Guessing Winning Policies in LTL Synthesis by Semantic Learning

Jan Křetínský[1,2(✉)], Tobias Meggendorfer[1,3], Maximilian Prokop[1,2], and Sabine Rieder[1]

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

Abstract. We provide a learning-based technique for guessing a winning strategy in a parity game originating from an LTL synthesis problem. A cheaply obtained guess can be useful in several applications. Not only can the guessed strategy be applied as best-effort in cases where the game's huge size prohibits rigorous approaches, but it can also increase the scalability of rigorous LTL synthesis in several ways. Firstly, checking whether a guessed strategy is winning is easier than constructing one. Secondly, even if the guess is wrong in some places, it can be fixed by strategy iteration faster than constructing one from scratch. Thirdly, the guess can be used in on-the-fly approaches to prioritize exploration in the most fruitful directions.

In contrast to previous works, we (i) reflect the highly structured logical information in game's states, the so-called semantic labelling, coming from the recent LTL-to-automata translations, and (ii) learn to reflect it properly by learning from previously solved games, bringing the solving process closer to human-like reasoning.

1 Introduction

LTL Synthesis. [38] is a framework for automatic construction of reactive systems specified by formulae of linear temporal logic (LTL) [37]. Since LTL is a prominent logic in the area of safety-critical and provably reliable dynamic systems, LTL synthesis is a very tempting option to construct such systems since it avoids error-prone manual implementation; instead it is replaced with the need for a complete specification of the system (which is not trivial either, but in some cases easier). However, there is also an important computational caveat: the problem of LTL synthesis is 2-EXPTIME complete. Despite the infeasibility in the worst-case, many heuristics have been designed that can cope with practical problems, as documented by the yearly progress in the synthesis competition

This research was funded in part by the German Research Foundation (DFG) project 427755713 *Group-By Objectives in Probabilistic Verification (GOPro).*

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 390–414, 2023.
https://doi.org/10.1007/978-3-031-37706-8_20

SYNTCOMP [18], which has an LTL track for a number of years. Yet, many reasonable instances even in the benchmark set of SYNTCOMP still remain practically unsolvable. In this paper, we aim at *guessing a solution* through a machine-learning model, even for hard cases, thus possibly providing an applicable answer, in a sense, without reading the input formula. We achieve that by learning from other games and by reflecting *semantic* information, bringing the process closer to human reasoning.

The classic technique for solving LTL synthesis is to

1. turn the LTL formula into a deterministic parity automaton (DPA),
2. turn the DPA (and the partitioning of atomic propositions into system variables and environment variables) into a parity game (PG) between the system and the environment players, and
3. solve the PG; any winning strategy of the system player then directly induces a system policy (also representable as a circuit) satisfying the LTL formula.

Due to the worst-case doubly-exponential blowup in the first step and the practically bad performance of (Safra's [39] and others' [36,40]) determinization procedures, this option was rarely used practically until direct, more practical translations were given [8,12]. The significantly smaller automata [20] have made this approach feasible and, in fact, winning in SYNTCOMP since then. The approach is implemented in the tool `Strix` [33], which additionally constructs the DPA/PG *only partially*, on-the-fly until it finds a winning strategy for one of the players. This helps to overcome some more cases where the DPA is still very large; yet, more complex specifications often remain out of reach.

Semantic Labelling. The key difficulty in the on-the-fly exploration is a good heuristic that prioritizes exploration in promising directions, so that a solution can be obtained quickly, without constructing "irrelevant" parts of the game.

In a concrete state of a PG, is it better to go left or right? While this question obviously does not have a simple answer in general, we take a step back and instead of a PG we solve the LTL synthesis problem. For instance, consider a state of a PG corresponding to satisfying $\mathbf{G}a$, i.e. "always a holds". Then, the letter $\{a\}$ is clearly a better choice (for the system) than $\emptyset$. The former leads to the obligation of satisfying again $\mathbf{G}a$; the latter to the obligation `ff` (falsifying the formula). Taking the former edge does not guarantee winning, but the chances are certainly higher than giving up directly. In order to estimate the chances of winning with some obligation, we can evaluate it by randomly assigning truth values to temporal subformulae; intuitively, $\mathbf{G}a$ can be true or false, so its "trueness" is 0.5, `ff` has trueness 0. *Trueness* is examined in [22] and utilized in newer versions of `Strix` [31] as guidance.

