} \wedge \neg f_U$ *a natural number.*

5.2 Symbolic Algorithms

We now describe symbolic versions of the explicit algorithms listed in Chapter 2. To do this, we first give a brief overview to Binary Decision Diagrams and Algebraic Decision Diagrams. Note that, we keep using the notions introduced previously.

5.2.1 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [1,26] are a compact way to represent function of the form $f : \mathbb{B} \to \{0, 1\}$. A BDD is defined as a directed acyclic graph (DAG), with nonterminal nodes and terminal nodes. Each nonterminal node is labeled by a Boolean variable *var(v)* and has two successors: *low(v)* corresponding to the case where the variable v is assigned to 0, and *high(v)* corresponding to the case where the variable v is assigned to 1. Each terminal node is labeled by *value(v)* which are either *true* or *false*. Given a BDD representing a function f, whether a truth assignment to the variables makes f true or false can be decided by traversing the DAG from the root to a terminal node, taking appropriate edges forward based on $value(v)$. The value of the terminal node will be the value of f.

One of the major features in using BDDs is the possibility to have a canonical representation for Boolean functions. In [26] is shown how to obtain a canonical representation for Boolean functions by repeatedly applying the following three transformation rules.

– Share identical terminal nodes: eliminate all but one terminal node with a given label and redirect all edges to the eliminated nodes to the remaining one.

– Share identical non terminal nodes: if two nonterminals v and u have $var(v) = var(u)$, $low(v) = high(u)$ and $high(v) = low(u)$, then eliminate v or u and redirect all incoming edges to the other node.

– Remove redundant test: if non terminal v has $low(v) = high(v)$, then eliminate v and redirect all incoming edges to $low(v)$.

And, by imposing a total order on the variables that label the nodes in the BDD. Such BDDs are called *ordered binary decision diagrams.* The above rules are applied until the size of the diagram can no longer be reduced, and the diagram so obtained is called *reduced ordered binary decision diagram* (ROBDD). Consequences of the canonicity are that one can check tautology or satisfiability of a f by looking at whether its resulting ROBDD is either 1 or 0, respectively. Moreover, the checking equivalence of two functions is reduced to checking if their ROBDDs are equals.

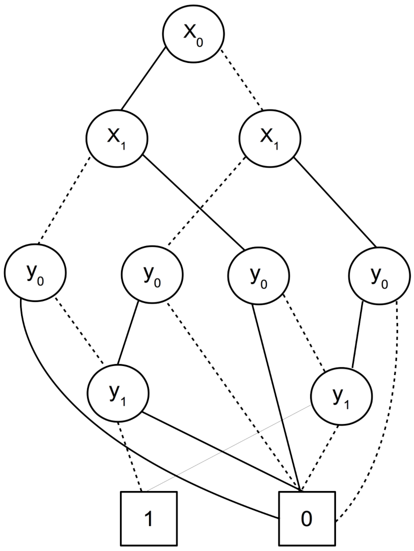


Figure 5.1: The ROBDD for $(x_0 \Leftrightarrow y_0) \wedge (x_1 \Leftrightarrow y_1)$.

Figure 5.1 illustrates an example of BDD. The $high(v)$ and $low(v)$ edges from a nonterminal are shown as solid and dashed lines, and represent the variable's true and false respectively. The leaves store the Boolean values 0 and 1.

Variable ordering plays a fundamental role in determining the size of an OBDD. When using OBDDs, it is crucial to select a good order of the variables in order to build a small OBDD, but finding an optimal order is an NP-Complete problem [22]. Moreover, there are Boolean functions that have an exponential size OBDDs for any variable ordering. Several heuristics have been proposed for finding a good variable ordering when such an order exists [56, 83, 125, 126, 140, 164].

5.2.2 Algebraic Decision Diagrams

Algebraic Decision Diagrams (ADDs) [12] were introduced to extend BDDs by allowing values from any arbitrary finite domain to be associated with the terminal nodes of the diagram, and then to represent in a compact way functions of the form $f : \mathbb{B} \rightarrow \mathbb{R}$. An ADD can be seen as a BDD whose leaves may take on values belonging to a set of constants different from 0 and 1. Hence, ADDs are defined similarly to BDDs, and the canonical representation is obtained by applying the rules described in the previous section. Similar to BDDs, given an ADD representing a function f, the function's value associated to a truth assignment to the variables can be decided by traversing the DAG from the root node to a terminal node taking appropriate edges forward based on $value(v)$. The resulting leaf node represents the function's value. For example, in Figure 5.2, the assignment $(x_0 = 1, x_1 = 1, y_0 = 0, y_1 = 0)$ leads to a leaf node labeled 5.

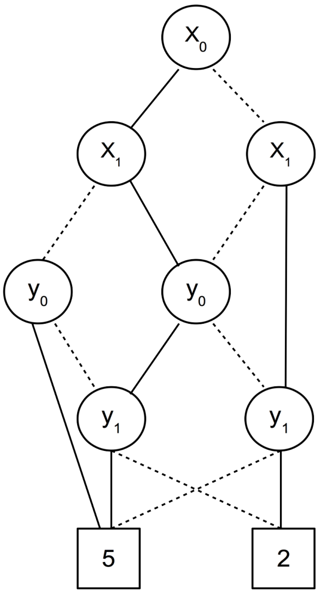


Figure 5.2: Example ADD

The key benefit of an ADD is, other to provide a compact representation, the existence of efficient algorithms for function manipulation and composition. For example operations like addition, multiplication, max, min, etc., can be performed directly within the ADD graph structures.

Applications of ADDs include computations on very big matrices including computing steady-state probabilities of Markov chains, probabilistic verification, and AI planning. Finally, because BDDs and ADDs have been used extensively in many domains, very efficient implementations are readily available in the CUDD package. Most of the popular algorithms have been tested using CUDD.

5.2.3 SPG Implementation

An SPG can be implemented using BDDs and ADDs to represent and manipulate the associated Boolean functions introduced along with its definition.

Given an SPG $\mathcal{F} = (\mathcal{X}_1, \mathcal{X}_M, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}})$ with maximal priority c, we use BDDs to represent the Boolean formulas f_{P_0}, f_{P_1} and f_{Mv}, and an ADD for the function η_{p}. Moreover, we decompose the function η_{p} into a sequence of BDDs $\mathcal{B} = \langle B_0, \dots, B_{c-1} \rangle$ where each B_i encodes the nodes with priority i, to easily manage the selection of a set of nodes with a specific priority. In the sequel, by BDD (*resp.*, ADD) *f*, we denote the BDD (*resp.*, ADD) representing the function *f*.

5.2.4 Symbolic `SPM` (`SSP`)

This is the first symbolic implementation of `SPM` we are aware of, and which we describe with some minor corrections compared to the one in [29]. Lift is encoded by using ADDs and the algorithm computes the progress measure as the least fixed point f_G of $\mathrm{Lift}(f, v)$ on a ranking function here given by the function $f : \mathrm{P} \to D$, with $D = M_G \cup \{\infty, -\infty\}$. The algorithm takes as input an SPG $\mathcal{F}$ and returns an ADD representing the least fixed point f_G such that the set of winning nodes for Player 0 is $\{v | f_G(v) < \infty\}$, and the set of winning nodes for Player 1 is $\{v | f_G(v) = \infty\}$. The symbolic implementation of SPM is reported in Algorithm 4.

Algorithm 4 Symbolic Small Progress Measures

```
procedure PARITY ($\mathcal{F}$)
    $f = \rightarrow (f_{\mathrm{P}}, -\infty)$;
    repeat
        $f_{old} = f$; $f =$ false;
        for $j = 0$ to $c - 1$ do
            $f = f$ OR MAXeo($f_{old}, j$) OR MINeo($f_{old}, j$);
    until $f = f_{old}$
```

The algorithm calls the following procedures:

- `MAXeo`, which given an ADD $f : \mathrm{P} \rightarrow D$, the BDD f_{Mv}, and $1 \leq j \leq k$, returns an ADD that assigns to every vertex $v \in \mathrm{P}_1$, with $\mathsf{p}(v) = j$, the value $\mathrm{inc}_j(\max\{f(v')|(v, v') \in Mv\})$;
- `MINeo`, which given an ADD $f : \mathrm{P} \rightarrow D$, the BDD f_{Mv}, and $1 \leq j \leq k$, returns an ADD that assigns to every vertex $v \in \mathrm{P}_0$, with $\mathsf{p}(v) = j$, the value $\mathrm{inc}_j(\min\{f(v')|(v, v') \in Mv\})$.

`MINeo` (*resp.,* `MAXeo`) aims at constructing an ADD that represents the ranking function $f_{\min}(v) = \min\{f(v')|(v, v') \in Mv\}$ (*resp.,* $f_{\max}(v) = \max\{f(v')|(v, v') \in Mv\}$). To do this, given an ADD $f : \mathrm{P} \rightarrow D$ and the BDD f_{Mv}, it is generated an ADD $f_{suc} : (\mathrm{P} \times \mathrm{P}) \rightarrow D$ such that $f_{suc}(v, v') = d$ if $(v, v') \in Mv$ and $f(v') = d$. Then, the ADD f_{suc} is given in input to the procedure `MIN`, described in Algorithm 5, that constructs the ADD for f_{min}. The procedure `MAX` is defined similarly. Let n be an ADD node, we refer to the left and right successors of n as $n.l$ and $n.r$, respectively, and refer to the variable that n represents as $n.v$. For a variable v, let $o(v)$ be the position of v in the BDD order. Since Mv may be a strict subset of $\mathrm{P} \times \mathrm{P}$, the function f_{suc} is not defined for all the pairs in $\mathrm{P} \times \mathrm{P}$. Thus, one leaf in the ADD f_{suc} stands for the value of the pairs from which f_{suc} is undefined, and we assume that is value is ∞. Since every node has at least one successor, the ADD f_{min} is defined for all nodes of P.

Algorithm 5 Procedure `MIN`

```
procedure MIN(ADD $n$)
    if $n$ is a terminal node then
        return $n$
    if $n.v$ is in $\mathcal{X}$ then
        return ($n.v$ AND MIN($n.r$)) OR ( NOT $n.v$ AND MIN($n.l$))
    if $n.v$ is in $\mathcal{X}'$ then
        return MERGE(MIN($n.r$), MIN($n.l$)))
```

The procedure `MIN` calls the procedure `MERGE`, reported in Algorithm 6, that gets in input the pointer to the roots n_1 and n_2 of two ADDs representing the functions f_1 and f_2, both from some set $U \subseteq \mathrm{P}$ to D, and merges them to an ADD in which every $u \in U$ is mapped into $\min(f_1(u), f_2(u))$.

Algorithm 6 Procedure `MERGE`

```
procedure MERGE(ADD $n_1$, ADD $n_2$)
    if $n_1$ and $n_2$ are a terminal nodes then
        return $\min(n1, n2)$
    if $o(n_1.v) < o(n_2.v)$ then
        return ($n_1.v$ AND MERGE($n_1.r, n_2$)) OR ( NOT $n_1.v$ AND MERGE($n_1.l, n_2$))
    if $o(n_1.v) > o(n_2.v)$ then
        return ($n_2.v$ AND MERGE($n_2.r, n_1$)) OR ( NOT $n_2.v$ AND MERGE($n_2.l, n_1$))
    return ($n_1.v$ AND MERGE($n_1.r, n_2.r$)) OR ( NOT $n_1.v$ AND MERGE($n_1.l, n_2.l$))
```

5.2.5 Set-Based Symbolic `SPM` (`SSP2`)

This is a symbolic implementation of `SPM` that has been introduced recently in [35]. It allows to use only basic set operations like $\cup$, $\cap$, $\setminus$, $\subseteq$, and one-step predecessor operations for its description. Unlike the implementation described previously, the ranking function is implicitly encoded by using sets of nodes. This allows representing the *Lift* operator just by BDDs.

To encode the ranking function the algorithm defines for each rank $r \in M_G^\top$ the set S_r containing the nodes with rank r or higher. Formally, given the ranking function $\varrho : \mathrm{P} \to M_G^\top$, the corresponding sets are defined as $S_r = \{v|\varrho(v) \geq r\}$. Conversely, given the family of sets $\{S_r\}_r$, the corresponding ranking function, say $\varrho_{\{S_r\}_r}$, is given by $\varrho_{\{S_r\}_r}(v) = \max\{r \in M_G^\top|v \in S_r\}$. This formulation encodes the ranking function with sets but uses exponential in c many sets.

