Strategy Representation by Decision Trees in Reactive Synthesis

Tomáš Brázdil[1], Krishnendu Chatterjee[2], Jan Křetínský[3(✉)], and Viktor Toman[2]

[1] Masaryk University, Brno, Czech Republic
[2] Institute of Science and Technology Austria, Klosterneuburg, Austria
[3] Technical University of Munich, Munich, Germany
jan.kretinsky@tum.de

Abstract. Graph games played by two players over finite-state graphs are central in many problems in computer science. In particular, graph games with ω-regular winning conditions, specified as parity objectives, which can express properties such as safety, liveness, fairness, are the basic framework for verification and synthesis of reactive systems. The decisions for a player at various states of the graph game are represented as strategies. While the algorithmic problem for solving graph games with parity objectives has been widely studied, the most prominent data-structure for strategy representation in graph games has been binary decision diagrams (BDDs). However, due to the bit-level representation, BDDs do not retain the inherent flavor of the decisions of strategies, and are notoriously hard to minimize to obtain succinct representation. In this work we propose decision trees for strategy representation in graph games. Decision trees retain the flavor of decisions of strategies and allow entropy-based minimization to obtain succinct trees. However, decision trees work in settings (e.g., probabilistic models) where errors are allowed, and overfitting of data is typically avoided. In contrast, for strategies in graph games no error is allowed, and the decision tree must represent the entire strategy. We develop new techniques to extend decision trees to overcome the above obstacles, while retaining the entropy-based techniques to obtain succinct trees. We have implemented our techniques to extend the existing decision tree solvers. We present experimental results for problems in reactive synthesis to show that decision trees provide a much more efficient data-structure for strategy representation as compared to BDDs.

1 Introduction

Graph Games. We consider nonterminating two-player graph games played on finite-state graphs. The vertices of the graph are partitioned into states controlled by the two players, namely, player 1 and player 2, respectively. In each round the state changes according to a transition chosen by the player controlling the current state. Thus, the outcome of the game being played for an infinite

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10805, pp. 385–407, 2018.
https://doi.org/10.1007/978-3-319-89960-2_21

number of rounds, is an infinite path through the graph, which is called a play. An objective for a player specifies whether the resulting play is either winning or losing. We consider zero-sum games where the objectives of the players are complementary. A strategy for a player is a recipe to specify the choice of the transitions for states controlled by the player. Given an objective, a winning strategy for a player from a state ensures the objective irrespective of the strategy of the opponent.

Games and Synthesis. These games play a central role in several areas of computer science. One important application arises when the vertices and edges of a graph represent the states and transitions of a reactive system, and the two players represent controllable versus uncontrollable decisions during the execution of the system. The *synthesis* problem for reactive systems asks for the construction of a winning strategy in the corresponding graph game. This problem was first posed independently by Church [17] and Büchi [14], and has been extensively studied [15,28,37,45]. Other than applications in synthesis of discrete-event and reactive systems [43,46], game-theoretic formulations play a crucial role in modeling [1,21], refinement [30], verification [3,20], testing [5], compatibility checking [19], and many other applications. In all the above applications, the objectives are ω-regular, and the ω-regular sets of infinite paths provide an important and robust paradigm for reactive-system specifications [36,50].

Parity Games. Graph games with parity objectives are relevant in reactive synthesis, since all common specifications for reactive systems are expressed as ω-regular objectives that can be transformed to parity objectives. In particular, a convenient specification formalism in reactive synthesis is LTL (linear-time temporal logic). The LTL synthesis problem asks, given a specification over input and output variables in LTL, whether there is a strategy for the output sequences to ensure the specification irrespective of the behavior of the input sequences. The conversion of LTL to non-deterministic Büchi automata, and non-deterministic Büchi automata to deterministic parity automata, gives rise to a parity game to solve the LTL synthesis problem. Formally, the algorithmic problem asks for a given graph game with a parity objective and a starting state, whether player 1 has a winning strategy. This problem is central in verification and synthesis. While it is a major open problem whether the problem can be solved in polynomial time, it has been widely studied in the literature [16,48,52].

Strategy Representation. In graph games, the strategies are the most important objects as they represent the witness to winning of a player. For example, winning strategies represent controllers in the controller synthesis problem. Hence all parity-games solvers produce the winning strategies as their output. While the algorithmic problem of solving parity games has received huge attention, quite surprisingly, data-structures for representation of strategies have received little attention. While the data-structures for strategies could be relevant in particular algorithms for parity games (e.g., strategy-iteration algorithm), our focus is very different than improving such algorithms. Our main focus is the representation of the strategies themselves, which are the main output of the parity-games

solvers, and hence our strategy representation serves as post-processing of the output of the solvers. The standard data-structure for representing strategies is binary decision diagrams (BDDs) [2,13] and it is used as follows: a strategy is interpreted as a lookup table of pairs that specifies for every controlled state of the player the transition to choose, and then the lookup table is represented as a binary decision diagram (BDD).

Strategies as BDDs. The desired properties of data-structures for strategies are as follows: (a) *succinctness*, i.e., small strategies are desirable, since strategies correspond to controllers, and smaller strategies represent efficient controllers that are required in resource-constrained environments such as embedded systems; (b) *explanatory*, i.e., the representation explains the decisions of the strategies. In this work we consider different data-structure for representation of strategies in graph games. The key drawbacks of BDDs to represent strategies in graph games are as follows. First, the size of BDDs crucially depends on the variable ordering. The variable ordering problem is notoriously difficult: the optimal variable ordering problem is NP-complete, and for large dimensions no heuristics are known to work well. Second, due to the fact that strategies have to be input to the BDD construction as Boolean formulae, the representation though succinct, does not retain the inherent important choice features of the decisions of the strategies (for an illustration see Example 2).

Strategies as Decision Trees. In this work, we propose to use *decision trees*, i.e. [38], for strategy representation in graph games. A decision tree is a structure similar to a BDD, but with nodes labelled by various predicates over the system's variables. In the basic algorithm for decision trees, the tree is constructed using an unfolding procedure where the branching for the decision making is done in order to maximize the information gain at each step.

The key advantages of decision trees over BDDs are as follows:

- The first two advantages are conceptual. First, while in BDDs, a level corresponds to one variable, in decision trees, a predicate can appear at different levels and different predicates can appear at the same level. This allows for more flexibility in the representation. Second, decision trees utilize various predicates over the given features in order to make decisions, and ignore all the unimportant features. Thus they retain the inherent flavor of the decisions of the strategies.
- The other important advantage is algorithmic. Since the data-structure is based on information gain, sophisticated algorithms based on entropy exist for their construction. These algorithms result in a succinct representation, whereas for BDDs there is no good algorithmic approach for variable reordering.

Key Challenges. While there are several advantages of decision trees, and decision trees have been extensively studied in the machine learning community, there are several key challenges and obstacles for representation of strategies in graph games by decision trees.

– First, decision trees have been mainly used in the probabilistic setting. In such settings, research from the machine learning community has developed techniques to show that decision trees can be effectively pruned to obtain succinct trees, while allowing small error probabilities. However, in the context of graph games, no error is allowed in the strategic choices.
– Second, decision trees have been used in the machine learning community in classification, where an important aspect is to ensure that there is no overfitting of the training data. In contrast, in the context of graph games, the decision tree must fit the entire representation of the strategies.

While for probabilistic models such as Markov decision processes (MDPs), decision trees can be used as a blackbox [9], in the setting of graph games their use is much more challenging. In summary, in previous settings where decision trees are used small error rates are allowed in favor of succinctness, and overfitting is not permitted, whereas in our setting no error is allowed, and the complete fitting of the tree has to be ensured. The basic algorithm for decision-tree learning (called ID3 algorithm [38,44]) suffers from the curse of dimensionality, and the error allowance is used to handle the dimensionality. Hence we need to develop new techniques for strategy learning with decision trees in graph games.