Does every state correspond to a goal in LTL? And if so, can we determine which continuation brings us closer to satisfying it? Recall that the classic translations of LTL to non-deterministic Büchi automata (NBA), stemming from [43], label the states of the NBA with a conjunction of LTL formulae, which are the current goals in this state. For deterministic automata, the situation is inevitably more complex. While the determinization procedures obfuscated any possible such semantic labelling, the more recent approach re-established it, e.g., [8] with

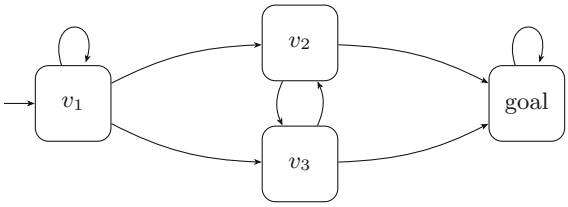

Fig. 1. Simple game where it is not clear which edges are "winning".

[26], or [42] with [9]. Beside the overall goal, it is necessary to also monitor the *progress of subgoals*. For example, consider $\mathbf{GF}(a \wedge \mathbf{X}b)$ "infinitely often a is followed by b". No matter what happens, the goal remains the same. However, whenever a, we are progressing with the subgoal of seeing the $a - b$ sequence once, yielding a subgoal b, which is regarded as promising.

Our Aim. In this paper, we aim at *better guessing of winning decisions* than in [22,31]. While the previous work only reflected trueness of the main goal, which is just the percentage of truth assignments leading to satisfaction of a Boolean formula, our approach reflects also (i) the temporal structure of the formulae, (ii) the monitored subgoals, and (iii) learns from previously solved games. On the technical level, we design over 200 *structural features* instead of just trueness, learn an *SVM* classifier comparing which edge is most promising, and use *data from previously solved games*, i.e. which edges are "winning". As it turns out, defining this notion already is surprisingly tricky: We cannot simply use the output of classical strategy improvement algorithms, as there may be multiple, incompatible solutions. Indeed, already for reachability, there are no maximal permissive strategies [3], see Fig. 1. Here the edge (v_2, v_3) is winning iff (v_3, v_2) is not used, and vice versa; using both makes them losing. Nevertheless, they are "better" than, e.g., the self-loop on v_1, which is always losing. Thus, we want to value both edges between v_2 and v_3 equally, and higher than the self loop on v_1.

Our Contribution can be summarized as follows:

- We learn a model predicting which edge has better chances to be winning. To this end, we define features on the semantic labelling in Sect. 5.1, introduce a way to measure the degree of "winning" of an edge in Sect. 4, and apply learning of support vector machines using our novel ground truth in Sect. 5.2.
- We evaluate "how winning" the suggested strategy is, i.e. how many wrong choices it made, on several inputs in Sect. 6.2. Surprisingly, this value often is 0, i.e. our strategy is often winning even for complex formulae, and even without reading them (meaning that our strategy is of constant size, *independent of the formula*, as opposed to a decision table in the concrete game; it can be run on the fly with no pre-computation, and decisions depend only on the labelling of the current state).
- Besides, while `Strix`'s architecture and interface ask for a significantly different type of advice (not just for the better of two edges), we show

`Strix` already profits from our advice and—modulo our unoptimized advice implementation—speeds up significantly, as we see in Sect. 6.3.

Usage of our Results:

- We provide an immediate solution (without even reading the input formula), which is often winning; moreover, it is applicable even to games too huge to be analyzed in any way. Besides, it is even of a constant size, i.e. independent of the size of the state space.
- Our approach opens the way to (i) a solver based on the semantic labelling, for instance, based on strategy iteration only quickly fine-tuning the already almost correct guess, and (ii) an on-the-fly-exploration advisor to `Strix`, with the proven potential to be the most efficient among the current techniques.