Space is reduced to a linear number of sets by encoding the value of each coordinate of the rank r, separately. In detail, for each odd priority i, the algorithm defines the sets $C_0^i, \dots, C_{n_i}^i$. Each set C_x^i with $x \in \{0, \dots, n_i\}$ contains the nodes that have x as i-th coordinate of their rank. Therefore, the algorithm has to construct the set S_r whenever it needs it.

Let $Cpre_i(\mathrm{X}) = \{q \in \mathrm{P}_i : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{P}_{1-i} : \mathrm{X} \subseteq Mv(q)\}$ the one-step controllable predecessor operator, *i.e.*, the $Cpre_i(\mathrm{X})$ operator computes all nodes from which Player i can ensure that in the next step the successor belongs to the given set X. The algorithm starts initializing the sets S_r for $r > 0$ to empty, and S_0 with the set of all nodes P. The rank r initially is set to the second lowest rank $\mathrm{inc}((0, \dots, 0))$. Then, at each iteration the set S_r is updated for the current value of r by using the *Lift* encoded by the $Cpre_i$ operator. After the update of S_r, it is checked if $S_r' \supseteq S_r$ for all $r' < r$, *i.e.*, if the property of the *anti-monotonicity* is preserved. Anti-monotonicity together with the definition of the sets S_r' allows to decide whether the rank of a node v can be increased to r by only considering one set S_r'. If the anti-monotonicity is preserved, then for $r < \top$ the value of r is increased to the next highest rank and for $r = \top$ the algorithm terminates. Otherwise the nodes newly added to S_r are also added to all sets with $r' < r$ that do not already contain them; the variable r is then updated to the lowest r' for which a new node is added to S_r' in this iteration. The symbolic implementation is reported in Algorithm 7. Note that, the parity condition α is defined as a finite sequence $\mathrm{F}_1 \cdot \ldots \cdot \mathrm{F}_k$, with $\mathrm{F}_i = \mathsf{p}^{-1}(i)$.

Algorithm 7 Symbolic Progress Measures Algorithm

```
    **Input**: Parity game $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p}, \alpha)$.
    **Output**: Winning set for Player 0.
function SymbolicSPM($\mathcal{G}$)
    $S_{\overline{0}} \leftarrow \mathrm{P}$; $S_r \leftarrow \emptyset$ for $r \in M_G^{\top}$;
    $r \leftarrow \mathrm{inc}(\overline{0})$;
    while *true* do
        if $r \neq \top$ then
            Let $l$ maximal such that $r = \langle r \rangle_L$;
            $S_r \leftarrow S_r \cup \bigcup_{1 \leq k \leq (l+1)/2} (\mathrm{CPre}_1(S_{\mathrm{dec}_{2k-1}(r)}) \cap F_{2k-1})$;
            repeat
                $S_r \leftarrow S_r \cup (CPre_1(S_r) \setminus \bigcup_{l \leq k \leq c} P_k)$;
            until *a fixed-point for $S_r$ is reached*;
        else if $r = \top$ then
            $S_\top \leftarrow S_\top \cup \bigcup_{1 \leq k \leq \lfloor c/2 \rfloor} (\mathrm{CPre}_1(S_{\mathrm{dec}_{2k-1}(\top)}) \cap F_{2k-1})$;
            repeat
                $S_\top \leftarrow S_\top \cup (CPre_1(S_\top)$;
            until *a fixed-point for $S_\top$ is reached*;
        $r' \leftarrow \mathrm{dec}(r)$;
        if $S'_r \supseteq S_r$ and $r < \top$ then
            break
        else
            repeat
                $S'_r \leftarrow S'_r \cup S_r$;
                $r' \leftarrow \mathrm{dec}(r')$;
            until $S'_r \supseteq S_r$;
            $r' \leftarrow \mathrm{inc}(r')$;
    return $\mathrm{P} \setminus S_\top$
```

5.2.6 Symbolic versions of RE (SRE) and APT (SAPT).

RE and APT can be easily rephrased symbolically by using BDDs to represent the operations they make use of: set basic operations like union, intersection, complement, and inclusion. The controllable predecessor operator used to implement the function force_i in APT and the attractor in RE can be encoded using existential quantification as follows.

$$CPre_i(f_U) = (f_{\mathrm{P}_i} \wedge \exists \mathcal{X}_M . f_{Mv} \wedge Prime(f_U)) \vee (f_{\mathrm{P}_{1-i}} \wedge \neg\exists \mathcal{X}_M . f_{Mv} \wedge \neg Prime(f_U))$$

That is, the BDD for f_U is first renamed to be over $\mathcal{X}_M$, then it is conjoined with the BDD f_{Mv}, and finally all variables in $\mathcal{X}_M$ are quantified existentially.

The symbolic construction of a subgame used in RE is implemented following the definition of symbolic subgame reported previously.

5.3 Experimental Evaluations: Methodology and Results

We now analyze the performance of the introduced symbolic approach to solve PGs and compare with the explicit one. We have implemented the symbolic algorithms described in Section 5.2 in a fresh platform tool, called SymPGSolver(Symbolic Parity Games Solver). SymPGSolver[1] is implemented in C++ and uses the CUDD[2] package as the underlying BDD and ADD library. The platform provides a collection of tools to randomly generate and solve SPGs, as well as compare the performance of different symbolic algorithms.

We have also compared them with *Oink*, a platform recently developed in C++ by Tom van Dijk [159], which collects the large majority of explicit PGs algorithms introduced in the literature [30, 89, 90, 167].

[1] The tool is available for download from https://github.com/antoniodistasio/sympgsolver

[2] http://vlsi.colorado.edu/ fabio/CUDD/

5.3.1 Experimental results

In this section we report on some experimental results on evaluating the performance for the explicit algorithms RE, APT, and SPM as well as their corresponding symbolic versions SRE, SAPT, SSP and SSP2 [35]. All tests have been run on an Intel Core i7 @2.40GHz, with 16GB of RAM running macOS 10.12. We have used different classes of parity games: random games with linear structures, ladder games, clique games as well as games corresponding to practical model checking problems. Random games are generated by SymPGSolver, while for ladder and clique games we use *Oink*. We have taken 100 different instances for each class of games and used the average time execution. In all tests, we use $\texttt{abort}_T$ to denote an aborted execution due to time-out (greater than 200 seconds). On the class of ladder games and in model checking problems the benchmarks have been executed using the variable ordering given by the heuristic WINDOW2 module available in the CUDD package.

Random Games with linear structure

Tabakov and Vardi showed that in the context of automata-theoretic problems, explicit algorithms generally dominate symbolic algorithms, as BDDs do not offer any compression for random sets [155]. We found that the same holds for parity-game solving (we omit details due to lack of space). In [155] it was observed that, in case of random games with linear structures, the symbolic algorithms are the best performing ones. Hence, we have investigated the same class here as well, but with a different outcome.

A random game with *linear structure* is built by restricting the transition relation as follows: a node v_i can make a transition to node v_j, where $0 \leq i, j \leq |\mathrm{P}| - 1$, if and only if $|i - j| \leq d$, where d is named as the *distance* parameter.

Table 5.1 collects the running time of the symbolic algorithms on random games with linear structures having priorities 2, 3, and 5, and distance $d = 25$. The results show that SAPT performs better thasolving parity games using an automata based algorithmn the others in solving games with $n \leq 10,000$ nodes and 2 priorities,

	2 Pr				3 Pr				5 Pr			
n	SRE	SAPT	SSP	SSP2	SRE	SAPT	SSP	SSP2	SRE	SAPT	SSP	SSP2
1,000	0.04	0.03	29.89	0,95	0.05	0.10	18.9	1,44	0.05	0.45	15.75	abort$_T$
2,000	0.14	0.12	128.06	2,87	0.13	0.18	79.22	26,24	0.12	1.34	69.6	abort$_T$
3,000	0.25	0.23	abort$_T$	10,15	0.21	0.41	193.06	75,49	0.21	2.03	135.04	abort$_T$
4,000	0.33	0.30	abort$_T$	32,42	0.28	0.60	abort$_T$	146,58	0.3	3.01	abort$_T$	abort$_T$
7,000	0.79	0.73	abort$_T$	abort$_T$	0.65	1.44	abort$_T$	abort$_T$	0.59	7.20	abort$_T$	abort$_T$
10,000	1.16	1.12	abort$_T$	abort$_T$	0.93	2.19	abort$_T$	abort$_T$	1.08	11.72	abort$_T$	abort$_T$
20,000	2.78	3.10	abort$_T$	abort$_T$	2.33	6.34	abort$_T$	abort$_T$	3.69	43.87	abort$_T$	abort$_T$
100,000	19.21	24.4	abort$_T$	abort$_T$	24.38	65.11	abort$_T$	abort$_T$	24.89	abort$_T$	abort$_T$	abort$_T$

Table 5.1: Runtime executions of the symbolic algorithms

while SRE is the best performing in all other cases. Also, they show that SSP and SSP2 have the worst performances in all instances, with SSP overcoming SSP2 of more than 200 seconds on games with $3,000$ nodes. In Table 5.2 we collect the execution time of the explicit algorithms on the same set of games. The results highlight that the explicit algorithms are faster than the symbolic ones in all instances.

	2 Pr			3 Pr			5 Pr		
n	RE	APT	SPM	RE	APT	SPM	RE	APT	SPM
1,000	0.0008	0.0006	0.0043	0.0008	0.0007	0.0049	0.0008	0.0008	0.0053
2,000	0.0015	0.0012	0.0084	0.0017	0.0016	0.0096	0.0019	0.0029	0.011
3,000	0.0023	0.0017	0.012	0.0025	0.0022	0.014	0.0029	0.0073	0.020
4,000	0.0031	0.0022	0.016	0.0033	0.0028	0.019	0.0035	0.0066	0.027
7,000	0.0051	0.0039	0.025	0.0053	0.0048	0.032	0.0056	0.012	0.039
10,000	0.0065	0.0057	0.035	0.0067	0.0076	0.046	0.0069	0.018	0.051
20,000	0.013	0.011	0.078	0.014	0.021	8.32	0.17	0.019	107.2
100,000	0.094	0.081	0.44	0.099	0.10	1.47	0.10	0.59	80.37

Table 5.2: Runtime executions of the explicit algorithms

Ladder Games

In a ladder game, every node in P_i has priority i. In addition, each node $v \in \mathrm{P}$ has two successors: one in P_0 and one in P_1, which form a node pair. Every pair is connected to the next pair forming a ladder of pairs. Finally, the last pair is connected to the top. The parameter m specifies the number of node pairs. Formally, a ladder game of index m is $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p})$ where $\mathrm{P}_0 = \{0, 2, \ldots, 2m-2\}$, $\mathrm{P}_1 = \{1, 3, \ldots, 2m-1\}$, $Mv = \{(v, w) | w \equiv_{2m} v + i \text{ for } i \in \{1, 2\}\}$, and $\mathsf{p}(v) = v \ mod \ 2$. Table 5.3 reports the benchmarks.

m	SRE	SAPT	SSP	SSP2
1,000	0	0.00013	24.86	0.47
10,000	0.00009	0.00016	abort$_T$	41.22
100,000	0.0001	0.00018	abort$_T$	abort$_T$
1,000,000	0.00012	0.00022	abort$_T$	abort$_T$
10,000,000	0.00015	0.00025	abort$_T$	abort$_T$

m	RE	APT	SPM
1,000	0.0007	0.0006	0.002
10,000	0.006	0.005	0.0017
100,000	0.057	0.054	0.18
1,000,000	0.59	0.56	1.84
10,000,000	6.31	5.02	20.83

Table 5.3: Runtime executions of the explicit and symbolic algorithms on ladder games.

The benchmarks indicate that SRE and SAPT outperform their explicit versions, showing an excellent runtime execution even on fairly large instances. Indeed, while RE needs 6.31 seconds for games with index $m = 10M$, SRE takes just 0.00015 seconds. Benchmarks also show that SSP and SSP2 have yet the worst performance.

Clique Games

Clique games are fully connected games without self-loops, where P_0 (*resp.,* P_1) contains the nodes with an even index (*resp., odd*) and each node $v \in \mathrm{P}$ has as priority the index of v. An important feature of the clique games is the high number of cycles, which may pose difficulties for certain algorithms. Formally, a clique game of index n is $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p})$ where $\mathrm{P}_0 = \{0, 2, \ldots, n-2\}$, $\mathrm{P}_1 = \{1, 3, \ldots, n-1\}$, $Mv = \{(v, w) | v \neq w\}$, and $\mathsf{p}(v) = v$. Benchmarks on clique games are reported in Table 5.4.

n	SRE	SAPT	SSP	SSP2
2,000	0.007	0.003	5.53	abort$_T$
4,000	0.018	0.008	19.27	abort$_T$
6,000	0.025	0.012	39.72	abort$_T$
8,000	0.037	0.017	76.23	abort$_T$

n	RE	APT	SPM
2,000	0.021	0.0105	0.0104
4,000	0.082	0.055	0.055
6,000	0.19	0.21	0.22
8,000	0.35	0.59	0.63

Table 5.4: Runtime executions of the explicit and symbolic algorithms on clique games.