Our Techniques. We present a new technique for learning strategies with decision trees based on *look-ahead*. In the basic algorithm for decision trees, at each step of the unfolding, the algorithm proceeds as long as there is any information gain. However, suppose for no possible branching there is any information gain. This represents the situation where the local (i.e., one-step based) decision making fails to achieve information gain. We extend this process so that look-ahead is allowed, i.e., we consider possible information gain with multiple steps. The look-ahead along with complete unfolding ensure that there is no error in the strategy representation. While the look-ahead approach provides a systematic principle to obtain precise strategy representation, it is computationally expensive, and we present heuristics used together with look-ahead for computational efficiency and succinctness of strategy representation.

Implementation and Experimental Results. Since in our setting existing decision tree solvers cannot be used as a blackbox, we extended the existing solvers with our techniques mentioned above. We have then applied our implementation to compare decision trees and BDDs for representation of strategies for problems in reactive synthesis. First, we compared our approach against BDDs for two classical examples of reactive synthesis from SYNTCOMP benchmarks [32]. Second, we considered randomly generated LTL formulae, and the graph games obtained for the realizability of such formulae. In both the above experiments the decision trees represent the winning strategies much more efficiently as compared to BDDs.

Related Work. Previous non-explicit representation of strategies for verification or synthesis purposes typically used BDDs [51] or automata [39,41] and do not explain the decisions by the current valuation of variables. *Decision trees* have been used a lot in the area of machine learning as a classifier that naturally

explains a decision [38]. They have also been considered for approximate representation of values in states and thus implicitly for an approximate representation of *strategies*, for the model of Markov decision processes (MDPs) in [7,8]. Recently, in the context of verification, this approach has been modified to capture strategies guaranteed to be ε-optimal, for MDPs [9] and partially observable MDPs [10]. Learning a compact decision tree representation of an MDP strategy was also investigated in [35] for the case of body sensor networks. Besides, decision trees are becoming more popular in verification and programming languages in general, for instance, they are used to capture program invariants [27,34]. To the best of our knowledge, decision trees were only used in the context of (possibly probabilistic) systems with only a single player. Our decision-tree approach is thus the first in the game setting with two players that is required in reactive synthesis.

Summary. To summarize, our main contributions are:

1. We propose decision trees as data-structure for strategy representation in graph games.
2. The representation of strategies with decision trees poses many obstacles, as in contrast to the probabilistic setting no error is allowed in games. We present techniques that overcome these obstacles while still retaining the algorithmic advantages (such as entropy-based methods) of decision trees to obtain succinct decision trees.
3. We extend existing decision tree solvers with our techniques and present experimental results to demonstrate the effectiveness of our approach in reactive synthesis.

Further details and proofs can be found in [12].

2 Graph Games and Strategies

Graph Games. A *graph game* consists of a tuple $G = \langle S, S_1, S_2, A_1, A_2, \delta \rangle$, where:

- S is a finite set of states partitioned into player 1 states S_1 and player 2 states S_2;
- A_1 (resp., A_2) is the set of actions for player 1 (resp., player 2); and
- $\delta\colon (S_1 \times A_1) \cup (S_2 \times A_2) \to S$ is the transition function that given a player 1 state and a player 1 action, or a player 2 state and a player 2 action, gives the successor state.

Plays. A *play* is an infinite sequence of state-action pairs $\langle s_0 a_0 s_1 a_1 \ldots \rangle$ such that for all $j \geq 0$ we have that if $s_j \in S_i$ for $i \in \{1, 2\}$, then $a_j \in A_i$ and $\delta(s_j, a_j) = s_{j+1}$. We denote by $\mathsf{Plays}(G)$ the set of all plays of a graph game G.

Strategies. A strategy is a recipe for a player to choose actions to extend finite prefixes of plays. Formally, a strategy π for player 1 is a function $\pi\colon S^\star \cdot S_1 \to A_1$

that given a finite sequence of visited states chooses the next action. The definitions for player 2 strategies γ are analogous. We denote by $\Pi(G)$ and $\Gamma(G)$ the set of all strategies for player 1 and player 2 in graph game G, respectively. Given strategies $\pi \in \Pi(G)$ and $\gamma \in \Gamma(G)$, and a starting state s in G, there is a unique play $\varrho(s, \pi, \gamma) = \langle s_0 a_0 s_1 a_1 \ldots \rangle$ such that $s_0 = s$ and for all $j \geq 0$ if $s_j \in S_1$ (resp., $s_j \in S_2$) then $a_j = \pi(\langle s_0 s_1 \ldots s_j \rangle)$ (resp., $a_j = \gamma(\langle s_0 s_1 \ldots s_j \rangle)$). A *memoryless* strategy is a strategy that does not depend on the finite prefix of the play but only on the current state, i.e., functions $\pi \colon S_1 \to A_1$ and $\gamma \colon S_2 \to A_2$.

Objectives. An *objective* for a graph game G is a set $\varphi \subseteq \mathsf{Plays}(G)$. We consider the following objectives:

- *Reachability and safety objectives.* A reachability objective is defined by a set $T \subseteq S$ of target states, and the objective requires that a state in T is visited at least once. Formally, $\mathsf{Reach}(F) = \{\langle s_0 a_0 s_1 a_1 \ldots \rangle \in \mathsf{Plays}(G) \mid \exists i : s_i \in T\}$. The dual of reachability objectives are safety objectives, defined by a set $F \subseteq S$ of safe states, and the objective requires that only states in F are visited. Formally, $\mathsf{Safe}(F) = \{\langle s_0 a_0 s_1 a_1 \ldots \rangle \in \mathsf{Plays}(G) \mid \forall i : s_i \in F\}$.
- *Parity objectives.* For an infinite play ϱ we denote by $\mathsf{Inf}(\varrho)$ the set of states that occur infinitely often in ϱ. Let $p \colon S \to \mathbb{N}$ be a *priority function.* The *parity* objective $\mathsf{Parity}(p) = \{\varrho \in \mathsf{Plays}(G) \mid \min\{p(s) \mid s \in \mathsf{Inf}(\varrho)\} \text{ is even } \}$ requires that the minimum of the priorities of the states visited infinitely often be even.

Winning Region and Strategies. Given a game graph G and an objective φ, a *winning* strategy π from state s for player 1 is a strategy such that for all strategies $\gamma \in \Gamma(G)$ we have $\varrho(s, \pi, \gamma) \in \varphi$. Analogously, a winning strategy γ for player 2 from s ensures that for all strategies $\pi \in \Pi(G)$ we have $\varrho(s, \pi, \gamma) \notin \varphi$. The *winning region* $W_1(G, \varphi)$ (resp., $W_2(G, \overline{\varphi})$) for player 1 (resp., player 2) is the set of states such that player 1 (resp., player 2) has a winning strategy. A fundamental result for graph games with parity objectives shows that the winning regions form a partition of the state space, and if there is a winning strategy for a player, then there is a memoryless winning strategy [25].

LTL Synthesis and Objectives. Reachability and safety objectives are the most basic objectives to specify properties of reactive systems. Most properties that arise in practice for analysis of reactive systems are ω-regular objectives. A convenient logical framework to express ω-regular objectives is the LTL (linear-time temporal logic) framework. The problem of synthesis from specifications, in particular, LTL synthesis has received huge attention [18]. LTL objectives can be translated to parity automata, and the synthesis problem reduces to solving games with parity objectives.

In reactive synthesis it is natural to consider games where the state space is defined by a set of variables, and the game is played by input and output player who choose the respective input and output signals. We describe such games below that easily correspond to graph games.