Related Work. To the best of our knowledge, there is only one other approach to using machine learning in LTL-synthesis. Here, the authors train a very powerful model (a hierarchical transformer) in order to directly predict a controller or counter example solely off the LTL specification [41]. Further, if their prediction is refuted by a classical model checking algorithm, they train a separated hierarchical transformer to repair it [5] until it is correct. While this turns out to be an overall competitive approach that also manages to solve some instances where classical synthesis tools as `Strix` [33] fail, this does not yield a complete procedure, as the repair loop is not guaranteed to ever terminate. In this work, we aim to improve existing, complete procedures such as implemented in `Strix` by means of machine learning based heuristics.

2 Preliminaries

We introduce notation and provide an overview of necessary background knowledge. Due to space constraints, we only briefly comment on several topics and refer the interested reader to the respective literature.

We use $\mathbb{N}$ to denote the set of non-negative integers. The constants `tt` and `ff` denote *true* and *false*, respectively.

2.1 Synthesis & Games

The synthesis problem in its general form asks whether a system can be controlled such that it satisfies a given specification under any (possible) environment. Moreover, one often is interested in obtaining a witness to this query, i.e. some *controller* or *strategy* which specifies the system's actions.

Parity Games are a standard formalism used in synthesis. A *parity game* is a tuple $\mathcal{G} = ((V, E), v_0, P, \mathfrak{p})$, where (V, E) is a finite digraph, $v_0 \in V$ a *starting vertex*, $P : V \to \{\mathcal{S}, \mathcal{E}\}$ a *player mapping*, and $\mathfrak{p} : V \to \mathbb{N}$ a *priority assignment*. Each vertex belongs to one of the two players $\mathcal{S}$ (called *system*) and $\mathcal{E}$ (called *environment*). In other words, the set of vertices is partitioned into player $\mathcal{S}$'s vertices $V_{\mathcal{S}}$ and player $\mathcal{E}$'s vertices $V_{\mathcal{E}}$. See Fig. 2 for an example.

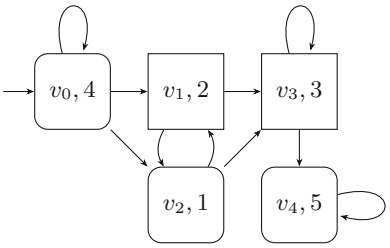

Fig. 2. An example parity game, taken from [22]. Rounded rectangles belong to the system $\mathcal{S}$ and normal rectangles to the environment $\mathcal{E}$. The vertices are additionally labelled with their priorities.

Remark 1. In our implementation priorities are assigned to edges instead of vertices, as this allows for a much more concise representation and suits most translations better. However, for ease of presentation, we consider *state-based acceptance* instead of *transition-based*.

Playing. To play the game, a token is placed in the initial vertex v_0. Then, the player owning the token's current vertex moves the token along an outgoing edge of the current vertex. This is repeated infinitely, giving rise to an infinite sequence of vertices containing the token $\rho = v_0 v_1 v_2 \cdots \in V^\omega$, called a *play*. We write ρ_i to refer to the i-th vertex in a play. A play ρ is *winning* (for the system player) if the smallest priority occurring infinitely often is odd. (Using "maximal" instead of "minimal" or "even" instead of "odd" does not fundamentally change the problem at hand.) Formally, we define $\inf(\rho) = \{v \in V \mid \forall j.\ \exists k \geq j.\ \rho_j = v\}$ as the set of infinitely occurring states. Since the game graph is finite, this set always is non-empty. The smallest priority occurring infinitely often is given as $\mathfrak{p}(\rho) = \min\{\mathfrak{p}(v) \mid v \in \inf(\rho)\}$ and system wins the play ρ if $\mathfrak{p}(\rho)$ is odd.

Strategies. A strategy of player p is a mapping $\sigma_p : V_p \to E$ assigning to each of p's vertices an appropriate edge along which the token will be moved, i.e. $(v, \sigma_p(v)) \in E$ for all $v \in V_p$.[1] Once both players fix a strategy, the game is fully determined and a unique run is induced. We call a strategy of system $\sigma_\mathcal{S}$ *winning* if for *all* strategies of the environment $\sigma_\mathcal{E}$ the induced play is winning, i.e. system wins no matter what the environment does.