Benchmarks show that `SAPT` is the best one among the symbolic algorithms in all instances, `SAPT` and `SRE` outperform the explicit ones (as in ladder games), and the symbolic versions of `SPM` do not show good results even on small games.

Finally, we evaluate the symbolic and explicit approaches on some practical model checking problems as in [94]. Specifically, we use models coming from: the Sliding Window Protocol (SWP) with window size (WS) of 2 and 4 (WS represents the boundary of the total number of packets to be acknowledged by the receiver), the Onebit Protocol (OP), and the Lifting Truck (Lift). The properties we check on these models concern: absence of deadlock (ND), a message of a certain type (d1) is received infinitely often (IORD1), if there are infinitely many read steps then there are infinitely many write steps (IORW), liveness, and safety. Note that, in all benchmarks, data size (DS) denotes the number of messages.

n	Pr	Property	SRE	SAPT	SSP	SSP2	RE	APT	SPM	WS	DS
14,065	3	ND	0.00009	0.00006	3.30	0.0001	0.004	0.004	0.029	2	2
17,810	3	IORD1	0.0003	0.0005	abort$_T$	85.4	0.006	0.006	0.037	2	2
34,673	3	IORW	0.0006	0.0008	164.73	56.44	0.015	0.014	0.053	2	2
2,589,056	3	ND	0.0002	abort$_T$	abort$_T$	0.29	1.02	0.93	9.09	4	2
3,487,731	3	IORD1	abort$_T$	abort$_T$	abort$_T$	abort$_T$	1.81	1.4	17.45	4	2
6,823,296	3	IORW	0.3	abort$_T$	abort$_T$	abort$_T$	3.87	3.13	22.26	4	2

Table 5.5: SWP (Sliding Window Protocol)

As we can see, by comparing Tables 5.5, 5.6, and 5.7, the experiments indicate more nuanced relationship between the symbolic and explicit approaches. Indeed,

n	Pr	Property	SRE	SAPT	SSP	SSP2	RE	APT	SPM	DS
81,920	3	ND	0.00002	31.69	1.37	0.0016	0.031	0.034	0.22	2
88,833	3	IORD1	0.0027	0.003	$\texttt{abort}_T$	$\texttt{abort}_T$	0.036	0.0038	0.27	2
170,752	3	IORW	14.37	98.4	$\texttt{abort}_T$	$\texttt{abort}_T$	0.07	0.07	0.47	2
289,297	3	ND	0.0001	154.89	12.3	0.0058	0.13	0.12	1.34	4
308,737	3	IORD1	0.0088	0.009	$\texttt{abort}_T$	$\texttt{abort}_T$	0.14	0.13	1.37	4
607,753	3	IORW	43.7	$\texttt{abort}_T$	$\texttt{abort}_T$	$\texttt{abort}_T$	0.29	0.27	2.06	4

Table 5.6: OP (Onebit Protocol)

n	Pr	Property	SRE	SAPT	SSP	SSP2	RE	APT	SPM	DS
328	1	ND	0.00002	0.002	0.005	0.00002	0.0001	0.0001	0.0004	2
308	1	safety	0.00002	0.003	0.028	0.00002	0.0001	0.0001	0.0004	2
655	3	liveness	0.00008	0.0001	5.52	0.09	0.0003	0.0002	0.001	2
51.220	1	safety	0.0001	1.48	32.14	0.00002	0.01	0.01	0.09	4
53.638	1	ND	0.0001	0.2	4.67	0.0001	0.017	0.015	0.07	4
107,275	3	liveness	0.005	0.001	$\texttt{abort}_T$	$\texttt{abort}_T$	0.03	0.03	0.18	4

Table 5.7: Lift (Lifting Truck)

they show a different behavior depending on the protocol and the property we are checking. Overall, we note that SRE outperforms the other symbolic algorithms in all protocols, although the advantage over RE is discontinued. Specifically, SRE is the best performing in checking absence of deadlock in all three protocols, but for IORD1 in the SWP protocol with $WS = 2$, or for IORW in the OP protocol, RE exhibits a significant advantage. Differently, SAPT and SSP2 show better performances on a smaller number of properties. Moreover, the results highlights that SSP exhibits the worst performances in all protocols and properties.

5.4 Conclusion and Discussion

I we have compared for the first time the performances of different symbolic and explicit versions of classic algorithms to solve parity games. To this aim we have implemented in a fresh tool, which we have called `SymPGSolver`, the symbolic versions of Recursive [167], `APT` [104,153], and the small-progress-measures algorithms presented in [29] and [35].

Our analysis started from constrained random games [155]. The results show that on these games the explicit approach is better than the symbolic one, exhibiting a different behavior than the one showed in [155]. To gain a fuller understanding of the performances of the symbolic and the explicit algorithms, we have further tested the two approaches on structured games. Precisely, we have considered ladder games, clique games, as well as game models coming from practical model-checking problems. We have showed several cases in which the symbolic algorithms have the advantage over the explicit ones.

Our empirical study let us to conclude that on comparing explicit and symbolic algorithms for solving parity games, it would be useful to have real scenarios and not only random games, as the common practice has been.

Chapter 6

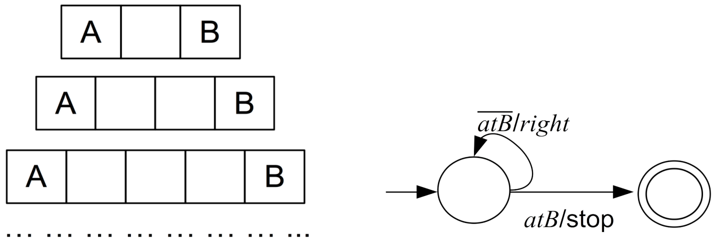
LTL Based Automated Planning

In this chapter we analyze an LTL based automated planning. In particular we introduce a generalized form of planning under partial observability, in which there are multiple, possibly infinitely many, planning domains with the same actions and observations, and goals expressed over observations, which are possibly temporally extended. The chapter starts with the definition of generalized planning, following the description in [81]. Then, we give a definition of generalized planning games, and finally we introduce a sound and complete mathematical technique for removing imperfect information from them.

Notation. We start with some notation. The positive integers are denoted $\mathbb{N}$, and $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. Write $\mathsf{2}$ for the set of Boolean values $\{\mathsf{true}, \mathsf{false}\}$, and write $\langle b \rangle$ to denote a vector of boolean values. For $n \in \mathbb{N}_0$, write $[n]$ for the set $\{0, 1, \cdots, n\}$. If π is a sequence, we write $\pi[i]$ for the ith element of π (here $i \in \mathbb{N}_0$; thus we start counting with the zero'th element), and $\pi[i, j]$ for the subsequence starting with the ith element and ending with the jth element. We write X^ω for the set of infinite sequences whose elements are from X, X^* for the finite sequences, and X^+ for the finite non-empty sequences (X a set). If π is a finite sequence then $Last(\pi)$ denotes its last element.

6.1 Generalized Planning

Informally, the problem of generalized planning is to find plans that can solve a set of problem instances. We also can rephrase it as the task of synthesizing a plan that works for multiple, possibly, infinitely many, cases [149]. This problem has been studied since the earliest work on STRIPS [66], and the fundamental motivations behind it stem from classical planning itself. Consider the simple planning problem depicted in Figure 6.1, taken from [24]. Given a linear grid world, the goal is to reach cell B, starting from cell A, moving left or right within the cells. In classical planning, this problem is solved for a specific instance, that is, for a specific number of cells. As we increase the number of cells in this problem, the complexity of solving it grows exponentially, although the solutions address common subproblems and are remarkably similar to each other. Approaches for finding generalized plans attempt to extract, and subsequently utilize such common solutions and problem structures. A simple generalized solution is the one showed in Figure 6.1b, which says ”repeatedly move right, until *atB* is not observed.



(a) (b)

Figure 6.1: Grid world example and its generalized plan.

To formally define generalized planning, we need to start from two main components of planning, namely, the *agent* that executes the plan, and the *environment* in which the agent's plan is executed. In the usual definition of planning problem these two parts are merged. But to highlight the notion of generalized planning as synthesizing a plan for multiple environments, they are kept separated.

Plans are always executed by some agent, which has limitations on what it can observe and what actions it can perform.

Definition 4. *An* agent A *is a tuple* $A = \langle \mathsf{Ac}, \mathsf{Obs} \rangle$*, where*

- Ac *is a set of actions the agent can perform, and*
- Obs *is a set of observations the agent can make.*

To this, we may add constraints on the behavior of the agent, *e.g.*, on the sequences of actions that it can perform, in order to further specify the agent's "capability". Such constraints could be in the form of temporal logic formulae over Ac and Obs which we define later.

Definition 5. *Given an agent* $A = \langle \mathsf{Ac}, \mathsf{Obs} \rangle$*, a plan is a partial function* $p : \mathsf{Obs}^+ \to \mathsf{Ac} \cup \{\mathtt{stop}\}$*, where* `stop` *stands for plan termination. Such a partial function is required to be* prefix closed*: if* $p(o_1, \ldots o_n)$ *is defined, then so is* $p(o_1, \ldots o_i)$ *for all* $i < n$*.*

This is a very general notion of plan that completely abstracts from syntactic or structural characterization of a plan representation. Such a notion of plan is common in automated process synthesis [129] as well as in POMDP-based planning [115]. Notice that in order to define a plan, we only need the specification of the agent. In particular, we do not need any knowledge about the environment the agent acts in, so it is possible that the plan of an agent can be executed in multiple environments, which we define next.

Definition 6. *An environment, in which an agent's plan is executed, is a tuple* $E = \langle Events, S, s_0, \delta \rangle$*, where*

- *Events is the set of all events in the environment;*
- S *is the internal state space of the environment;*
- $s_0 \in S$ *is its initial (internal) state;*

– *$\delta : S \times Events \rightarrow S$ is the (partial) transition function.*

As in classical planning we assume the environment is deterministic, *i.e.*, there is only one initial state, and every event, if it happens in one state, may only take the environment to at most one single next state.

Definition 7. *A trace on E is a sequence $t = s_0e_1s_1e_2 \cdots e_ns_n$, where s_0 is the initial state of the environment E, and $s_i = \delta(s_{i-1}, e_i)$ (and hence $\delta(s_{i-1}, e_i)$ is defined) for all $i = 1, \ldots, n$. We denote by $Last(t)$ the last state s_n of t.*

A *goal* for the environment E is a specification of desired traces on E. Note that (possibly by allowing for infinite traces) this definition is general enough to capture several types of goals, including temporally-extended and long-running (infinite) ones [11,53,132].

Executing an agent's plan in an environment. In order to characterize the execution of an agent's plan in an environment, we need to know how the agent observations are related to the environment states, and the agent actions to the events happening in the environment. In particular, we need:

- An *observation function* $obs : S \rightarrow \mathsf{Obs}$, which determines how much of the environment the agent can observe for the purpose of plan execution, *i.e.*, when selecting an action to perform, the states s_1 and s_2 cannot be distinguished by the agent if $obs(s_1) = obs(s_2)$;

- An *execution function* $exec : \mathsf{Ac} \rightarrow Events$, which determines the events in the environment that the agent causes by doing its actions. This function enables separation between what the agent can do and what changes the environment may have.

Given the observation and execution functions, we can determine the execution of an agent A's plan p in the environment E. A *run* r of plan p in environment E is the trace $trace(p) = s_0e_1s_1e_2 \cdots e_ns_n$ such that for all $i = 1, \ldots, n$, $e_i = exec\big(p(obs(s_0), \ldots obs(s_{i-1}))\big)$. We call r *complete* if $p(obs(s_0), \ldots obs(s_n)) =$

$\texttt{stop}$ and for all $i < n$, $p(obs(s_0), \ldots obs(s_i)) \neq \texttt{stop}$. Notice that a plan, being deterministic, has at most one complete run in any given environment.