I/O Games with Variables. Consider a finite set $X = \{x_1, x_2, \ldots, x_n\}$ of variables from a finite domain; for simplicity, we consider Boolean variables only. A *valuation* is an assignment to each variable, in our case 2^X denotes the set of all valuations. Let X be partitioned into input signals, output signals, and state variables, i.e., $X = I \uplus O \uplus V$. Consider the alphabet $\mathcal{I} = 2^I$ (resp., $\mathcal{O} = 2^O$) where each letter represents a subset of the input (resp., output) signals and the alphabet $\mathcal{V} = 2^V$ where each letter represents a subset of state variables. The input/output choices affect the valuation of the variables, which is given by the next-step valuation function $\Delta\colon \mathcal{V} \times \mathcal{I} \times \mathcal{O} \to \mathcal{V}$. Consider a game played as follows: at every round the input player chooses a set of input signals (i.e., a letter from $\mathcal{I}$), and given the input choice the output player chooses a set of output signals (i.e., a letter from $\mathcal{O}$). The above game can be represented as a graph game $\langle S, S_1, S_2, A_1, A_2, \delta\rangle$ as follows:

- $S = \mathcal{V} \cup (\mathcal{V} \times \mathcal{I})$;
- player 1 represents the input player and $S_1 = \mathcal{V}$; player 2 represents the output player and $S_2 = \mathcal{V} \times \mathcal{I}$;
- $A_1 = \mathcal{I}$ and $A_2 = \mathcal{O}$; and
- given a valuation $v \in \mathcal{V}$ and $a_1 \in A_1$ we have $\delta(v, a_1) = (v, a_1)$, and for $a_2 \in A_2$ we have $\delta((v, a_1), a_2) = \Delta(v, a_1, a_2)$.

In this paper, we use decision trees to represent memoryless strategies in such graph games, where states are represented as vectors of Boolean values. In Sect. 5 we show how such games arise from various sources (AIGER specifications [31], LTL synthesis) and why it is sufficient to consider memoryless strategies only.

3 Decision Trees and Decision Tree Learning

In this section we recall decision trees and learning decision trees. A key application domain of games on graphs is reactive synthesis (such as safety synthesis from SYNTCOMP benchmarks as well as LTL synthesis) and our comparison for strategy representation is against BDDs. BDDs are particularly suitable for states and actions represented as bitvectors. Hence for a fair comparison against BDDs, we consider a simple version of decision trees over bitvectors, though decision trees and their corresponding methods can be naturally extended to richer domains (such as vectors of integers as used in [9]).

Decision Trees. A *decision tree* over $\{0,1\}^d$ is a tuple $\mathcal{T} = (T, \rho, \theta)$ where T is a finite rooted binary (ordered) tree with a set of inner nodes N and a set of leaves L, ρ assigns to every inner node a number of $\{1, \ldots, d\}$, and θ assigns to every leaf a value *YES* or *NO*.

The language $\mathcal{L}(\mathcal{T}) \subseteq \{0,1\}^d$ of the tree is defined as follows. For a vector $\boldsymbol{x} = (x_1, \ldots, x_d) \in \{0,1\}^d$, we find a path p from the root to a leaf such that for each inner node n on the path, $\boldsymbol{x}(\rho(n)) = 0$ iff the first child of n is on p. Denote the leaf on this particular path by ℓ. Then $\boldsymbol{x}$ is in the language $\mathcal{L}(\mathcal{T})$ of $\mathcal{T}$ iff $\theta(\ell) = \mathit{YES}$.

Example 1. Consider dimension $d = 3$. The language of the tree depicted in Fig. 1 can be described by the following regular expression $\{0,1\}^2 \cdot 0 + \{0,1\} \cdot 1 \cdot 1$. Intuitively, the root node represents the predicate of the third value, the other inner node represents the predicate of the second value. For each inner node, the first and second children correspond to the cases where the value at the position specified by the predicate of the inner node is 0 and 1, respectively. We supply the edge labels to depict the tree clearly. The leftmost leaf corresponds to the subset of $\{0,1\}^3$ where the third value is 0, the rightmost leaf corresponds to the subset of $\{0,1\}^3$ where the third value is 1 and the second value is 1.

Standard DT Learning. We describe the standard process of binary classification using decision trees (see Algorithm 1). Given a *training set* $Train \subseteq \{0,1\}^d$, partitioned into two subsets *Good* and *Bad*, the process of learning according to the algorithm ID3 [38,44] computes a decision tree $\mathcal{T}$ that assigns *YES* to all elements of *Good* and *NO* to all elements of *Bad*. In the algorithm, a leaf $\ell \subseteq \{0,1\}^d$ is *mixed* if ℓ has a non-empty intersection with both *Good* and *Bad*. To split a leaf ℓ on $bit \in \{1,\ldots,d\}$ means that ℓ becomes an internal node with the two new leaves ℓ_0 and ℓ_1 as its children. Then, the leaf ℓ_0 contains the samples of ℓ where the value in the position *bit* equals 0, and the leaf ℓ_1 contains the rest of the samples of ℓ, since these have the value in the position *bit* equal to 1. The *entropy* of a node is defined as

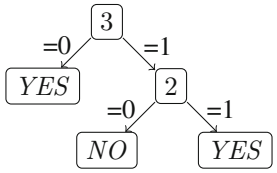

Fig. 1. A decision tree over $\{0,1\}^3$

$$H(\ell) = -\frac{|\ell \cap Good|}{|\ell|} log_2 \frac{|\ell \cap Good|}{|\ell|} - \frac{|\ell \cap Bad|}{|\ell|} log_2 \frac{|\ell \cap Bad|}{|\ell|}$$

An *information gain* of a given $bit \in \{1,\ldots,d\}$ (and thus also of the split into ℓ_0 and ℓ_1) is defined by

$$H(\ell) - \frac{|\ell_0|}{|\ell|} H(\ell_0) - \frac{|\ell_1|}{|\ell|} H(\ell_1) \tag{1}$$

where ℓ_0 is the set of all $\boldsymbol{x} = (x_1,\ldots,x_d) \in \ell \subseteq \{0,1\}^d$ with $x_{bit} = 0$ and $\ell_1 = \ell \setminus \ell_0$. Finally, given $\ell \subseteq \{0,1\}^d$ we define

$$maxclass(\ell) = \begin{cases} YES & |\ell \cap Good| \geq |\ell \cap Bad| \\ NO & \text{otherwise.} \end{cases}$$

Intuitively, the splitting on the component with the highest gain splits the set so that it maximizes the portion of *Good* in one subset and the portion of *Bad* in the other one.

Remark 1 (Optimizations). The basic ID3 algorithm for decision tree learning suffers from the curse of dimensionality. However, decision trees are primarily applied to machine learning problems where small errors are allowed to obtain succinct trees. Hence the allowance of error is crucially used in existing solvers (such as WEKA [29]) to combat dimensionality. In particular, the error rate is

Algorithm 1. ID3 learning algorithm

Inputs: $\mathit{Train} \subseteq \{0,1\}^d$ partitioned into subsets *Good* and *Bad*.
Outputs: A decision tree $\mathcal{T}$ such that $\mathcal{L}(\mathcal{T}) \cap \mathit{Train} = \mathit{Good}$.
/* train $\mathcal{T}$ on positive set *Good* and negative set *Bad* */
1: $\mathcal{T} \leftarrow (\{\mathit{Train}\}, \emptyset, \{\mathit{Train} \mapsto^{\theta} \mathit{YES}\})$
2: **while** a mixed leaf ℓ exists **do**
3: $\quad \mathit{bit} \leftarrow$ an element of $\{1, \ldots, d\}$ that maximizes the information gain
4: $\quad$ split ℓ on *bit* into two leaves ℓ_0 and ℓ_1, $\rho(\ell) = \mathit{bit}$
5: $\quad \theta(\ell_0) \leftarrow \mathit{maxclass}(\ell_0)$ and $\theta(\ell_1) \leftarrow \mathit{maxclass}(\ell_1)$
6: **return** $\mathcal{T}$

exploited in the unfolding, where the unfolding proceeds only when the information gain exceeds the error threshold. Further error is also introduced in the pruning of the trees, which ensures that the overfitting of training data is avoided.