For example, consider again the game depicted in Fig. 2. Fixing the strategies $\sigma_\mathcal{S} = \{v_0 \mapsto (v_0, v_2), v_2 \mapsto (v_2, v_3), v_4 \mapsto (v_4, v_4)\}$ and $\sigma_\mathcal{E} = \{v_1 \mapsto (v_1, v_2), v_3 \mapsto (v_3, v_3)\}$ induces the play $v_0 v_2 v_3 v_3 \cdots$. The set of infinitely often seen priorities equals $\{3\}$, hence the system player wins with these strategies. Moreover, the strategy σ_0 is winning, since the play always ends up in either v_3 or v_4.

Synthesis. With these notions, we can compactly define the synthesis question: *Given a parity game $\mathcal{G}$, does there exist a winning strategy for the system player?* In the example above, σ_0 is a witness to this question.

[1] Strategies may be more complex, e.g., by using memory. However, "positional" strategies are sufficient for parity games, thus we omit the general definition.

This problem is still intensely studied due to its broad applications. It also is one of the few problems which canonically lie in **NP** ∩ **coNP** (even in **UP** ∩ **coUP** [19]), with recent breakthroughs achieving quasi-polynomial algorithms [4,14,28].

Extensive-Form Game. A common notion in game theory is the *extensive-form* game. Intuitively, this means completely "unrolling" the game into an explicit representation. See e.g. [34, Chp. 5–7] for details. In our case, we consider the *game tree*, where each node corresponds to a simple path in the game $\mathcal{G}$. Suppose we are in state $s = (v_1, \ldots, v_i)$ of the game tree. Then, the successors of s are determined by all successors of v_i in the game, i.e. $\{u \mid (v_i, u) \in E\}$ as follows. Suppose such a successor u already occurs along s, i.e. a loop is closed, we check if the corresponding play is winning or losing. In that case, the choice leads to a corresponding winning or losing leaf of the tree, respectively. Otherwise, i.e. when no loop is closed by the choice, it leads to $s \circ u$. Essentially, this game tree represents all potential simple paths (and thus, intuitively, all potential positional strategies) that can arise in the game, and each edge corresponds to a particular move of a player (also called *ply* in game theory). In particular, it is finite, however of potentially exponential size. Note that we can restrict to simple paths only because positional strategies are sufficient.

Minimax Game Solving. A fundamental way to solve games is the *minimax decision* rule, which intuitively corresponds to exhaustively exploring the extensive-form game (also discussed in [34]). Suppose we assign a value of 0 to "losing" leaves of the game tree and a value of 1 to the "winning" leaves. Then, we can "back-propagate" values by setting $V(s)$ the maximum of all successors of s if it currently is the turn of the system player and the minimum if instead it is environment's turn (which wants the system to lose). The game is winning if the value in the initial state of the game tree is 1. This approach is also called *backward induction* or *retrograde analysis*: starting from the winning / losing positions of the game, we consider all moves which could lead to such situations.

Strategy Improvement (or *strategy iteration*, abbreviated by *SI*) is the most prominent practical way of solving parity games, i.e. answering the synthesis question. It received significant attention due to recent practical advances [13,15,17,32] and modern tool developments [6,33]. We explain the approach briefly, since its details are not important for this work. Intuitively, SI starts from arbitrary initial strategies for each player, and then performs the following steps in a loop. First, we check whether either strategy is winning. If yes, the algorithm exits, returning this strategy. Otherwise, one of the strategies is improved by changing its choices in some vertices. If an improvement is not possible, there exists no winning strategy for the respective player. Otherwise, the process is repeated with the new strategy.

This algorithm converges to the correct result in finite time for any initial strategy. However, if this initial strategy is chosen "close" to a winning strategy, then SI intuitively needs to perform fewer steps to converge to an optimal

one. Thus, a heuristic which often comes up with a "good" initial strategy may improve the runtime significantly over arbitrary or random initialization.