Definition 8 (**Basic planning problem**). *A* (basic) planning problem *P consists of an agent $A = \langle \mathsf{Ac}, \mathsf{Obs} \rangle$, an environment $E = \langle Events, S, s_0, \delta \rangle$ with a goal G for E, and related obs and exec functions. Formally a* basic planning problem *is a tuple $P = \langle \mathsf{Ac}, \mathsf{Obs}, Events, S, s_0, \delta, G, obs, exec \rangle$ where all the components are as above. A* solution *to a basic planning problem P is a plan p that generates a complete run r that fulfill the goal,* i.e.*, such that $Last(r) \in G$.*

Definition 9 (**Generalized planning problem**). *A* generalized planning problem $\mathbf{P} = \{P_1, P_2, \ldots\}$ *is a (finite or infinite) set of basic planning problems P_i, where all of the $P_i = \langle \mathsf{Ac}, \mathsf{Obs}, Events_i, S_i, s_{i0}, \delta_i, G_i, obs_i, exec_i \rangle$ share the same agent,* i.e.*, Ac and Obs are kept fixed. A solution for a generalized planning problem $\mathbf{P}$ is a plan p, such that p is a solution for every $P_i \in \mathbf{P}$. Intuitively, we require that the plan p for a fixed agent $A = \langle \mathsf{Ac}, \mathsf{Obs} \rangle$ achieves, on all of the environments $E_i = \langle Events_i, S_i, s_{i0}, \delta_i \rangle$, their respective goals G_i. In other words, p is a solution for the generalized planning problem iff it generates, for each corresponding environment E_i, a complete run r_i such that $Last(r_i) \in G_i$.*

6.1.1 One-Dimensional Planning Problems

One-Dimensional planning problems (1DPP) are a specific class of generalized planning problems. Intuitively, 1DDP model cases where an unknown and unbounded number of entities exist, which require independent treatment to achieve the goal. Roughly, a state in a 1DPP consists of a vector $\langle b \rangle \in 2^m$ (for fixed m) which represents properties of the environment, and an (unbounded) integer $n \in \mathbb{N}$ that represents the number of remaining tasks. An example of 1DPP includes tree-chopping [82] that we formally describe in later sections.

Definition 10. *A generalized problem $\mathbf{P} = \{P_1, P_2, \cdots\}$ is one-dimensional (*1DPP *for short) if all $P_i = \langle \mathsf{Ac}, \mathsf{Obs}, Events, S_i, s_{i0}, \delta_i, G, obs_i, exex \rangle$ share the same set* *of* Ac *(which includes the action stop),* Obs*, Events, G and exec, and there exists* $m \in \mathbb{N}$ *and* $\langle b_0 \rangle$ *such that, for all* $i \in \mathbb{N}$*:*

1. $S_i = \{\langle n, \langle b \rangle\rangle \mid n \in [i], \langle b \rangle \in 2^m\}$;

2. $s_{i0} = \langle i, \langle b_0 \rangle\rangle$;

3. *for every* $\langle b \rangle \in 2^m$ *and* $a \in A$ *there exists* $\langle b' \rangle$ *and* $d \in \{0, 1\}$ *such that for all* i *and* $n \in \mathbb{N}$*:* $\delta_i(\langle n, \langle b \rangle\rangle, a) = \langle n - d, \langle b' \rangle\rangle$. *Note that this condition does not say anything about the case that* $n = 0$. *Shorthand: we write* $\langle b \rangle \overset{\mathsf{a/d}}{\rightsquigarrow} \langle b' \rangle$.

4. *for every* $\langle b \rangle \in 2^m$ *and* $a \in A$ *there exists* $\langle b' \rangle$ *such that for all* i*:* $\delta_i(\langle 0, \langle b \rangle\rangle, a) = \langle 0, \langle b' \rangle\rangle$. *Shorthand: we write* $\langle b \rangle \overset{\mathsf{a}}{\rightsquigarrow}_0 \langle b' \rangle$.

5. $G \subseteq \{0\} \times 2^m$;

6. (a) *if* $n_1, n_2 \in \mathbb{N}$ *and* $i, j \in \mathbb{N}$ *then* $obs_i(\langle n_1, \langle b \rangle\rangle) = obs_j(\langle n_2, \langle b \rangle\rangle)$;

   (b) *if* $i, j \in \mathbb{N}$ *then* $obs_i(\langle 0, \langle b \rangle\rangle) = obs_j(\langle 0, \langle b \rangle\rangle)$;

   Shorthand: we write $obs'(\langle b \rangle)$ *for* $obs_i(\langle n, \langle b \rangle\rangle)$ *where* $i, n \in \mathbb{N}$ *(note that this is well defined, i.e., it does not depend on the values of* i, n*), and* $obs''(\langle b \rangle)$ *for* $obs_i(\langle 0, \langle b \rangle\rangle)$ *where* $i \in \mathbb{N}$ *(note that this is also well-defined, i.e., it does not depend on the value of* i*).*

The next lemma follows immediately from the definition of plan:

Lemma 2. *Let* s *and* t *be two observationally equivalent states. Starting from* s *and* t *the plan* p *gives the same set of actions as long as the resulting states are also observationally equivalent.*

6.2 Generalized-Planning Games

In this section we define generalized-planning (GP) games, known as games of imperfect information in the Formal Methods literature [136], that capture many generalized forms of planning.

Informally, two players (agent and environment) play on a transition-system. Play proceeds in rounds. In each round, from the current state s of the transition-system, the agent observes $obs(s)$ (some information about the current state), and picks an action a from the set of actions Ac, and then the environment picks an element of $\mathsf{tr}(s, a)$ (tr is the transition function of the transition-system) to become the new current state. Note that the players are asymmetric, i.e., the agent picks actions and the environment resolves non-determinism.

Linear-temporal logic. Formulas of linear-time propositional temporal logic (LTL) are built from a set atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective X (next), and the binary temporal connective U (until) [71, 128]. We define LTL over a finite set of letters Σ.[1] The *formulas of* LTL *(over* Σ*)* are generated by the following grammar:

$$\varphi ::= x \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

where $x \in \Sigma$.

We introduce the usual abbreviations for, e.g., $\vee, \mathsf{F}, \mathsf{G}$. Formulas of LTL (over Σ) are interpreted over infinite words $\alpha \in \Sigma^\omega$. Define the satisfaction relation $\models$ as follows:

1. $(\alpha, n) \models x$ iff $\alpha_n = x$;
2. $(\alpha, n) \models \varphi_1 \wedge \varphi_2$ iff $(\alpha, n) \models \varphi_i$ for $i = 1, 2$; i
3. $(\alpha, n) \models \neg\varphi$ iff it is not the case that $(\alpha, n) \models \varphi$;
4. $(\alpha, n) \models \mathsf{X}\,\varphi$ iff $(\alpha, n+1) \models \varphi$;
5. $(\alpha, n) \models \varphi_1\,\mathsf{U}\,\varphi_2$ iff there exists $i \geq n$ such that $(\alpha, i) \models \varphi_2$ and for all $j \in [n, i)$, $(\alpha, j) \models \varphi_1$.

[1] This is without loss of generality, since if LTL were defined over a set of atomic propositions AP we let $\Sigma = 2^{\mathrm{AP}}$ and replace atoms $p \in \mathrm{AP}$ by $\bigvee_{p \in x} x$ to get equivalent LTL formulas over Σ.

Thus, the formula $\mathbf{true}\,\mathsf{U}\,\varphi$, abbreviated as F, says that holds *eventually*, and the formula $\neg\,\mathsf{F}\,\neg\varphi$, abbreviated G, says that holds *henceforth.* For example, the formula $\mathsf{G}(\neg\text{request} \vee (\text{request}\,\mathsf{U}\,\text{grant})$ says whenever a request is made it holds continuously until it is eventually granted. We will write $\alpha \models \varphi$ if $(\alpha, 0) \models \varphi$ and say that α *satisfies* the LTL formula φ.

We shall see that the set of computations satisfying a given formula are exactly those accepted by some finite automaton on infinite words [145].

We briefly recall the basics notions of two player games of imperfect information.

Arena. An *arena of imperfect information*, or simply an *arena*, is a tuple $\mathbf{A} = (\mathsf{S}, I, \mathsf{Ac}, \mathsf{tr}, \mathsf{Obs}, obs)$, where

- S is a (possibly infinite) set of *states*,
- $I \subseteq \mathsf{S}$ is the set of *initial states*,
- Ac is a finite set of *actions*,
- $\mathsf{tr} : \mathsf{S} \times \mathsf{Ac} \to 2^{\mathsf{S}} \setminus \{\emptyset\}$ is the *transition function*,
- Obs is a (possibly infinite) set of *observations*,
- $obs : \mathsf{S} \to \mathsf{Obs}$, the *observation function*, maps each state to an observation.

We extend tr to sets of states: for $\emptyset \neq Q \subseteq \mathsf{S}$, let $\mathsf{tr}(Q, a)$ denote the set $\cup_{q \in Q}\mathsf{tr}(q, a)$.

Sets of the form $obs^{-1}(x)$ for $x \in \mathsf{Obs}$ are called *observation sets.* The set of all observation sets is denoted ObsSet. Non-empty subsets of observation sets are called *belief-states.* Informally, a belief-state is a subset of the states of the game that the play could be in after a given finite sequence of observations and actions.

Finite and finitely-branching. An arena is *finite* if S is finite, and *infinite* otherwise. An arena is *finitely-branching* if i) I is finite, and ii) for every s, a the cardinality of $\mathsf{tr}(s, a)$ is finite. Clearly, being finite implies being finitely-branching.

Strategies. A *play* in $\mathbf{A}$ is an infinite sequence $\pi = s_0 a_0 s_1 a_1 s_2 a_2 \ldots$ such that $s_0 \in I$ and for all $i \in \mathbb{N}_0$, $s_{i+1} \in \mathsf{tr}(s_i, a_i)$. A *history* $h = s_0 a_0 \ldots s_{n-1} a_{n-1} s_n$ is a finite prefix of a play ending in a state. The set of plays is denoted $\mathsf{Ply}(\mathsf{A})$, and the set of histories is denoted $\mathsf{Hist}(\mathbf{A})$ (we drop $\mathbf{A}$ when it is clear from the context). For a history or play $\pi = s_0a_0s_1a_1\dots$ write $obs(\pi)$ for the sequence $obs(s_0)a_0obs(s_1)a_1\dots$. A *strategy* (for the agent) is a function $\sigma : \mathsf{Hist}(\mathbf{A}) \to \mathsf{Ac}$. A strategy is *observational* if $obs(h) = obs(h')$ implies $\sigma(h) = \sigma(h')$. In Section 6.3 we will briefly mention an alternative (but essentially equivalent) definition of observational strategy, i.e., as a function $\mathsf{Obs}^+ \to \mathsf{Ac}$. We do not define strategies for the environment. A play $\pi = s_0a_0s_1a_1\dots$ is *consistent* with a strategy σ if for all $i \in \mathbb{N}$ we have that if $h \in \mathsf{Hist}(\mathbf{A})$ is a prefix of π, say $h = s_0a_0\dots s_{n-1}a_{n-1}s_n$, then $\sigma(h) = a_{n+1}$.

Generalized Planning Games. A *generalized-planning (GP) game*, is a tuple $\mathbf{G} = \langle \mathbf{A}, W\rangle$ where the *winning objective* $W \subseteq \mathsf{Obs}^\omega$ is a set of infinite sequences of observation sets. A *GP game with restriction* is a tuple $\mathbf{G} = \langle \mathbf{A}, W, F\rangle$ where, in addition, $F \subseteq \mathsf{S}^\omega$ is the *restriction.* Note that unlike the winning objective, the restriction need not be closed under observations. A GP game is *finite* (resp. *finitely branching*) if the arena $\mathbf{A}$ is finite (resp. finitely branching).

Winning. A strategy σ is *winning* in $\mathbf{G} = \langle \mathbf{A}, W\rangle$ if for every play $\pi \in \mathsf{Ply}(\mathsf{A})$ consistent with σ, we have that $obs(\pi) \in W$. Similarly, a strategy is winning in $\mathbf{G} = \langle \mathbf{A}, W, F\rangle$ if for every play $\pi \in \mathsf{Ply}(\mathsf{A})$ consistent with σ, if $\pi \in F$ then $obs(\pi) \in W$. Note that a strategy is winning in $\langle \mathbf{A}, W, \mathsf{Ply}(\mathbf{A})\rangle$ if and only if it is winning in $\langle \mathbf{A}, W\rangle$.

Solving a GP game. A central decision problem is the following, called *solving a GP game*: given a (finite representation of a) GP game of imperfect information $\mathbf{G}$, decide if the agent has a winning observational-strategy.