4 Learning Winning Strategies Efficiently

In this section we present our contributions. We first start with the representation of strategies as training sets, and then present our strategy decision-tree learning algorithm.

4.1 Strategies as Training Sets and Decision Trees

Strategies as Training Sets. Let us consider a game $G = \langle S, S_1, S_2, A_1, A_2, \delta \rangle$. We represent strategies of both players using the same method. So in what follows we consider either of the players and denote by S_* and A_* the sets of states and actions of the player, respectively. We fix $\tilde{\sigma} \colon S_* \to A_*$, a memoryless strategy of the player.

We assume that G is an I/O game with binary variables, which means $S_* \subseteq \{0,1\}^n$ and $A_* \subseteq \{0,1\}^a$. A memoryless strategy is then a partial function $\tilde{\sigma} \colon \{0,1\}^n \to \{0,1\}^a$. Furthermore, we fix an initial state s_0, and let $S_*^R \subseteq \{0,1\}^n$ be the set of all states reachable from s_0 using σ against some strategy of the other player. We consider all objectives only on plays starting in the initial state s_0. Therefore, the strategy can be seen as a function $\sigma \colon S_*^R \to A_*$ such that $\sigma = \tilde{\sigma}|_{S_*^R}$.

Now we define

- $\mathit{Good} = \{\langle s, \sigma(s) \rangle \in S_*^R \times A_*\}$
- $\mathit{Bad} = \{\langle s, a \rangle \in S_*^R \times A_* \mid a \neq \sigma(s)\}$

The set of all training examples is a disjunctive union $\mathit{Train} = \mathit{Good} \uplus \mathit{Bad} \subseteq \{0,1\}^{n+a}$.

As we do not use any pruning or stopping rules, the ID3 algorithm returns a decision tree $\mathcal{T}$ that fits the training set *Train* exactly. This means that for all

$s \in S^R_*$ we have that $\langle s, a\rangle \in \mathcal{L}(\mathcal{T})$ iff $\sigma(s) = a$. Thus $\mathcal{T}$ represents the strategy σ. Note that for any sample of $\{0,1\}^{n+a} \setminus \mathit{Train}$, the fact whether it belongs to $\mathcal{L}(\mathcal{T})$ or not is immaterial to us. Thus strategies are naturally represented as decision trees, and we present an illustration below.

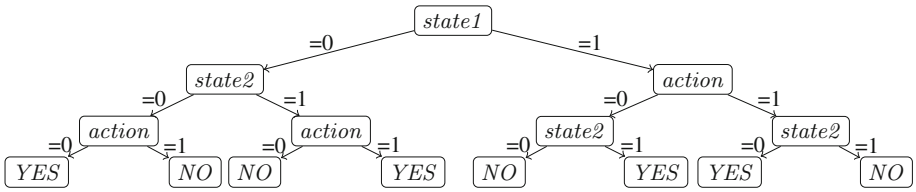

Fig. 2. Tree representation of strategy σ

Example 2. Let the state binary variables be labeled as *state1*, *state2*, and *state3*, respectively, and let the action binary variable be labeled as *action*. Consider a strategy σ such that $\sigma(0,0,0) = 0$, $\sigma(0,1,0) = 1$, $\sigma(1,0,0) = 1$, $\sigma(1,1,1) = 0$. Then

- $\mathit{Good} = \{(0,0,0,0),(0,1,0,1),(1,0,0,1),(1,1,1,0)\}$
- $\mathit{Bad} = \{(0,0,0,1),(0,1,0,0),(1,0,0,0),(1,1,1,1)\}$

Figure 2 depicts a decision tree $\mathcal{T}$ representing the strategy σ.

Remark 2. The above example demonstrates the conceptual advantages of decision trees over BDDs. First, in decision trees, different predicates can appear at the same level of the tree (e.g. predicates *state2* and *action* appear at the second level). At the same time, a predicate can appear at different levels of the tree (e.g. predicate *action* appears once at the second level and twice at the third level).

Second advantage is a bit technical, but very crucial. In the example there is no pair of samples $g \in \mathit{Good}$ and $b \in \mathit{Bad}$ that differs only in the value of *state3*. This suggests that the feature *state3* is unimportant w.r.t. differentiating between *Good* and *Bad*, and indeed the decision tree $\mathcal{T}$ in Fig. 2 contains no predicate *state3* while still representing σ. However, to construct a BDD that ignores *state3* is very difficult, since a Boolean formula is provided as the input to the BDD construction, and this formula inevitably sets the value for every sample. Therefore, it is impossible to declare "the samples of $\{0,1\}^{n+a} \setminus \mathit{Train}$ can be resolved either way". One way to construct a BDD $\mathcal{B}$ would be $\mathcal{B} \equiv \bigvee_{g \in \mathit{Good}} g$. But then $\mathcal{B}(0,0,0,0) = 1$ and $\mathcal{B}(0,0,1,0) = 0$, so *state3* has to be used in the representation of $\mathcal{B}$. Another option could be $\mathcal{B} \equiv \bigwedge_{b \in \mathit{Bad}} \neg b$, but then $\mathcal{B}(0,0,0,1) = 0$ and $\mathcal{B}(0,0,1,1) = 1$, so *state3* still has to be used in the representation.

Example 3. Consider $\mathit{Good} = \{(0,0,0,0,1)\}$ and $\mathit{Bad} = \{(0,0,0,0,0)\}$. Algorithm 1 outputs a simple decision tree differentiating between *Good* and *Bad* only according to the value of the last variable. On the other hand, a BDD constructed as $\mathcal{B} \equiv \bigvee_{g \in \mathit{Good}} g$ contains nodes for all five variables.

4.2 Strategy-DT Learning

Challenges. In contrast to other machine learning domains, where errors are allowed, since strategies in graph games must be represented precisely, several challenges arise. Most importantly, the machine-learning philosophy of classifiers is to generalize the experience, trying to achieve good predictions on any (not just training) data. In order to do so, overfitting the training data must be avoided. Indeed, specializing the classifier to cover the training data precisely leads to classifiers reflecting the concrete instances of random noise instead of generally useful predictors. Overfitting is prevented using a tolerance on learning all details of the training data. Consequently, the training data are not learnt exactly. Since in our case, the training set is exactly what we want to represent, our approach must be entirely different. In particular, the optimizations in the setting where errors are allowed (see Remark 1) are not applicable to handle the curse of dimensionality. In particular, it may be necessary to unfold the decision tree even in situations where none of the one-step unfolds induces any information gain.

Solution: Look-Ahead. In the ID3 algorithm Algorithm 1, when none of the splits has a positive information gain (see Formula (1)), the corresponding node is split arbitrarily. This can result in very large decision trees. We propose a better solution. Namely, we extend ID3 with a *"look-ahead"*: If no split results in a positive information gain, one can pick a split so that next, when splitting the children, the information gain is positive. If still no such split exists, one can try and pick a split and splits of children so that afterwards there is a split of grandchildren with positive information gain. And so on, possibly until a constant depth k, yielding a *k-look-ahead*.

Before we define the look-ahead formally, we have a look at a simple example:

Example 4. Consider $Good = \{(0,0,0,0,0,1,1),(0,0,0,0,0,0,0)\}$ and $Bad = \{(0,0,0,0,0,1,0),(0,0,0,0,0,0,1)\}$, characterising $x_6 = x_7$. Splitting on any x_i, $i \in \{1,...,7\}$ does not give a positive information gain. Standard DT learning procedures would either stop here and not expand this leaf any more, or split arbitrarily. With the look-ahead, one can see that using x_6 and then x_7, the information gain is positive and we obtain a decision tree classifying the set perfectly.