2.2 Linear Temporal Logic and Reactive Synthesis

Linear Temporal Logic (LTL) [37] is a standard logic used to specify desired behaviour of a system. The syntax usually is given by

$$\phi ::= \mathtt{ff} \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi \,\mathbf{U}\, \phi,$$

where $a \in \mathsf{AP}$ is an *atomic proposition*, inducing the *alphabet* $\Sigma = 2^{\mathsf{AP}}$. These formulae are interpreted over infinite sequences $w \in \Sigma^\omega$ called ω-words. A word $w = w_0 w_1 \cdots \in \Sigma^\omega$ satisfies the *next* operator $\mathbf{X}\phi$ iff ϕ is satisfied in the next step. Similarly, the *until* operator $\phi \mathbf{U} \psi$ is satisfied iff ϕ holds until ψ is eventually satisfied. Usual abbreviations are defined as *finally* $\mathbf{F}\phi \equiv \mathtt{tt}\,\mathbf{U}\,\phi$ and *globally* $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$, which require that ϕ holds at least once or always, respectively. Moreover, the construction underlying our work also considers *strong release* $\phi\,\mathbf{M}\,\psi \equiv \psi\,\mathbf{U}\,(\psi \wedge \phi)$, *(weak) release* $\phi\,\mathbf{R}\,\psi \equiv \mathbf{G}\psi \vee (\phi\,\mathbf{M}\,\psi)$, and *weak until* $\phi\,\mathbf{W}\,\psi \equiv \mathbf{G}\phi \vee (\phi\,\mathbf{U}\,\psi)$. Considering these additional operators allows formulas to be represented in *negation normal form*, i.e. the negation $\neg$ only appears in front of atomic propositions. In the interest of space, we refer to [12] for precise definition on the semantics and discussion of these subtleties. Understanding these issues is however not required for this work.

LTL Synthesis is an instance of the general synthesis problem, where the specification to be satisfied is given in form of an LTL formula [38]. Due to recent advances [11,12,16,20,21,25], the *automata-based approach* [43] to LTL synthesis received significant attention. In particular, the tool `Strix` [33], built on top of `Owl` [24], which in turn implements these ideas, won several iterations of the synthesis competition SYNTCOMP [18]. Essentially, the given LTL formula is translated into an ω-automaton, which in turn is transformed into a parity game. Solving the resulting game yields a solution to the original synthesis question.

This game is obtained by "splitting" the automaton, as follows. The set of atomic propositions is split into system- and environment-controlled propositions, i.e. $\mathsf{AP} = \mathsf{AP}_{\mathcal{S}} \cup \mathsf{AP}_{\mathcal{E}}$, and the players' actions correspond to choosing which of their propositions to enable. Once both players chose their propositions' values, the automaton moves to the next vertex according to the players' choices. Concretely, for an automaton state p, the environment can choose to move into (p, v) where $v \subseteq 2^{\mathsf{AP}_{\mathcal{E}}}$, and from there, system can move to any automaton state $q = \delta(p, v' \cup v)$ where $v' \subseteq 2^{\mathsf{AP}_{\mathcal{S}}}$ and δ is the transition function of the automaton. In particular, this means that the obtained game is *alternating*, i.e. system and environment take turns in alternation. Moreover, by convention the environment moves first. See e.g. [33] for more details on this approach.

Semantic Translations from LTL to automata are the key ingredient to our approach. On top of providing a parity game, they also give a *semantic labelling*,

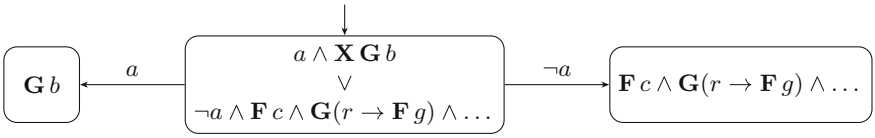

Fig. 3. Motivational example to provide guidance through semantic labelling.

i.e. interpretable meaning, to the game's vertices. In particular, the approach introduced in [8] (see also [10–12]) and implemented in `Owl` [25] intuitively yields for each vertex a list of LTL formulae, which roughly correspond to (sub-)goals which still have to be fulfilled, possibly repetitively.