GP games of perfect information. An arena/GP game has *perfect information* if $\mathsf{Obs} = \mathsf{S}$ and $obs(s) = s$ for all s. We thus suppress mentioning Obs and obs completely, e.g., we write $\mathbf{A} = (\mathsf{S}, I, \mathsf{Ac}, \mathsf{tr})$ and $W, F \subseteq \mathsf{S}^\omega$. Note that in a GP game of perfect information every strategy is observational.

Example 1 (continued). We formalise the tree-chopping planning problem. Define the GP game $\mathbf{G}_{\mathsf{chop}} = \langle \mathbf{A}_{\mathsf{chop}}, W\rangle$ where $\mathbf{A}_{\mathsf{chop}} = \langle \mathsf{S}, I, \mathsf{Ac}, \mathsf{tr}, \mathsf{Obs}, obs\rangle$, and:

- $\mathsf{S} = \{down, success, failure\} \cup (\{uk\} \times \mathbb{N}_0) \cup (\{up\} \times \mathbb{N})$,
- $\mathsf{Ac} = \{\mathsf{chop}, \mathsf{look}, \mathsf{store}\}$, $I = \{uk\} \times \mathbb{N}$,
- tr is illustrated in Figure 6.2,
- $\mathsf{Obs} = \{\mathsf{DN}, \checkmark, \times, \mathsf{UK}, \mathsf{UP}\}$,
- obs maps $down \mapsto \mathsf{DN}$, $(up, i) \mapsto \mathsf{UP}$ for $i \in \mathbb{N}$, $(uk, i) \mapsto \mathsf{UK}$ for $i \in \mathbb{N}_0$, $failure \mapsto \times$, and $success \mapsto \checkmark$, and
- the objective W is defined as $\alpha \in W$ iff $\alpha \models \mathsf{F}\checkmark$.

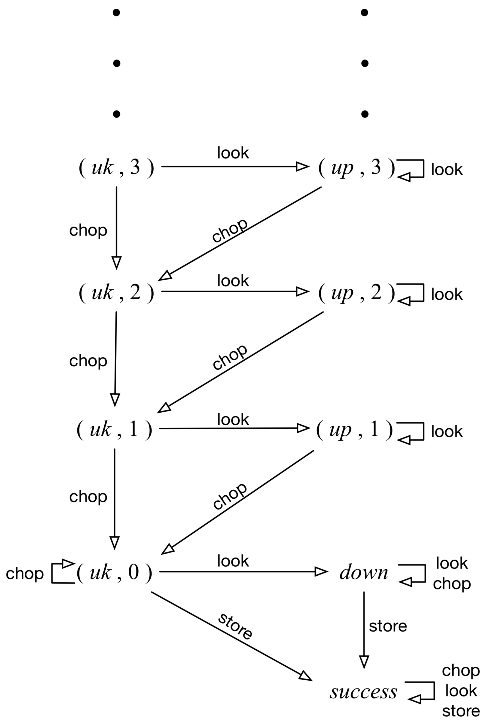


Figure 6.2: Part of the arena $\mathbf{A}_{\mathsf{chop}}$ (missing edges go to the $failure$ state). The numbers correspond to the numbers of chops required to fell the tree. The arena is not finitely-branching since it has infinitely many initial states $\{uk\} \times \mathbb{N}$.

The mentioned plan is formalized as the observational-strategy σ_{chop} that maps any history ending in (uk, i) to look (for $i \in \mathbb{N}_0$), (up, i) to chop (for $i \in \mathbb{N}$), *down* to store, and all others arbitrarily (say to store).

Note: σ_{chop} is a winning strategy, i.e., no matter which initial state the environment chooses, the strategy ensures that the play (it is unique because the rest of the GP game is deterministic) reaches the state *success* having observation $\checkmark$.

6.3 Generalized Form of Planning

In this section we establish that Generalized-planning (GP) games can model many different types of planning from the AI literature, including a variety of generalized forms of planning:

1. planning on finite transition-systems, deterministic actions, actions with conditional effects, partially observable states, incomplete information on the initial state, and temporally extended goals [53];

2. planning under partial observability with finitely many state variables, nondeterministic actions, reachability goals, and partial observability [138];

3. planning on finite transition systems, nondeterministic actions, looking for strong plans (i.e., adversarial nondeterminism) [17];

4. generalized planning, consisting of multiple (possibly infinitely many) related finite planning instances [81, 82].

We discuss the latter in detail. Following [81], a *generalized planning problem* $\mathfrak{P}$ is defined as a sequence of related classical planning problems. In our terminology, fix finite sets $\mathsf{Ac}, \mathsf{Obs}$ and let $\mathfrak{P}$ be a countable sequence $\mathbf{G}_1, \mathbf{G}_2, \dots$ where each $\mathbf{G}_n$ is a finite GP game of the form $\langle \mathsf{S}_n, \{\iota_n\}, \mathsf{Ac}, \mathsf{tr}_n, \mathsf{Obs}, obs_n, W_n \rangle$. In [81], a plan is an observational-strategy $p : \mathsf{Obs}^+ \to \mathsf{Ac}$, and a solution is a single plan that solves all of the GP games in the sequence. Now, we view $\mathfrak{P}$ as a single infinite

GP game as follows. Let $\mathbf{G}_{\mathfrak{P}}$ denote the *disjoint* union of the GP games in $\mathfrak{P}$. Formally, $\mathbf{G}_{\mathfrak{P}} = \langle \mathsf{S}, I, \mathsf{Ac}, \mathsf{tr}, \mathsf{Obs}, obs, W \rangle$ where

- $\mathsf{S} = \{(s, n) : s \in \mathsf{S}_n, n \in \mathbb{N}\}$,
- $I = \{(\iota_n, n) : n \in \mathbb{N}\}$,
- $\mathsf{tr}((s, n), a) = \{(t, n) : t \in \mathsf{tr}_n(s, a)\}$,
- $obs(s, n) = obs_n(s)$,
- $W = \cup_n W_n$.

Then: there is a correspondence between solutions for $\mathfrak{P}$ and winning observational-strategies in $\mathbf{G}_{\mathfrak{P}}$.

For example, consider the tree-chopping problem as formalized in [81, 82]: there are infinitely many planning instances which are identical except for an integer parameter denoting the number of chops required to fell the tree. The objective for all instances is to fell the tree. Using the translation above we get a GP game with an infinite arena which resembles (and, in fact, can be transformed to) the GP game in Example 1.

6.4 Generalized Belief-State Construction

In this section we show how to remove imperfect information from generalized-planning (GP) games $\mathbf{G}$. That is, we give a transformation of GP games of imperfect information $\mathbf{G}$ to GP games of perfect information $\mathbf{G}^{\beta}$ such that the agent has a winning observational-strategy in $\mathbf{G}$ if and only if the agent has a winning strategy in $\mathbf{G}^{\beta}$. The translation is based on the classic belief-state construction [136, 137]. Thus, we begin with a recap of that construction.

Belief-state Arena.[2] Let $\mathbf{A} = (\mathsf{S}, I, \mathsf{Ac}, \mathsf{tr}, \mathsf{Obs}, obs)$ be an arena (not necessarily finite). Recall from Section 6.2 that *observation sets* are of the form $obs^{-1}(x)$

[2] In the AI literature, this is sometimes called the *belief-space*.

for $x \in \mathsf{Obs}$, and are collectively denoted ObsSet. Define the arena of perfect information $\mathbf{A}^{\beta} = (\mathsf{S}^{\beta}, I^{\beta}, \mathsf{Ac}, \mathsf{tr}^{\beta})$ where,

- S^{β} is the set of belief-states, i.e., the non-empty subsets of the observation-sets,
- I^{β} consists of all belief-states of the form $I \cap X$ for $X \in \mathsf{ObsSet}$,
- $\mathsf{tr}^{\beta}(Q, a)$ consists of all belief-states of the form $\mathsf{tr}(Q, a) \cap X$ for $X \in \mathsf{ObsSet}$.

The idea is that $Q \in \mathsf{S}^{\beta}$ represents a refinement of the observation set: the agent, knowing the structure of $\mathbf{G}$ ahead of time, and the sequence of observations so far in the game, may deduce that it is in fact in a state from Q which may be a strict subset of its corresponding observation set X.[3]

NB. Since $\mathbf{A}^{\beta}$ is an arena, we can talk of its histories and plays. Although we defined S^{β} to be the set of all belief-states, only those belief-states that are reachable from I^{β} are relevant. Thus, overload notation and write S^{β} for the set of reachable belief-states, and $\mathbf{A}^{\beta}$ for the corresponding arena. This notation has practical relevance since if $\mathbf{A}$ is countable there are uncountably many belief-states; but in many cases only countably many (or, as in the running example, finitely many) reachable belief-states.

The intuition for the rest of this section is illustrated in the next example.

Example 2 (continued). Figure 6.3 shows the arena $\mathbf{A}^{\beta}_{\mathsf{chop}}$ corresponding to the arena from tree-chopping game $\mathbf{G}_{\mathsf{chop}}$, i.e.,

- S^{β} are the following belief-states: $\{(uk, n) \mid n \in \mathbb{N}_0\}$, denoted UK; $\{(up, n) \mid n \in \mathbb{N}\}$, denoted UP; $\{down\}$, denoted DN; $\{success\}$, denoted ✓; and $\{failure\}$.
- I^{β} is the belief-state UK,
- and tr^{β} is shown in the figure.

[3] To illustrate simply, suppose there is a unique initial state s, and that it is observationally equivalent to other states. At the beginning of the game the agent can deduce that it must be in s. Thus, its initial belief-state is $\{s\}$ and not its observation-set $obs^{-1}(obs(s))$. This belief can (and, in general, must) be exploited if the agent is to win.

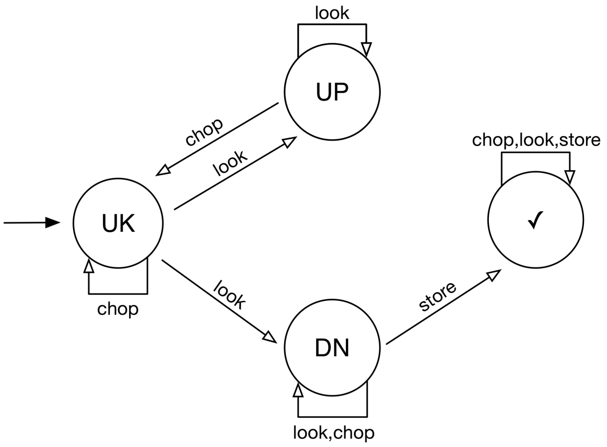


Figure 6.3: Part of the arena $\mathbf{A}^{\beta}_{\mathsf{chop}}$ (missing edges go the the $failure$ state). Each circle is a belief-state. The winning objective is $\mathsf{F}\,\checkmark$, and the restriction is $\neg\,\mathsf{G}\,\mathsf{F}[\mathsf{UK} \wedge \mathsf{X}\,\mathsf{look} \wedge \mathsf{X}\,\mathsf{X}\,\mathsf{UP}]$.

Note that the agent does not have a winning strategy in the GP game with arena $\mathbf{A}^{\beta}_{\mathsf{chop}}$ and winning condition $\mathsf{F}\,\checkmark$. The informal reason is that the strategy σ_{chop} (which codifies "alternately look and chop until the tree is sensed to be down, and then store the axe"), which wins in $\mathbf{G}$, does not work. The reason is that after every look the opponent can transition to UP (and never DN), resulting in the play $\rho = (\mathsf{UK}\ \mathsf{look}\ \mathsf{UP}\ \mathsf{chop})^{\omega}$, i.e., the repetition of ($\mathsf{UK}\ \mathsf{look}\ \mathsf{UP}\ \mathsf{chop}$) forever. Such a play of $\mathbf{A}^{\beta}_{\mathsf{chop}}$ does not correspond to any play in $\mathbf{A}_{\mathsf{chop}}$. This is a well known phenomena of the standard belief-set construction [142], which our construction overcomes by adding a restriction that removes from consideration plays such as ρ (as discussed in Example 4).

The following definition is central. It maps a history $h \in \mathsf{Hist}(\mathsf{A})$ to the corresponding history $h^{\beta} \in \mathsf{Hist}(\mathbf{A}^{\beta})$ of belief-states.