Here we could as well introduce more complex predicates such as x_6 `xor` x_7 instead of look-ahead. However, in general the look-ahead has the advantage that each of the 0 and 1 branches may afterwards split on different bits (currently best ones), whereas with x_6 `xor` x_7 we commit to using x_7 in both branches.

The example illustrates the 2-look-ahead with the following formal definition. (For explanatory reasons, the general case follows afterwards.) Consider a node $\ell \subseteq \{0,1\}^d$. For every $bit, bit_0, bit_1 \in \{1,\dots,d\}$, consider splitting on bit and subsequently the 0-child on bit_0 and the 1-child on bit_1. This results in a partition

$P(bit, bit_0, bit_1) = \{\ell_{00}, \ell_{01}, \ell_{10}, \ell_{11}\}$ of ℓ. We assign to $P(bit, bit_0, bit_1)$ its *2-look-ahead information gain* defined by

$$IG(bit, bit_0, bit_1) = H(\ell) - \frac{|\ell_{00}|}{|\ell|}H(\ell_{00}) - \frac{|\ell_{01}|}{|\ell|}H(\ell_{01}) - \frac{|\ell_{10}|}{|\ell|}H(\ell_{10}) - \frac{|\ell_{11}|}{|\ell|}H(\ell_{11})$$

The *2-look-ahead information gain of* $bit \in \{1, \ldots, d\}$ is defined as

$$IG(bit) = \max_{bit_0, bit_1} IG(bit, bit_0, bit_1)$$

We say that $bit \in \{1, \ldots, d\}$ *maximizes the 2-look-ahead information gain* if

$$bit \in \arg\max IG$$

In general, we define the *k-step weighted entropy* of a node $\ell \subseteq \{0,1\}^d$ with respect to a predicate $bit \in \{1, \ldots, d\}$ by

$$WE^k(\ell, bit) = \min_{bit_0, bit_1} WE^{k-1}(\{\boldsymbol{x} \in \ell \mid x_{bit} = 0\}, bit_0) + WE^{k-1}(\{\boldsymbol{x} \in \ell \mid x_{bit} = 1\}, bit_1)$$

and

$$WE^0(\ell, bit) = |\ell| \cdot H(\ell)$$

Then we say that $\hat{bit} \in \{1, \ldots, d\}$ *maximizes the k-look-ahead information gain in* ℓ if

$$\hat{bit} \in \arg\max_{bit \in \{1,\ldots,d\}} \big(H(\ell) - WE^k(\ell, bit)/|\ell|\big) = \arg\min WE^k(\ell, \cdot)$$

Note that 1-look-ahead coincides with the choice of split by ID3. For a fixed k, if the information gain for each i-look-ahead, $i \leq k$ is zero, we split based on a heuristic on Line 8 of Algorithm 2. This heuristic is detailed on in the following subsection. Note that Algorithm 2 is correct-by-construction since we enforce representation of the entire input training set. We present a formal correctness proof in [12, Appendix B].

Remark 3 (Properties of look-ahead algorithm). We now highlight some desirable properties of the look-ahead algorithm.

- *Incrementality.* First, the algorithm presents an incremental approach: computation of the k-look-ahead can be done by further refining the results of the $(k-1)$-look-ahead analysis due to the recursive nature of our definition. Thus the algorithm can start with $k = 2$ and increase k only when required.
- *Entropy-based minimization.* Second, the look-ahead approach naturally extends the predicate choice of ID3, and thus the entropy-based minimization for decision trees is still applicable.
- *Reduction of dimensionality.* Finally, Algorithm 2 uses the look-ahead method in an incremental fashion, thus only considering more complex "combinations" when necessary. Consequently, we do not produce all these combinations of predicates in advance, and avoid the problem of too high dimensionality and only experience local blowups.

Algorithm 2. k-look-ahead ID3

Inputs: $Train \subseteq \{0,1\}^d$ partitioned into subsets $Good$ and Bad.
Outputs: A decision tree $\mathcal{T}$ such that $\mathcal{L}(\mathcal{T}) \cap Train = Good$.
/* train $\mathcal{T}$ on positive set $Good$ and negative set Bad */
$\mathcal{T} \leftarrow (\{Train\}, \emptyset, \{Train \mapsto^{\theta} YES\})$
while a mixed leaf ℓ exists **do**
  if $\exists bit \in \{1,\ldots,d\}$ with a positive 1-look-ahead information gain **then**
    $bit \leftarrow$ an element of $\{1,\ldots,d\}$ that maximizes the 1-look-ahead information gain ▷ maximum information gain is positive
  ⋮
  else if $\exists bit \in \{1,\ldots,d\}$ with a positive k-look-ahead information gain **then**
    $bit \leftarrow$ an element of $\{1,\ldots,d\}$ that maximizes the k-look-ahead information gain ▷ maximum k-look-ahead information gain is positive
  else
    $bit \leftarrow \arg\max_{i\in\{1,..,d\}} \max\left\{\frac{|\ell[i=0]\cap Bad|}{|\ell[i=0]|} + \frac{|\ell[i=1]\cap Good|}{|\ell[i=1]|}, \frac{|\ell[i=0]\cap Good|}{|\ell[i=0]|} + \frac{|\ell[i=1]\cap Bad|}{\ell[i=1]|}\right\}$
  split ℓ on bit into two leaves ℓ_0 and ℓ_1, $\rho(\ell) = bit$
  $\theta(\ell_0) \leftarrow maxclass(\ell_0)$ and $\theta(\ell_1) \leftarrow maxclass(\ell_1)$
return $\mathcal{T}$

In general, k-look-ahead clearly requires resources exponential in k. However, in our benchmarks, it was typically sufficient to apply the look-ahead for k equal to two, which is computationally feasible.

A different look-ahead-based technique was considered in order to dampen the greedy nature of decision tree construction [24], examining the predicates yielding the highest information gains. In contrast, our technique retains the greedy approach but focuses on the case where none of the predicates provides any information gain itself at all and thus ID3-based techniques fail to advance. The main goal of our technique is to capture strong dependence between the features of the training set, in order to solve a different problem than the one treated by [24]. Moreover, the look-ahead description in [24] is very informal, which prevents us from implementing their solution and comparing the two approaches experimentally.

4.3 Heuristics

Statistical Split-Decision. The look-ahead mentioned above provides a very systematic principle on how to resolve splitting decisions. However, the computation can be demanding in terms of computational resources. Therefore we present a very simple statistical heuristic that gives us one more option to decide a split. The precise formula is $bit =$

$$\arg\max_{i\in\{1,..,d\}} \max\left\{\frac{|\ell[i=0]\cap Bad|}{|\ell[i=0]|} + \frac{|\ell[i=1]\cap Good|}{|\ell[i=1]|}, \frac{|\ell[i=0]\cap Good|}{|\ell[i=0]|} + \frac{|\ell[i=1]\cap Bad|}{\ell[i=1]|}\right\}$$

Intuitively, we choose a bit that maximizes the portion of good samples in one subset and the portion of bad samples in the other subset, which mimics the

entropy-based method, and at the same time is very fast to compute. One can consider using this heuristic exclusively every time the basic ID3-based splitting technique fails. However, in our experiments, using 2-look-ahead and then (once needed) proceeding with the heuristic yields better results, and is still computationally undemanding.