2.3 Our Goal

In this work, we want to demonstrate that this semantic labelling can be efficiently exploited for reactive synthesis. For a motivational example to consider semantic labelling, we display a (vastly simplified) labelled game in Fig. 3. We are offered with the choice of choosing a or $\neg a$. While it is not completely clear that choosing a is indeed better, it certainly seems to be more promising, as the subsequent labelling seems much "easier" to handle. Thus, faced with a choice, we likely would first try to win with a. Observe that without the semantic labelling, our best option in this situation would be a random guess. In [22], the authors used a simple, manually designed mechanism trying to capture this notion, called *trueness*. Motivated by the (surprisingly good) results of this approach, we want to tackle this problem by more sophisticated means. Concretely, we want to make meaningful decisions based on the labelling. However, while the theory underpinning semantic translations is quite clean and pleasant [12], the actual labellings appearing in practice are quite complex. To further complicate things, the highly optimized implementation thereof [25] employs several subtle optimizations and special cases. We provide an example to showcase the complexity of this labelling in practice later in Sect. 5, kept brief in the interest of space, and a small real-world example in [23, Appendix A.1]. Since we have a simple intuition which however seems difficult to formalize, we opt to tackle this problem through means of machine learning.

3 Previous Approaches and Their Limitations

In this section, we briefly summarize the ideas of [22] and the inherent problems associated with them. The primary motivation of [22] is to exploit the semantic labelling provided by [25], which gives us an indication of the long term goals in the game. As an analogy, consider the game of chess. Here, the "semantic labelling" is given by the board state, i.e. the position of each piece. This labelling provides us with a reasonable indication of (i) our current situation and (ii) which moves might be better than others. In particular, understanding

and evaluating the semantics of the game is what allows humans to have a good intuition about the quality of moves, without thinking through the intractably large game tree. Likewise, this understanding is what enabled algorithms to perform beyond human capabilities.

3.1 Parity Game Solving by Trueness

A central notion of [22] is *trueness*, an approximation of how close a formula is to being satisfied, i.e. `tt`. The intuition is that the semantic labelling of states effectively describes "goals" of the system player. If the formula is `tt`, the system has satisfied all goals and consequently won the game. Likewise, increasing the trueness is indicative for a good move. Remaining with the analogy of chess, trueness somewhat corresponds to counting the number of pieces on the board (or rather the difference between our and the opponent's pieces): If no enemy pieces remain, we certainly have won, and a change of this difference, i.e. capturing an enemy piece or avoiding capture of own pieces, is a good indicator for the quality of a move. In particular, this prohibits us from taking moves which immediately lead to a piece being taken.

In [22], the authors propose two ideas. First, they suggest to use a trueness-maximizing strategy as initial one for strategy iteration, i.e. in each state select the edge which maximizes (or minimizes, in the case of $\mathcal{E}$) the obtained trueness. Second, they use *Q-Learning*, a popular reinforcement learning approach, as a solver for parity games, i.e. as competitor to strategy iteration, using three different reward signals. There, each edge is given a reward, which is mostly based on (the change of) trueness, and these values then are back-propagated until choosing optimal rewards in each step yields a winning strategy.

While they also show Q-Learning to be an interesting avenue, we primarily focus on the "initializing strategy iteration" approach, since our goal is to augment exiting strategy iteration solvers. Moreover, the experimental evaluation of [22] suggests that Q-Learning scales poorly to large real-world formulae.

3.2 Problems

We now outline two key issues of this approach.

Myopic Trueness The primary heuristic in [22] is trueness. While this approach already performs surprisingly well, especially for so called *safety* and *co-safety* formulae, it fails to take into account temporal dependencies; trueness is myopic. Again, considering chess, while counting the change of pieces does help us avoid "obviously stupid" moves, it does not stop us from moving pieces into positions where they are effectively guaranteed to be taken eventually and does not allow for sacrificing a piece in exchange for a long-term advantage.

Manual Design Their reward functions were defined manually, in contrast to being obtained from a learning process. While the intuition behind these definitions is reasonable, obtaining a guidance heuristic as a result of an optimization process is a much more principled approach.