Definition 11. *For $h \in \mathsf{Hist}(\mathbf{A})$ define $h^{\beta} \in \mathsf{Hist}(\mathbf{A}^{\beta})$ inductively as follows.*

- *For $s \in I$, define $s^{\beta} \in I^{\beta}$ to be $I \cap obs^{-1}(obs(s))$. In words, s^{β} is the set of initial states the GP game could be in given the observation $obs(s)$.*

- *If* $h \in \mathsf{Hist}(\mathbf{A}), a \in \mathsf{Ac}, s \in \mathsf{S}$*, then* $(has)^{\beta} := h^{\beta}aB$ *where* $B := \mathsf{tr}(Last(h^{\beta}), a) \cap obs^{-1}(obs(s))$*. In words,* B *is the set of possible states the GP game could be in given the observation sequence* $obs(has)$*.*

In the same way, for $\pi \in \mathsf{Ply}(\mathsf{A})$ *define* $\pi^{\beta} \in \mathsf{Ply}(\mathsf{A}^{\beta})$*. Extend the map pointwise to sets of plays* $P \subseteq \mathsf{Ply}(\mathsf{A})$*, i.e., define* $P^{\beta} := \{\pi^{\beta} \in \mathsf{Ply}(\mathsf{A}^{\beta}) \mid \pi \in P\}$*. Finally, we give notation to the special case that* $P = \mathsf{Ply}(\mathsf{A})$*: write* $\mathsf{Im}(\mathbf{A})$ *for the set* $\{\pi^{\beta} \mid \pi \in \mathsf{Ply}(\mathsf{A})\}$*, called the* image of $\mathbf{A}$.

By definition, $\mathsf{Im}(\mathbf{A}) \subseteq \mathsf{Ply}(\mathbf{A}^{\beta})$. However, the converse is not always true.

Example 3 (continued). There is a play of $\mathbf{A}^{\beta}_{\mathsf{chop}}$ that is not in $\mathsf{Im}(\mathbf{A}_{\mathsf{chop}})$, e.g., $\rho = (uk\ \mathsf{look}\ up\ \mathsf{chop})^{\omega}$. Indeed, suppose $\pi^{\beta} = \rho$ and consider the sequence of counter values of π. Every look action establishes that the current counter value in π is positive (this is the meaning of the tree being *up*), but every chop action reduces the current counter value by one. This contradicts that counter values are always non-negative.

Remark 6.4.1. If $\mathbf{A}$ is finitely-branching then $\mathsf{Im}(\mathbf{A}) = \mathsf{Ply}(\mathsf{A}^{\beta})$. To see this, let ρ be a play in $\mathbf{A}^{\beta}$, and consider the forest whose nodes are the histories h of $\mathbf{A}$ such that h^{β} is a prefix of ρ. Each tree in the forest is finitely branching (because $\mathbf{A}$ is), and at least one tree in this forest is infinite. Thus, by Kőnig's lemma, the tree has an infinite path π. But π is a play in $\mathbf{A}$ and $\pi^{\beta} = \rho$.

Definition 12. *For* $\rho \in \mathsf{Ply}(\mathbf{A}^{\beta})$*, say* $\rho = B_0a_0B_1a_1\dots$*, define* $obs(\rho)$ *to be the sequence* $obs(q_0)a_0obs(q_1)a_1\dots$ *where* $q_i \in B_i$ *for* $i \in \mathbb{N}_0$ *(this is well defined since, by definition of the state set* S^{β}*, each* B_i *is a subset of a unique observation-set).*

The classic belief-state construction transforms $\langle \mathbf{A}, W\rangle$ into $\langle \mathbf{A}^{\beta}, W\rangle$. Example 2 shows that this transformation may not preserve the agent having a winning strategy if $\mathbf{A}$ is infinite. We now define the generalized belief-state construction and the main technical theorem of this work.

Definition 13. *Let $\mathbf{G} = \langle \mathbf{A}, W \rangle$ be a GP game. Define $\mathbf{G}^\beta = \langle \mathbf{A}^\beta, W, \mathsf{Im}(\mathbf{A}) \rangle$, a GP game of perfect information with restriction. The restriction $\mathsf{Im}(\mathbf{A}) \subseteq \mathsf{Ply}(\mathbf{A}^\beta)$ is the image of $\mathsf{Ply}(\mathbf{A})$ under the map $\pi \mapsto \pi^\beta$.*

Theorem 6.4.2. *Fix a (possibly infinite) arena $\mathbf{A}$ of imperfect information, and consider the belief-state arena $\mathbf{A}^\beta$ of perfect information and the set $\mathsf{Im}(\mathbf{A}) \subseteq \mathsf{Ply}(\mathbf{A}^\beta)$. For every winning objective W, the agent has a winning observational-strategy in the GP game $\mathbf{G} = \langle \mathbf{A}, W \rangle$ if and only if the agent has a winning strategy in the GP game $\mathbf{G}^\beta = \langle \mathbf{A}^\beta, W, \mathsf{Im}(\mathbf{A}) \rangle$. Moreover, if $\mathbf{A}$ is finitely-branching then $\mathbf{G}^\beta = \langle \mathbf{A}^\beta, W \rangle$.*[4]

Proof. The second statement follows from the first statement and Remark 6.4.1. For the first statement, we first need some facts that immediately follow from Definition 11.

1. $(h_1)^\beta = (h_2)^\beta$ if and only if $obs(h_1) = obs(h_2)$.

2. For every $h \in \mathsf{Hist}(\mathbf{A}^\beta)$ that is also a prefix of π^β there is a history $h' \in \mathsf{Hist}(\mathbf{A})$ that is also a prefix of π such that $(h')^\beta = h$. Also, for every $h' \in \mathsf{Hist}(\mathbf{A})$ that is also a prefix of π there is a history $h \in \mathsf{Hist}(\mathbf{A}^\beta)$ that is also a prefix of π^β such that $(h')^\beta = h$.

Second, there is a natural correspondence between observational strategies of $\mathbf{A}$ and strategies of $\mathbf{A}^\beta$.

- If σ is a strategy in $\mathbf{A}^\beta$ then define the strategy $\omega(\sigma)$ of $\mathbf{A}$ as mapping $h \in \mathsf{Hist}(\mathbf{A})$ to $\sigma(h^\beta)$. Now, $\omega(\sigma)$ is observational by Fact 1. Also, if π is consistent with $\omega(\sigma)$ then π^β is consistent with σ. Indeed, let h be a history that is also a prefix of π^β. We need to show that $h\sigma(h)$ is a prefix of π^β. Suppose that $\sigma(h) = a$. Take prefix h' of π such that $(h')^\beta = h$ (Fact 2). Then $\omega(\sigma)(h') = \sigma((h')^\beta) = \sigma(h) = a$. Since π is assumed consistent with $\omega(\sigma)$, conclude that $h'a$ is a prefix of π. Thus ha is a prefix of π^β.

[4] The case that $\mathbf{A}$ is finite appears in [136].

- If σ is an observational strategy in $\mathbf{A}$ then define the strategy $\kappa(\sigma)$ of $\mathbf{A}^{\beta}$ as mapping $h \in \mathsf{Hist}(\mathbf{A}^{\beta})$ to $\sigma(h')$ where h' is any history such that $h'^{\beta} = h$. This is well-defined by (†) and the fact that σ is observational. Also, if ρ is consistent with $\kappa(\sigma)$, then every π with $\pi^{\beta} = \rho$ (if there are any) is consistent with σ. Indeed, let h' be a history of π and take a prefix h of π^{β} such that $(h')^{\beta} = h$ (Fact 2). Then $\kappa(\sigma)(h) = \sigma(h')$, call this action a. But π^{β} is assumed consistent with $\kappa(\sigma)$, and thus ha is a prefix of π^{β}. Thus $h'a$ is a prefix of π.

We now put everything together and show that the agent has a winning observational-strategy in $\mathbf{G}$ iff the agent has a winning strategy in $\mathbf{G}^{\beta}$.

Suppose σ is a winning strategy in $\mathbf{G}^{\beta}$. Let $\pi \in \mathsf{Ply}(\mathbf{A})$ be consistent with the observational strategy $\omega(\sigma)$ of $\mathbf{G}$. Then $\pi^{\beta} \in \mathsf{Im}(\mathbf{A})$ is consistent with σ. But σ is assumed to be winning, thus $obs(\pi^{\beta}) \in W$. But $obs(\pi) = obs(\pi^{\beta})$. Conclude that $\omega(\sigma)$ is a winning strategy in $\mathbf{G}$.

Conversely, suppose σ is a winning strategy in $\mathbf{G}$. Let $\rho \in \mathsf{Im}(\mathbf{A})$ be consistent with the strategy $\kappa(\sigma)$ of $\mathbf{G}^{\beta}$, and take $\pi \in \mathsf{Ply}(\mathsf{A})$ be such that $\pi^{\beta} = \rho$ (such a π exists since we assumed $\rho \in \mathsf{Im}(\mathbf{A})$). Then π is consistent with σ. But σ is assumed to be winning, thus $obs(\pi) \in W$. But $obs(\rho) = obs(\pi^{\beta}) = obs(\pi)$. Conclude that $\kappa(\sigma)$ is a winning strategy in $\mathbf{G}^{\beta}$. □

Remark 6.4.3. The proof of Theorem 6.4.2 actually shows how to transform strategies between the GP games, i.e., $\sigma \mapsto \omega(\sigma)$ and $\sigma \mapsto \kappa(\sigma)$, and moreover, these transformations are inverses of each other.

We end with our running example:

Example 4 (Continued). Solving $\mathbf{G}_{\mathsf{chop}}$ (Figure 6.2) is equivalent to solving the finite GP game $\mathbf{G}^{\beta}_{\mathsf{chop}}$ of perfect information, i.e., $\langle \mathbf{A}^{\beta}_{\mathsf{chop}}, W, \mathsf{Im}(\mathbf{A}_{\mathsf{chop}}) \rangle$, where the arena $\mathbf{A}$ is shown in Figure 6.3. To solve this we should understand the structure of $\mathsf{Im}(\mathbf{A}_{\mathsf{chop}})$. It is not hard to see that a play $\rho \in \mathsf{Ply}(\mathbf{A}^{\beta}_{\mathsf{chop}})$ is in $\mathsf{Im}(\mathbf{A}_{\mathsf{chop}})$ if and only if it contains only finitely many infixes of the form "UK look UP". This property is expressible in LTL by the formula $\neg\, \mathsf{G}\, \mathsf{F}[\mathsf{UK} \wedge \mathsf{X}\, \mathsf{look} \wedge \mathsf{X}\, \mathsf{X}\, \mathsf{UP}]$. Thus we can apply the algorithm for solving finite games of perfect information with LTL objectives (see, e.g., [51, 130]) to solve $\mathbf{G}^{\beta}_{\mathsf{chop}}$, and thus the original GP game $\mathbf{G}_{\mathsf{chop}}$.

6.5 Application of the Construction

We now show how to use our generalized-planning (GP) games and our generalized belief-state constructions to obtain effective planning procedures for sophisticated problems. For the rest of this section we assume Obs is finite (**A** may be infinite) so that we can consider LTL temporally extended goals over the alphabet Obs. For instance, LTL formulas specify persistent surveillance missions such as "get items from region A, drop items at region B, infinitely often, and always avoid region C".

Definition 14. *Let φ be an* LTL *formula over* $\mathsf{Obs} \times \mathsf{Ac}$*. For an arena* $\mathbf{A}$*, define* $[[\varphi]] = \{\pi \in \mathsf{Ply}(\mathbf{A}) \mid obs(\pi) \models \varphi\}$*.*

The following is immediate from Theorem 6.4.2 and the fact that solving finite LTL games of perfect information is decidable [51, 130]:

Theorem 6.5.1. *Let* $\mathbf{G} = \langle \mathbf{A}, [[\varphi]] \rangle$ *be a GP game with a finite arena (possibly obtained as the disjoint union of several arenas sharing the same observations), and φ be an* LTL *winning objective. Then solving* $\mathbf{G}$ *can be reduced to solving the finite GP game* $\mathbf{G}^{\beta} = \langle \mathbf{A}^{\beta}, [[\varphi]] \rangle$ *of perfect information, which is decidable.*

Although we defined winning objectives to be observable, one may prefer general winning conditions, i.e., $W \subseteq S^{\omega}$. In this case, for finite arenas there is a translation from parity-objectives to observable parity-objectives [32]; moreover, for reachability objectives, a plan reaches a goal $T \subseteq \mathsf{S}$ iff it reaches a belief-state $B \subseteq T$ [17].