Chain Disjunction. The entropy-based approach favors the splits where one of the branches contains a completely resolved data set ($\ell_* \subseteq \mathit{Good}$ or $\ell_* \subseteq \mathit{Bad}$), as this provides notable information gain. Therefore, as the algorithm proceeds, it often happens that at some point multiple splits provide a resolved data set in one of the branches. We consider a heuristic that *chains* all such splits together and computes the information gain of the resulting disjunction. More specifically, when considering each *bit* as a split candidate (line 3 of Algorithm 2), we also consider (a) the disjunction of all bits that contain a subset of *Good* in either of the branches, and (b) the disjunction of bits containing a subset of *Bad* in a branch. Then we choose the candidate that maximizes the information gain. These two extra checks are very fast to compute, and can improve succinctness and readability of the decision trees substantially, while maintaining the fact that a decision tree fits its training set exactly. [12, Appendix D] provides two examples where the decision tree obtained without this heuristic is presented, and then the decision tree obtained when using the heuristic is presented.

5 Experimental Results

In our experiments we use two sources of problems reducible to the representation of memoryless strategies in I/O games with binary variables: AIGER specifications [31] and LTL specifications [42]. Given a game, we use an explicit solver to obtain a strategy in the form of a list of played and non-played actions for each state, which can be directly used as a training set. Throughout our experiments, we compare succinctness of representation (expressed as the number of inner nodes) using decision trees and BDDs.

We implemented our method in the programming language Java. We used the external library CuDD [49] for the manipulation of BDDs. We used the Algorithm 2 with $k = 2$ to compute the decision trees. We obtained all the results on a single machine with Intel(R) Core(TM) i5-6200U CPU (2.40 GHz) with the heap size limited to 8 GB.

5.1 AIGER Specifications

SYNTCOMP [32] is the most important competition of synthesis tools, running yearly since 2014. Most of the benchmarks have the form of AIGER specifications [31], describing safety specifications using circuits with input, output, and latch variables. This reduces directly to the I/O games with variables since the latches describe the current configuration of the circuit, corresponding to the

state variables of the game. Since the objectives here are safety/reachability, the winning strategies can be computed and guaranteed to be memoryless.

We consider two benchmarks: scheduling of washing cycles in a washing system and a simple bit shifter model (the latter presented only in [12, Appendix D] due to space constraints), introduced in SYNTCOMP 2015 [32] and SYNTCOMP 2014, respectively.

Scheduling of Washing Cycles. The goal is to design a centralized controller for a washing system, composed of several tanks running in parallel [32]. The model of the system is parametrized by the number of tanks, the maximum allowed reaction delay before filling a tank with water, the delay after which the tank has to be emptied again, and the number of tanks that share a water pipe. The controller should satisfy a safety objective, that is, avoid reaching an error state, which means that the objective of the other player is reachability. In total, we obtain 406 graph games with safety/reachability objectives. In 394 cases we represent a winning strategy of the safety player, in the remaining 12 cases a winning strategy of the reachability player. The number of states of the graph games ranges from 30 to 43203, the size of training example sets ranges from 40 to 3359232.

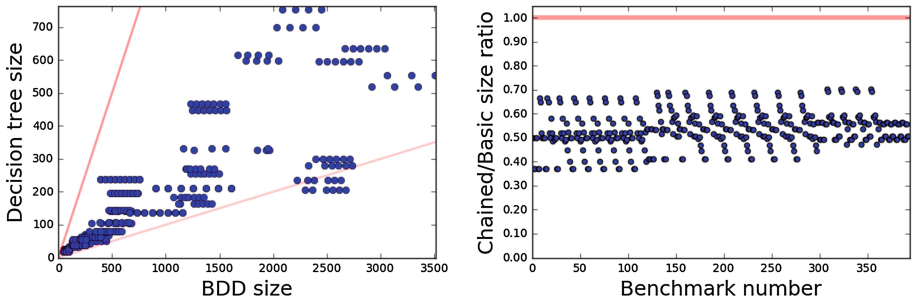

Fig. 3. Washing cycles – safety

The left plot in Fig. 3 displays the size of our decision tree representation of the controller winning safety strategies versus the size of their BDD representations. The decision tree is smaller than the corresponding BDD in all 394 cases. The arithmetic average ratio of decision tree size and BDD size is ∼24%, the geometric average is ∼22%, and the harmonic average is ∼21%.

In these experiments, we obtain the BDD representation as follows: we consider 1000 randomly chosen variable orderings and for each construct a corresponding BDD, in the end we consider the BDD with the minimal size. As a different set of experiments, we compare against BDDs obtained using several algorithms for variable reordering, namely, Sift [47], Window4 [26], simulated-annealing-based algorithm [6], and a genetic algorithm [22]. The results with

these algorithms are very similar and provided in [12, Appendix C]. Furthermore, the information about execution time is also provided in [12, Appendix C].

Moreover, in the experiments described above, we do not use the chain heuristic described in Sect. 4.3, in order to provide a fair comparison of decision trees and BDDs. The right plot in Fig. 3 displays the difference in decision tree size once the chain heuristic is enabled. Each dot represents the ratio of decision tree size with and without it.

The decision trees also allow us to get some insight into the winning strategies. Namely, for a fixed number of water tanks and a fixed empty delay, we obtain a solution that is affected by different values of the fill delay in a minimal way, and is easily generalizable for all the values of the parameter. This fact becomes more apparent once the chain heuristic described in Sect. 4.3 is enabled. This phenomenon is not present in the case of BDDs as they differ significantly, even in size, for different values of the parameter (see [12, Appendix C]). For two tanks and empty delay of one, the solution is small enough to be humanly readable and understandable, see Fig. 4 (where the fill delay is set to 7). Additional examples of the parametric solutions can be found in [12, Appendix C]. This example suggests that decision tree representation might be useful in solving parametrized synthesis (and verification) problems.

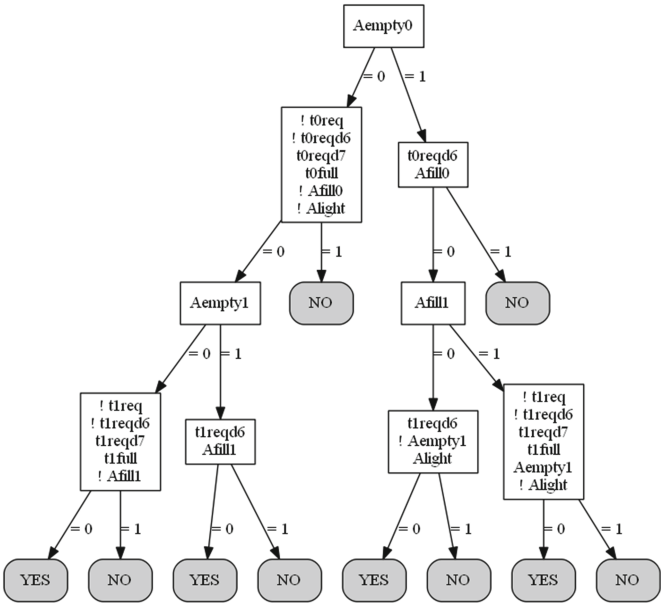

Fig. 4. A solution for two tanks and empty delay of one, illustration for fill delay of 7. Solution for other values p are the same except for replacing values p and $p-1$ for 7 and 6, respectively. Thus a parametric solution could be obtained by a simple syntactic analysis of the difference of any two instance solutions.