We proceed to outline how we tackle these issues by a more sophisticated approach.

4 A New Hope

We want to improve reactive synthesis by applying machine learning. As already motivated by [22], we want to approach this problem by identifying "promising" edges, choosing those as initial strategy for SI. Naturally, as a first step, we need training data for our learning approach. In particular, we need to identify which actually are the actual good choices in games, i.e. the *ground truth*. As it turns out, this is more complicated than one might expect.

4.1 Obtaining Training Data with SI

As SI allows us to solve a game and determine winning edges, one might try to employ SI for obtaining a ground truth (as we did initially). However, SI actually provides us with potentially misleading or even conflicting information! As we already hinted in the introduction through Fig. 1, SI cannot give us a canonical ground truth. In the example, one edge is winning iff the other is not used, and vice versa. Thus, SI will yield a strategy which does not take both edges and we would consider one of them losing. Moreover, note that there is no fundamental reason to prefer one edge over the other, so SI might in one run classify the edge from v_2 to v_3 as good and in a second run (or on a similar game) do the opposite or even consider neither winning. The underlying problem is that parity games do not allow for a unique *maximally permissive* strategy (see e.g. [3]), thus we cannot derive the "suitability" of an edge from a single solution strategy.

4.2 Solving the Game Tree

Instead of using a particular strategy obtained from SI, we therefore propose to identify "all" solutions, i.e. all edges which are part of a winning strategy. More formally, for each vertex v we want to determine the value of each outgoing edge in the corresponding game tree rooted at v. To prefer "shorter" solutions over larger, we add a beta-decay to the value. Concretely, suppose we consider the game tree state $s = (v_1, \ldots, v_i)$ which ends in a system state v_i. Then, the value of s is defined by $\mathrm{val}(s) = \beta \cdot \max_{s' \in \mathrm{successors}(s)} \mathrm{val}(s')$ for a fixed $0 < \beta < 1$.

As we already mentioned, evaluating this tree is intractably large, namely exponential in the size of the game, which itself is already doubly-exponential in the input formula [27,38]. Thus, we employ a classical technique of game theory.

4.3 Monte Carlo Tree Search (MCTS)

Intuitively, we explicitly unfold the tree up to a specified depth, e.g. 7 plies, and then assign the results of (guided) random sampling to the occurring leaves, approximating the (beta-decayed) value of the game in these vertices.

We describe our method to approximate the value of a node $s = (v_1, \ldots, v_i)$ in the game tree. In essence, starting from v_i, we randomly select successors, with the following restrictions for each player. The environment plays *optimally*, i.e. if a state is winning for the environment (which we can determine beforehand through classical approaches) we immediately stop sampling and return a value of 0. Otherwise, the environment heuristically tries to delay the play as long as possible (decreasing the value the system player obtains due to beta-decay). In contrast, the system player checks in a one-step lookahead if a choice is trivially winning, i.e. leading to a state labelled `tt`, always choosing such an edge if one exists. Otherwise, the system randomly chooses among edges which are not trivially losing, i.e. lead to a `ff` state. If either player closes a loop, i.e. selects a successor which already occurs along the path, we determine the value by checking if the loop is winning or losing. A loss yields a value of 0, while a win yields β^{length}. In summary, we approximate the probability of winning by playing randomly (avoiding obvious mistakes) against an optimal opponent, under-approximating the true value. We deliberately opt for this random-choice approach to prefer regions where there is less potential for error.

4.4 Optimizations

While MCTS makes approximation of the game tree value feasible, we added several further technical improvements to arrive at a practically viable method.

SCC Decomposition. We exploit the structure of the game by decomposing it into its strongly connected components (SCCs) and put them in reverse topological order. Computing (or approximating) the value in that order allows for caching: Once a run in the game tree leaves an SCC, it can only reach SCCs further down in the topological order, and, since we compute values in this order, the value of the reached state is already known, allowing us to re-use it immediately.