Next we look a case where the arena is actually infinite. Recently, the AI community has considered games generated by pushdown-automata [40, 124]. However, the games considered are of perfect information and cannot express generalized-planning problems or planning under partial observability. In contrast, our techniques can solve these planning problems on pushdown domains assuming that the stack is not hidden (we remark that if the stack is hidden, then game-solving becomes undecidable [9]):

Theorem 6.5.2. *Let* $\mathbf{G} = \langle \mathbf{A}, [[\varphi]] \rangle$ *be a GP game with a pushdown-arena with observable stack, and* φ *is an* LTL *formula. Then solving* $\mathbf{G}$ *can be reduced to solving* $\mathbf{G}^{\beta} = \langle \mathbf{A}^{\beta}, [[\varphi]] \rangle$*, a GP game with pushdown-arena with perfect information, which is decidable.*

Proof. Let P be a pushdown-automaton with states Q, initial state q_0, finite input alphabet Ac, and finite stack alphabet Γ. We call elements of Γ^* stacks, and denote the empty stack by ϵ. Also, fix an observation function on the states, i.e., $f : Q \to \mathsf{Obs}$ for some set Obs (we do not introduce notation for the transition function of P). A pushdown-arena $\mathbf{A}_P = (\mathsf{S}, I, \mathsf{Ac}, \mathsf{tr}, \mathsf{Obs}, \mathit{obs})$ is generated by P as follows: the set of states S is the set of configurations of P, i.e., pairs (q, γ) where $q \in Q$ and $\gamma \in \Gamma^*$ is a stack-content of P; the initial state of $\mathbf{A}$ is the initial configuration, i.e., $I = \{(q_0, \epsilon)\}$; the transition function of $\mathbf{A}$ is defined as $\mathsf{tr}((q, \gamma), a) = (q', \gamma')$ if P can move in one step from state q and stack content γ to state q' and stack content γ' by consuming the input letter a; the observation function *obs* maps a configuration (q, γ) to $f(q)$ (i.e., this formalizes the statement that the stack is observable). Observe now that: (1) the GP game $\mathbf{A}$ is finitely-branching; (2) the GP game $\mathbf{A}^{\beta}$ is generated by a pushdown automaton (its states are subsets of Q). Thus we can apply Theorem 6.4.2 to reduce solving $\mathbf{A}$, a GP-game with imperfect information and pushdown arena, to $\mathbf{A}^{\beta}$, a GP-game with perfect information and pushdown arena. The latter is decidable [163]. □

6.6 Related work in Formal Methods

Games of imperfect information on *finite* arenas have been studied extensively. Reachability winning-objectives were studied in [137] from a complexity point of view: certain games were shown to be universal in the sense that they are the hardest games of imperfect information, and optimal decision procedures were given. More generally, ω-regular winning-objectives were studied in [136], and symbolic algorithms were given (also for the case of randomized strategies).

To solve (imperfect-information) games on infinite arenas one needs a finite-representation of the infinite arena. One canonical way to generate infinite arenas is by parametric means. In this line, [84] study the synthesis problem for distributed architectures with a parametric number of finite-state components. They leverage results from the Formal Methods literature that say that for certain types of token-passing systems there is a cutoff [58], i.e., an upper bound on the number of components one needs to consider in order to synthesize a protocol for any number of components. Another way to generate infinite arenas is as configuration spaces of pushdown automata. These are important in analysis of software because they capture the flow of procedure calls and returns in reactive programs. Module-checking pushdown systems of imperfect information [4, 25] can be thought of as games in which the environment plays non-deterministic strategies. Although undecidable, by not hiding the stack (cf. Theorem 6.5.2) decidability of module-checking is regained.

Finally, we note that synthesis of distributed systems has been studied in the Formal Methods literature using the techniques of games, starting with [131]. Such problems can be cast as multi-player games of imperfect information, and logics such as ATL with knowledge can be used to reason about strategies in these games. However, even for three players, finite arenas, and reachability goals, the synthesis problem (and the corresponding model checking problem for ATLK) is undecidable [55].

6.7 Conclusions and Discussion

Although our technique for removing partial observability is sound and complete, it is, necessarily, not algorithmic: indeed, no algorithm can always remove partial observability from computable infinite domains and result in a solvable planning problem (e.g, one with a finite domain).[5]

The main avenue for future technical work is to establish natural classes of generalized-planning problems that can be solved algorithmically. We believe the methodology of this work will be central to this endeavor. Indeed, as we showed in Section 6.5, we can identify $\mathsf{Im}(\mathbf{A})$ in a number of cases. We conjecture that one can do the same for all of the one-dimensional planning problems of [81, 82].

The framework presented in this work is non-probabilistic, but extending it with probabilities and utilities associated to agent choices [10, 72, 92, 111] are of great interest. In particular, POMDPs with temporally-extended winning objectives (e.g., LTL, Büchi) have been studied for finite domains [34]. We leave for future work the problem of extending our setting to deal with such POMDPs over the infinite domains.

[5] In fact, there is no algorithm solving games of perfect observation on computable domains with reachability objectives.

List of Figures

List of Tables

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 1978.

[2] R. Alur and S. La Torre. Deterministic generators and games for ltl fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[3] B. Aminof, O. Kupferman, and A. Murano. Improved Model Checking of Hierarchical Systems. *Inf. Comput.*, 210:68–86, 2012.

[4] B. Aminof, A. Legay, A. Murano, O. Serre, and M. Y. Vardi. Pushdown module checking with imperfect information. *Inf. Comput.*, 223, 2013.

[5] B. Aminof, V. Malvone, A. Murano, and S. Rubin. Graded modalities in strategy logic. *Inf. Comput.*, 261(Part):634–649, 2018.

[6] B. Aminof, F. Mogavero, and A. Murano. Synthesis of Hierarchical Systems. *Science of Comp. Program.*, 83:56–79, 2013.

[7] R. Arcucci, U. Marotta, A. Murano, and L. Sorrentino. Parallel parity games: a multicore attractor for the zielonka recursive algorithm. In *ICCS 2017*, pages 525–534, 2017.

[8] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447 – 452, 1998.

[9] S. Azhar, G. Peterson, and J. Reif. Lower bounds for multiplayer non-cooperative games of incomplete information. *J. Comp. Math. Appl.*, 41:957–992, 2001.

[10] B. and H. Geffner. Solving POMDPs: Rtdp-bel vs. point-based algorithms. In *IJCAI 2009*, pages 1641–1646, 2009.

[11] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. Math. Artif. Intell.*, 22(1-2):5–27, 1998.

[12] R. Iris Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Form. Meth. in Sys. Des.*, pages 171–206, 1997.

[13] M. Bakera, S. Edelkamp, P. Kissmann, and C. D. Renner. Solving μ-calculus parity games by symbolic planning. In *MoChArt 2008*, pages 15–33, 2008.

[14] M. Ben-Ari. *Principles of the Spin Model Checker.* Springer, 2008.

[15] M. Benerecetti, D. Dell'Erba, and F. Mogavero. Solving parity games via priority promotion. In *CAV 2016*, pages 270–290, 2016.

[16] B. Bérard. *Systems and software verification: model checking techniques and tools.* Springer, 2001.

[17] P. Bertoli, A. Cimatti, Marco R., and P. Traverso. Strong planning under partial observability. *Artif.. Intell.*, 170(4):337 – 384, 2006.

[18] P. Bertoli and M. Pistore. Planning with extended goals and partial observability. In *ICAPS 2004*, pages 270–278, 2004.

[19] D. Berwanger. Admissibility in Infinite Games. In *STACS 2007*, pages 188–199, 2007.

[20] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[21] U. Boker and K. Lehtinen. On the way to alternating weak automata. In *FSTTCS 2018*, 2018.

[22] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[23] P.A. Bonatti, C. Lutz, A. Murano, and M.Y. Vardi. The Complexity of Enriched Mu-Calculi. 4(3):1–27, 2008.

[24] B. Bonet, H. Palacios, and H. Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS 2009*, pages 34–41, 2009.

[25] L. Bozzelli, A. Murano, and A. Peron. Pushdown module checking. *Form. Meth. in Sys. Des.*, 36(1):65–95, 2010.

[26] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, pages 677–691, 1986.

[27] J. R. Buchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[28] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *LICS 1990*, pages 428–439, 1990.

[29] D. Bustan, O. Kupferman, and M. Y. Vardi. A measured collapse of the modal μ-calculus alternation hierarchy. In *STACS 2004*, pages 522–533, 2004.

[30] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In *STOC 2017*, pages 252–263, 2017.

[31] P. Cermák, A. Lomuscio, and A. Murano. Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In *AAAI 2015*, pages 2038–2044, 2015.

[32] K. Chatterjee and L. Doyen. The complexity of partial-observation parity games. In *LPAR 2010*, pages 1–14. Springer, 2010.

[33] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Generalized Mean-payoff and Energy Games. In *FSTTCS 2010*, pages 505–516, 2010.

[34] K. Chatterjee, L. Doyen, and T.A. Henzinger. Qualitative analysis of partially-observable markov decision processes. In *MFCS 2010*, pages 258–269, 2010.

[35] K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer. Improved set-based symbolic algorithms for parity games. In *CSL 2017*, pages 18:1–18:21, 2017.

[36] K. Chatterjee and M. Henzinger. An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In *SODA 2012*, pages 1386–1399, 2012.

[37] K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Mean-payoff parity games. In *LICS 2005*, pages 178–187, 2005.

[38] K. Chatterjee, T. A. Henzinger, and N. Piterman. Generalized parity games. In *FOSSACS 2007*, pages 153–167, 2007.

[39] K. Chatterjee, M. Jurdzinski, and T. A. Henzinger. Quantitative stochastic parity games. In *SODA 2004*, pages 121–130, 2004.

[40] T. Chen, F. Song, and Z. Wu. Global model checking on pushdown multi-agent systems. In *AAAI 2016*, pages 2459–2465, 2016.

[41] Alonzo Church. Logic, arithmetic, and automata. *Journal of Symbolic Logic*, 29(4):210–210, 1964.

[42] K. Claessen, J. Fisher, S. Ishtiaq, N. Piterman, and Q. Wang. Model-checking signal transduction networks through decreasing reachability sets. In *CAV 2013*, pages 85–100, 2013.

[43] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*. Springer Berlin Heidelberg, 1996.

[44] E. M. Clarke, A. Fehnker, S. Kumar Jha, and H. Veith. Temporal logic model checking. In *Handbook of Networked and Embedded Control Systems*, pages 539–558. 2005.

[45] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[46] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *LP 1981*, pages 52–71, 1981.

[47] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 2002.

[48] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *FOCS 1988*, pages 338–345, 1988.

[49] L. Daviaud, M. Jurdzinski, and R. Lazic. A pseudo-quasi-polynomial algorithm for mean-payoff parity games. In *LICS 2018*, pages 325–334, 2018.

[50] L. de Alfaro and M. Faella. An Accelerated Algorithm for 3-Color Parity Games with an Application to Timed Games. In *CAV 2007*, pages 108–120, 2007.

[51] L. de Alfaro, T. A. Henzinger, and R. Majumdar. From verification to control: Dynamic programs for omega-regular objectives. In *LICS 2001*, pages 279–290, 2001.

[52] G. De Giacomo, A. Di Stasio A. Murano, and S. Rubin. Imperfect-information games and generalized planning. In *IJCAI 2016*, pages 1037–1043, 2016.

[53] G. De Giacomo and M. Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In *ECP 1999*, pages 226–238, 1999.

[54] A. Di Stasio, A. Murano, and M. Y. Vardi. Solving parity games: Explicit vs symbolic. In *CIAA 2018*, pages 159–172, 2018.

[55] C. Dima and F. L. Tiplea. Model-checking ATL under imperfect information and perfect recall semantics is undecidable. *CoR*, abs/1102.4225, 2011.

[56] R. Drechsler, B. Becker, N. Göckel, and A. Jahnke. A genetic algorithm for decomposition type choice in okfdds. *International Journal on Artificial Intelligence Tools*, 4(4):525, 1995.

[57] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[58] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. of POPL'95*, 1995.

[59] E.A. Emerson and J.Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time. *Journal of the ACM*, 33(1):151–178, 1986.

[60] E.A. Emerson and C. Jutla. Tree Automata, μ-Calculus and Determinacy. In *FOCS 1991*, pages 368–377, 1991.

[61] J. Esparza and J. Kretínský. From LTL to deterministic automata: A safraless compositional approach. In *CAV 2014*, pages 192–208, 2014.

[62] J. Esparza, J. Kretínský, J. Raskin, and S. Sickert. From LTL and limit-deterministic büchi automata to deterministic parity automata. In *TACAS 2017*, pages 426–442, 2017.