Name	$\|S\|$	$\|I\|$	$\|O\|$	$\|Train\|$	$\|BDD\|$	$\|DT\|$	$\|DT+\|$
wash_3_1_1_3	102	3	7	40	45	3	1
wash_4_1_1_3	466	4	9	144	76	4	1
wash_4_1_1_4	346	4	9	96	78	4	1
wash_4_2_1_4	958	4	9	432	157	4	1
wash_4_2_2_4	3310	4	9	432	301	4	1
wash_5_1_1_3	1862	5	11	416	127	5	1
wash_5_1_1_4	1630	5	11	352	121	5	1
wash_5_2_1_4	5365	5	11	2368	255	5	1
wash_5_2_2_4	27919	5	11	2368	554	5	1
wash_6_1_1_3	6962	6	13	1088	193	6	1
wash_6_1_1_4	6622	6	13	1024	172	6	1
wash_6_2_1_4	27412	6	13	10432	419	6	1

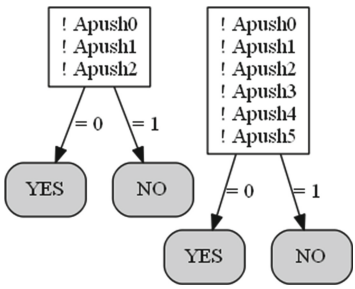

Fig. 5. Washing cycles – reachability

The table in Fig. 5 summarizes the results for the cases where the controller cannot be synthesized and we synthesize a counterexample winning reachability strategy of the environment. The benchmark parameters specify the total number of tanks, the fill delay, the empty delay, and the number of tanks sharing a pipe, respectively. In all of these cases, the size of the decision tree is substantially smaller compared to its BDD counterpart. The decision trees also provide some structural insight that may easily be used in debugging. Namely, the trees have a simple repeating structure where the number of repetitions depends just on the number of tanks. This is even easier to see once the chain heuristic of Sect. 4.3 is used. Figure 5 shows the tree solution for the case of three and six tanks, respectively. The structural phenomenon is not apparent from the BDDs at all.

5.2 Random LTL

In reactive synthesis, the objectives are often specified as LTL (linear-time temporal logic) formulae over input/output letters. In our experiments, we use formulae randomly generated using SPOT [23][1]. LTL formulae can be translated into deterministic parity automata; for this translation we use the tool Rabinizer [33]. Finally, given a parity automaton, we consider various partitions of the atomic propositions into input/output letters, which gives rise to graph games with parity objectives. See [12, Appendix F] for more details on the translation. We retain all formulae that result in games with at most three priorities.

Consequently, we use two ways of encoding states of the graph games as binary vectors. First, *naive encoding*, allowed by the fact that the output of tools such as [23,33] in HOA format [4] always assigns an id to each state.

[1] First, we run randltl from the Spot tool-set `randltl -n10000 5--tree-size=20..25 seed=0 --simplify=3 -p --ltl-priorities ap=3,false=1,true=1,not=1, F=1,G=1,X=1,equiv=1,implies=1,xor=0,R=0,U=1, W=0,M=0,and=1,or=1 | ltlfilt -- unabbreviate="eiMRW"` to obtain the formulae. Then we run Rabinizer to obtain the respective automata and we retain those with at least 100 states.

As this id is an integer, we may use its binary encoding. Second, we use a more sophisticated *Rabinizer encoding* obtained by using internal structure of states produced by Rabinizer [33]. Here the states are of the form "formula, set of formulae, permutation, priority". We propose a very simple, yet efficient procedure of encoding the state structure information into bitvectors. Although the resulting bitvectors are longer than in the naive encoding, some structural information of the game is preserved, which can be utilized by decision trees to provide a more succinct representation. BDDs perform a lot better on the naive encoding than on the Rabinizer encoding, since they are unable to exploit the preserved state information. As a result, we consider the naive encoding with BDDs and both, the naive and the Rabinizer encodings, with decision trees.

We consider 976 examples where the goal of the player, whose strategy is being represented, is that the least priority occurring an infinite number of times is odd.

Figure 6 plots the size ratios when we compare BDDs and decision trees (note that the y-axis scales logarithmically). For each case, we consider 1000 random variable orderings and choose the BDD that is minimal in size, and after that we construct a decision tree (without the chain heuristic of Sect. 4.3). For BDDs, we also consider all the ordering algorithms mentioned in the previous set of experiments, however, they provide no improvement compared to the random orderings.

In 925 out of 976 cases, the resulting decision tree is smaller than the corresponding BDD (in 3 cases they are of a same size and in 48 cases the BDD is smaller). The arithmetic average ratio of decision tree size and BDD size is ∼46%, the geometric average is ∼38%, and the harmonic average is ∼28%.

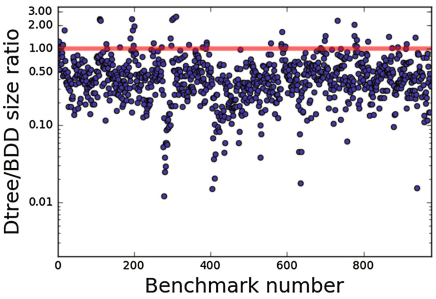

Fig. 6. BDDs vs DTrees

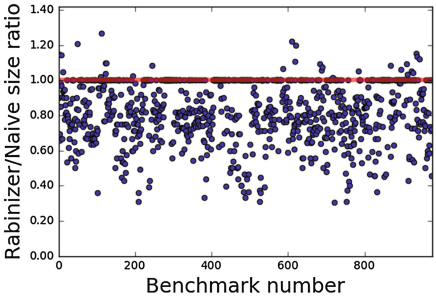

Fig. 7. DTrees improvement with Rabinizer enc.

Figure 7 demonstrates how decision tree representation improves once the features of the game-structural information can be utilized. Each dot corresponds to a ratio of the decision tree size once the Rabinizer encoding is used, and once the naive encoding is used. In 638 cases the Rabinizer encoding is superior, in 309 cases there is no difference, and in 29 cases the naive encoding is superior.

All three types of the average ratio are around 80%. In [12, Appendix E] we present the further improvement of decision trees once we use the chain heuristic of Sect. 4.3.

6 Conclusion

In this work we propose decision trees for strategy representation in graph games. While decision trees have been used in probabilistic settings where errors are allowed and overfitting of data is avoided, for graph games, strategies must be entirely represented without errors. Hence optimization techniques for existing decision-tree solvers do not apply, and we develop new techniques and present experimental results to demonstrate the effectiveness of our approach. Moreover, decision trees have several other advantages: First, in decision trees the nodes represent predicates, and in richer domains, e.g., where variables represent integers, the internal nodes of the tree can represent predicates in the corresponding domain, e.g., comparison between the integer variables and a constant. Hence richer domains can be directly represented as decision trees without conversion to bitvectors as required by BDDs. However, we restricted ourselves to the boolean domain to show that even in such domains that BDDs are designed for the decision trees improve over BDDs. Second, as illustrated in our examples, decision trees can often provide similar and scalable solution when some parameters vary. This is quite attractive in reactive synthesis where certain parameters vary, however they affect the strategy in a minimal way. Our examples show decision trees exploit this much better than BDDs, and can be useful in parametrized synthesis. Our work opens up many interesting directions of future work. For instance, richer versions of decision trees that are still well-readable could be used instead, such as decision trees with more complex expressions in leaves [40]. The applications of decision trees in other applications related to reactive synthesis is an interesting direction of future work. Another interesting direction is the application of the look-ahead technique in the probabilistic settings.

Data Availability Statement and Acknowledgments. This work has been partially supported by the Czech Science Foundation, Grant No. P202/12/G061, Vienna Science and Technology Fund (WWTF) Project ICT15-003, Austrian Science Fund (FWF) NFN Grant No. S11407-N23 (RiSE/SHiNE), ERC Starting grant (279307: Graph Games), DFG Grant No KR 4890/2-1 (SUV: Statistical Unbounded Verification), TUM IGSSE Grant 10.06 (PARSEC) and EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant No. 665385. We thank Fabio Somenzi for detailed information about variable reordering in BDDs. The source code and binary files used to obtain the results presented in this paper are available in the figshare repository: https://doi.org/10.6084/m9.figshare.5923915.v1 [11].

Appendix

A Artifact Description

We provide instructions to replicate the experimental results presented in this paper, using our artifact that is openly available at [11]. All the results can be obtained with the heap size limited to 8 GB.

Results for Scheduling of Washing Cycles (Sect. 5.1). Running this batch takes roughly 30 h and generates 7.1 GB of training data. Note that we did not include around 30 most resource-demanding benchmarks of this batch in the artifact. (i) in folder art, execute ./run.sh wTOTAL, (ii) observe the results at art/results/reports/reprWash{2,3,4,reach}.txt, (iii) in folder art/results, execute python plotsWash.py and observe the plots that correspond to Fig. 3. Alternatively, to run a subset of this batch that takes only 30 min to run and generates only 265MB of training data, in (i) execute ./run.sh wPART. To additionaly generate dot representation of DTs/BDDs, in (i) execute either ./run.sh wTOTALdot or ./run.sh wPARTdot.

Results for Scheduling of Washing Cycles BDD Reordering ([12, Appendix C]). Running this batch takes roughly 30 min. (i) make sure you have the training data obtained by running the batch above, (ii) in folder art/results, execute ./runBDDreorder.sh, (iii) observe the results at art/results/reports/BDDreorder.txt.

Results for Random LTL (Sect. 5.2). Running this batch takes roughly 2 h and generates 84 MB of training data. (i) in folder art, execute ./run.sh rTOTAL, (ii) observe the results at art/results/reports/reprRandomLTL{naive,encoded}.txt, (iii) in folder art/results, execute python plotsRandomLTL.py and observe the plots that correspond to Figs. 6 and 7.

Results for Bit Shifter ([12, Appendix D]). Running this experiment batch takes roughly 5 min. Note that we did not include two benchmarks in the artifact since they take considerable execution time. (i) in folder art, execute ./run.sh aTOTAL, (ii) observe the results at art/results/reports/reprAiger.txt.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0035748
2. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
3. Alur, R., Henzinger, T., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**, 672–713 (2002)

4. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
5. Blass, A., Gurevich, Y., Nachmanson, L., Veanes, M.: Play to test. In: FATES 2005 (2005)
6. Bollig, B., Lbbing, M., Wegener, I.: Simulated annealing to improve variable orderings for OBDDs. Presented at the International Workshop on Logic Synthesis, Granlibakken, CA (1995)
7. Boutilier, C., Dearden, R.: Approximate value trees in structured dynamic programming. In: Saitta, L. (ed.) ICML, pp. 54–62. Morgan Kaufmann (1996)
8. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: IJCAI, pp. 1104–1113. Morgan Kaufmann (1995)
9. Brázdil, T., Chatterjee, K., Chmelík, M., Fellner, A., Křetínský, J.: Counterexample explanation by learning small strategies in Markov decision processes. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 158–177. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_10
10. Brázdil, T., Chatterjee, K., Chmelík, M., Gupta, A., Novotný, P.: Stochastic shortest path with energy constraints in POMDPs: (extended abstract). In: AAMAS, pp. 1465–1466 (2016)
11. Brázdil, T., Chatterjee, K., Křetínský, J., Toman, V.: Artifact and instructions to generate experimental results for TACAS 2018 paper Strategy Representation by Decision Trees in Reactive Synthesis. Figshare (2018). https://doi.org/10.6084/m9.figshare.5923915.v1
12. Brázdil, T., Chatterjee, K., Křetínský, J., Toman, V.: Strategy representation by decision trees in reactive synthesis. arXiv.org:1802.00758 (2018)
13. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
14. Büchi, J.: On a decision method in restricted second-order arithmetic. In: Nagel, E., Suppes, P., Tarski, A. (eds.) Proceedings of the First International Congress on Logic, Methodology, and Philosophy of Science 1960, pp. 1–11. Stanford University Press (1962)
15. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. AMS **138**, 295–311 (1969)
16. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: Hatami, H., McKenzie, P., King, V. (eds.) Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, 19–23 June 2017, pp. 252–263. ACM (2017). https://dblp.org/rec/bib/conf/stoc/2017
17. Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians, pp. 23–35. Institut Mittag-Leffler (1962)
18. Clarke, E., Henzinger, T., Veith, H. (eds.): Handbook of Model Checking. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-10575-8. Chapter: Games and Synthesis
19. de Alfaro, L., Henzinger, T.: Interface automata. In: FSE 2001, pp. 109–120. ACM (2001)
20. de Alfaro, L., Henzinger, T., Mang, F.: Detecting errors before reaching them. In: CAV 2000, pp. 186–201 (2000)
21. Dill, D.: Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits. The MIT Press, Cambridge (1989)

22. Drechsler, R., Becker, B., Gockel, N.: Genetic algorithm for variable ordering of OBDDs. Presented at the International Workshop on Logic Synthesis, Granlibakken, CA (1995)
23. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - a framework for LTL and ω-automata manipulation. In: ATVA, pp. 122–129 (2016)
24. Elomaa, T., Malinen, T.: On lookahead heuristics in decision tree learning. In: Zhong, N., Raś, Z.W., Tsumoto, S., Suzuki, E. (eds.) ISMIS 2003. LNCS (LNAI), vol. 2871, pp. 445–453. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39592-8_63
25. Emerson, E., Jutla, C.: Tree automata, mu-calculus and determinacy. In: FOCS 1991, pp. 368–377. IEEE (1991)
26. Fujita, M., Matsunaga, Y., Kakuda, T.: On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In: EURO-DAC, pp. 50–54 (1991)
27. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL (2016)
28. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: STOC 1982, pp. 60–65. ACM Press (1982)
29. Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explor. **11**(1), 10–18 (2009)
30. Henzinger, T., Kupferman, O., Rajamani, S.: Fair simulation. I&C **173**, 64–81 (2002)
31. Jacobs, S.: Extended AIGER format for synthesis. CoRR, abs/1405.5793 (2014)
32. Jacobs, S., Bloem, R., Brenguier, R., Könighofer, R., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The second reactive synthesis competition (SYNTCOMP 2015). In: SYNT, pp. 27–57 (2015)
33. Komárková, Z., Křetínský, J.: Rabinizer 3: safraless translation of LTL to small deterministic automata. In: ATVA, pp. 235–241 (2014)
34. Krishna, S., Puhrsch, C., Wies, T.: Learning invariants using decision trees. CoRR, abs/1501.04725 (2015)
35. Liu, S., Panangadan, A., Raghavendra, C.S., Talukder, A.: Compact representation of coordinated sampling policies for body sensor networks. In: Proceedings of Workshop on Advances in Communication and Networks (Smart Homes for Tele-Health), pp. 6–10. IEEE (2010)
36. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1992). https://doi.org/10.1007/978-1-4612-0931-7
37. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Log. **65**, 149–184 (1993)
38. Mitchell, T.M.: Machine Learning, 1st edn. McGraw-Hill Inc., New York (1997)
39. Neider, D.: Small strategies for safety games. In: ATVA, pp. 306–320 (2011)
40. Neider, D., Saha, S., Madhusudan, P.: Synthesizing piece-wise functions by learning classifiers. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 186–203. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_11
41. Neider, D., Topcu, U.: An automaton learning approach to solving safety games over infinite graphs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 204–221. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_12

42. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE Computer Society Press (1977)
43. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989, pp. 179–190. ACM Press (1989)
44. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986)
45. Rabin, M.: Automata on Infinite Objects and Church's Problem. Number 13 in Conference Series in Mathematics. American Mathematical Society, Providence (1969)
46. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete-event processes. SIAM J. Contr. Opt. **25**(1), 206–230 (1987)
47. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: ICCAD, pp. 42–47. IEEE Computer Society Press (1993)
48. Schewe, S.: Solving parity games in big steps. JCSS **84**, 243–262 (2017)
49. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0 (2015)
50. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 389–455. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6_7
51. Wimmer, R., Braitling, B., Becker, B., Hahn, E.M., Crouzen, P., Hermanns, H., Dhama, A., Theel, O.: Symblicit calculation of long-run averages for concurrent probabilistic systems. In: QEST, pp. 27–36. IEEE Computer Society, Washington, DC (2010)
52. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1–2), 135–183 (1998)