[63] J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak. An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In *SPIN 2017*, pages 112–121, 2017.

[64] P. Felli, G. De Giacomo, and A. Lomuscio. Synthesizing agent protocols from LTL specifications against multiple partially-observable environments. In *KR 2012*, pages 457–466, 2012.

[65] A. Ferrante, A. Murano, and M. Parente. Enriched μ-calculi module checking. *Logical Methods in Computer Science*, 4(3), 2008.

[66] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. *Artif. Intell.*, 3:251 – 288, 1972.

[67] E. Filiot, N. Jin, and J. Raskin. An antichain algorithm for LTL realizability. In *CAV 2009*, pages 263–277, 2009.

[68] E. Filiot, N. Jin, and J. Raskin. Compositional algorithms for LTL synthesis. In *ATVA 2010*, pages 112–127, 2010.

[69] O. Friedmann and M. Lange. The PGSolver collection of parity game solvers. *University of Munich*, 2009.

[70] O. Friedmann and M. Lange. Solving Parity Games in Practice. In *ATVA 2009*, pages 182–196, 2009.

[71] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL 1980*, pages 163–173, 1980.

[72] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning.* Morgan & Claypool, 2013.

[73] R. P. Goldman and M. S. Boddy. Expressive planning and explicit knowledge. In *AIPS 1996*, pages 110–117, 1996.

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

[75] A. Harding, M. Ryan, and P. Schobbens. A new algorithm for strategy synthesis in LTL games. In *TACAS 2005*, pages 477–492, 2005.

[76] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In *FME 1996*, pages 662–681, 1996.

[77] K. Heljanko, M. Keinänen, M. Lange, and I. Niemelä. Solving parity games by a reduction to sat. *J. Comput. Syst. Sci.*, 78(2):430–440, 2012.

[78] T. A. Henzinger and N. Piterman. Solving games without determinization. In *Computer Science Logic, 20th International Workshop, CSL 2006*, pages 395–410, 2006.

[79] T. A. Henzinger, . Jhala R, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN 2003*, pages 235–239, 2003.

[80] P. Hoffmann and M. Luttenberger. Solving parity games on the GPU. In *ATVA 2013*, pages 455–459, 2013.

[81] Y. Hu and G. De Giacomo. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI 2011*, pages 918–923, 2011.

[82] Y. Hu and H. J. Levesque. A correctness result for reasoning about one-dimensional planning problems. In *KR 2010*, pages 2638–2643.

[83] N Ishiura, H. Sawada, and S. Yajima. Minimazation of binary decision diagrams based on exchanges of variables. In *ICCAD 1993*, pages 472–475, 1991.

[84] S. Jacobs and R. Bloem. Parameterized synthesis. *LMCS*, 10(1), 2014.

[85] W. Jamroga and A. Murano. On module checking and strategies. In *AAMAS 2014*, pages 701–708, 2014.

[86] W. Jamroga and A. Murano. Module checking of strategic ability. In *AAMAS 2015*, pages 227–235, 2015.

[87] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV 2005*, pages 226–238, 2005.

[88] M. Jurdzinski. Deciding the Winner in Parity Games is in UP ∩ co-Up. *Inf. Process. Lett.*, 68(3):119–124, 1998.

[89] M. Jurdzinski. Small Progress Measures for Solving Parity Games. In *STACS 2000*, pages 290–301, 2000.

[90] M. Jurdzinski and R. Lazic. Succinct progress measures for solving parity games. In *LICS 2017*, pages 1–9, 2017.

[91] M. Jurdzinski, M. Paterson, and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.

[92] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[93] G. Kant and J. van de Pol. Generating and solving symbolic parity games. In *GRAPHITE 2014*, pages 2–14, 2014.

[94] J.J. A. Keiren. Benchmarks for parity games. In *FSEN 2015*, pages 127–142, 2015.

[95] D. Kini and M. Viswanathan. Optimal translation of LTL to limit deterministic automata. In *TACAS 2017*, pages 113–129, 2017.

[96] D. Kozen. Results on the Propositional μ-Calculus. *TCS*, 27(3):333–354, 1983.

[97] J. Kretínský and J. Esparza. Deterministic automata for the (f, g)-fragment of LTL. In *CAV 2012*, pages 7–22, 2012.

[98] J. Kretínský and R. Ledesma-Garza. Rabinizer 2: Small deterministic automata for LTL $\setminus$ GU. In *ATVA 2013*, pages 446–450, 2013.

[99] J. Kretínský, T. Meggendorfer, S. Sickert, and C. Ziegler. Rabinizer 4: From LTL to your favourite deterministic automaton. In *CAV 2018*, pages 567–577, 2018.

[100] J. Kretínský, T. Meggendorfer, C. Waldmann, and M. Weininger. Index appearance record for transforming rabin automata into parity automata. In *TACAS 2017*, pages 443–460, 2017.

[101] O. Kupferman, G. Morgenstern, and A. Murano. Typeness for omega-regular automata. *Int. J. Found. Comput. Sci.*, 17(4):869–884, 2006.

[102] O. Kupferman, N. Piterman, and M. Y. Vardi. Safraless compositional synthesis. In *CAV 2006*, pages 31–44, 2006.

[103] O. Kupferman, M. Vardi, and P. Wolper. Module Checking. *Inf. Comput.*, 164(2):322–344, 2001.

[104] O. Kupferman and M. Y. Vardi. Weak Alternating Automata and Tree Automata Emptiness. In *STOC 1998*, pages 224–233, 1998.

[105] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[106] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *FOCS 2005*, pages 531–542, 2005.

[107] O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, 2000.

[108] R. P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach.* Princeton University Press, 1994.

[109] S. La Torre, A. Murano, and M. Parente. Model-checking the secure release of a time-locked secret over a network. *Electr. Notes Theor. Comput. Sci.*, 99:229–243, 2004.

[110] L. Lamport. What good is temporal logic? *Information Processing 83, R. E. A. Mason, ed., Elsevier Publishers*, 83:657–668, 1983.

[111] S. M. LaValle. *Planning algorithms.* Cambridge, 2006.

[112] K. Lehtinen. A modal μ perspective on solving parity games in quasi-polynomial time. In *LICS 2018*, pages 639–648, 2018.

[113] H. Levesque. Planning with loops. In *IJCAI 2005*, pages 509–515, 2005.

[114] H. J. Levesque. What is planning in the presence of sensing. In *AAAI 1996*, pages 1139–1146, 1996.

[115] D. Nau M. Ghallab and P. Traverso. *Automated Planning: Theory & Practice.* Elsevier, 2008.

[116] V. Malvone, A. Murano, and L. Sorrentino. Concurrent multi-player parity games. In *AAMAS 2016*, pages 689–697, 2016.

[117] D.A. Martin. Borel determinacy. *Annals of Mathematics*, 102:363–371, 1975.

[118] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[119] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl. Logic*, pages 149–184, 1993.

[120] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. Reasoning about strategies: On the model-checking problem. *ACM Trans. Comput. Log.*, 15(4):34:1–34:47, 2014.

[121] F. Mogavero, A. Murano, and L. Sorrentino. On Promptness in Parity Games. In *LPAR 2013*, pages 601–618, 2013.

[122] D.E. Muller, A. Saoudi, and P.E. Schupp. Weak Alternating Automata Give a Simple Explanation of Why Most Temporal and Dynamic Logics are Decidable in Exponential Time. In *LICS 1988*, pages 422–427, 1988.

[123] A. Murano, M. Napoli, and M. Parente. Program complexity in hierarchical module checking. In *LPAR 2008*, pages 318–332, 2008.

[124] A. Murano and G. Perelli. Pushdown multi-agent system verification. In *IJCAI 2015*, pages 1090–1097, 2015.

[125] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *ICCAD 1995*, pages 74–77, 1995.

[126] S. Panda, F. Somenzi, and B. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *ICCAD 1994*, pages 628–631, 1994.

[127] N. Piterman. From nondeterministic büchi and streett automata to deterministic parity automata. *Logical Methods in Computer Science*, 3(3), 2007.

[128] A. Pnueli. The temporal logic of programs. In *FOCS 1977*, pages 46–57, 1977.

[129] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL 1989*, pages 179–190, 1989.

[130] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP 1989*, pages 652–671, 1989.

[131] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *STOC 1990*, pages 746–757, 1990.

[132] C. Pralet, G. Verfaillie, M. Lemaître, and G. Infantes. Constraint-based controller synthesis in non-deterministic and partially observable domains. In *ECAI 2010*, pages 681–686, 2010.

[133] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Programs in Cesar. In *SP 1982*, pages 337–351, 1982.

[134] M. O. Rabin. *Automata on Infinite Objects and Church's Problem.* American Mathematical Society, 1972.

[135] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *TAMS*, 141:1–35, 1969.

[136] J. Raskin, K. Chatterjee, L. Doyen, and T. A. Henzinger. Algorithms for omega-regular games with imperfect information. *LMCS*, 3(3), 2007.

[137] J. H. Reif. The complexity of two-player games of incomplete information. *JCSS*, 29(2), 1984.

[138] J. Rintanen. Complexity of planning with partial observability. In *ICAPS 2004*, pages 345–354, 2004.

[139] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[140] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD 1993*, pages 42–47, 1993.

[141] S. Safra. On the complexity of omega-automata. In *FOCS 1988*, pages 319–327, 1988.

[142] S. Sardiña, G. De Giacomo, Y. Lespérance, and H. J. Levesque. On the limits of planning over belief states under strict uncertainty. In *KR 2016*, pages 463–471, 2006.

[143] S. Schewe. Solving Parity Games in Big Steps. In *FSTTCS 2007*, pages 449–460, 2007.

[144] S. Schewe. An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games. In *CSL 2008*, pages 369–384, 2008.

[145] R. Sherman, A. Pnueli, and D. Harel. Is the interesting part of process logic uninteresting?: A translation from pl to pdl. In *SIAM J. on Computing*, pages 347–360, 1982.

[146] S. Sickert, J. Esparza, S. Jaax, and J. Kretínský. Limit-deterministic büchi automata for linear temporal logic. In *CAV 2016*, pages 312–332, 2016.

[147] S. Sohail and F. Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *STTT*, 15(5-6):433–454, 2013.

[148] S. Sohail, F. Somenzi, and K. Ravi. A hybrid algorithm for LTL games. In *VMCAI 2008*, pages 309–323, 2008.

[149] S. Srivastava. Foundations and applications of generalized planning. *AI Commun.*, 24(4):349–351, 2011.

[150] S. Srivastava, N. Immerman, and S. Zilberstein. Learning generalized plans using abstract counting. In *Proc. of AAAI 2008*, 2008.

[151] S. Srivastava, N. Immerman, and S. Zilberstein. A new representation and associated algorithms for generalized planning. *Artif. Intell.*, 175(2):615–647, 2011.

[152] S. Srivastava, S. Zilberstein, A. Gupta, P. Abbeel, and S. J. Russell. Tractability of planning with loops. In *AAAI 2015*, pages 3393–3401, 2015.

[153] A. Di Stasio, A. Murano, G. Perelli, and M. Y. Vardi. Solving parity games using an automata-based algorithm. In *CIAA 2016*, pages 64–76, 2016.

[154] A. Di Stasio, A. Murano, V. Prignano, and L. Sorrentino. Solving parity games in scala. In *FACS 2014*, pages 145–161, 2014.

[155] D. Tabakov. Evaluation of explicit and symbolic automata-theoretic algorithm. Master's thesis, Rice University, 2005.

[156] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. 1990.

[157] W. Thomas. Facets of Synthesis: Revisiting Church's Problem. In *FOSSACS 2009*, pages 1–14, 2009.

[158] T. van Dijk. Attracting tangles to solve parity games. In *CAV 2018*, pages 198–215, 2018.

[159] T. van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In *TACAS 2018*, pages 291–308, 2018.

[160] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS 1986*, pages 332–344, 1986.

[161] J. Vöge and M. Jurdzinski. A Discrete Strategy Improvement Algorithm for Solving Parity Games. In *CAV 2000*, pages 202–215, 2000.

[162] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *CIAA 2003*, pages 11–22, 2003.

[163] I. Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001.

[164] I. Wegener. Simulated annealing beats metropolis in combinatorial optimization. In *ICALP 2005*, pages 589–601, 2005.

[165] T. Wilke. Alternating Tree Automata, Parity Games, and Modal μ-Calculus. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 8(2):359, 2001.

[166] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi. A symbolic approach to safety ltl synthesis. In *HVC 2017*, pages 147–162, 2017.

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