Pruning. In addition to employing the MCTS values as game values in the tree expansion, we also use it to prune the game tree. In particular, once we computed the Monte Carlo values for each state, we restrict the choice of the environment to the successors which yield (close to) the lowest Monte Carlo value (recall that the environment prefers lower values). We empirically chose 0.02 as a threshold, i.e. we only keep those edges for the environment which are within 0.02 value of the lowest decision. While in theory this might remove crucial paths due to statistical fluctuations of MCTS, in practice it allows for a much deeper game tree, which in our experiments heavily outweighed the theoretical downside.

5 Handling the Truth

We introduced a way how to obtain a well-founded notion of "value" (to be precise, an approximation thereof) for a choice, i.e. an indication how good this choice is. As such, we can rank edges by their value in each state. Intuitively,

picking an edge which is ranked very highly should correspond to a good chance of winning. A high value means that even against an optimal player we can very likely close a winning loop, and, due to beta decay, do so quickly, thus minimizing the chance for an error.

Recall that our goal is to provide a good initial strategy. Thus, the exact values actually are irrelevant, since we only want to give the best edge as initial choice. Instead of trying to predict the exact value, we therefore want to learn this relative ranking. Formally, suppose we consider a system vertex $v \in V_S$ with edges $E_v = \{(v, u) \mid (v, u) \in E\}$. A ranking of edges effectively corresponds to a (total) order $\prec_v \subseteq E_v \times E_v$. The principle of *pairwise ranking* [30] suggests that we learn a function $f : E_v \times E_v \to \{-1, 1\}$ that classifies pairs of edges depending on which one is the better choice, i.e. $f(e, e') = 1$ if $e' \prec_v e$ and -1 otherwise. However, such a function might not be perfect. For example, we could get $f(e_1, e_2) = 1$, $f(e_2, e_3) = 1$, and $f(e_3, e_1) = 1$, which is incompatible with any order. Thus, learning to rank suggests to determine an ordering $\prec$ that minimizes the *inversions* w.r.t. f, i.e. the number of cases where $f(e, e') = 1$ but $e \prec_v e'$. This problem, called rank aggregation, is known to be **NP**-hard, and we employ a greedy approximation as suggested by [30].

Our concrete goal thus now is to learn such a function f based on the semantic labelling of the start and end vertices of the two edges. We want to employ machine learning for this purpose: While the high-level intuition of the semantic labelling is rather clear, the actual implementation used to obtain the games [24] employs numerous optimizations, separate cases, etc. To provide the reader with a sense of the complexity, we display a single edge in the automaton obtained for a simple formula in Fig. 4, and a real-world scenario in [23, Appendix A.1].

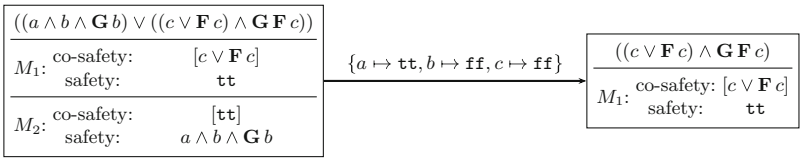

Fig. 4. A single transition in the automaton computed for the formula $(a \wedge \mathbf{G}b) \vee \mathbf{GF}c$.

We proceed to describe (i) (some of) the features we use, i.e. which quantities we extract from the labelling, (ii) the model we employ, and (iii) the dataset and methodology used to train our model.

5.1 Features

In total, we have defined over 200 different features to convert the edges into a usable vector of reals. In the interest of space we only present the high-level ideas of a small subset which covers most interesting ideas.

Since most information is contained in the states rather than in the edges themselves, the majority of our features are defined for the former. An edge is

then either associated with the feature value of its successor or with the change in a feature value between its predecessor and successor. As indicated in Fig. 4, the semantic labelling comprises several formulae, namely a "master" formula, which intuitively indicates the global state, and several "monitors" (which themselves comprise several formulae), monitoring repeating sub-goals. We define *base features*, which convert a single formula to a single number. These features can then be applied to both the master as well as monitor formulae, where further aggregation is necessary. Some notable base features are the following:

Number of Conjuncts We count the number of conjuncts if the top level operator is a conjunction and otherwise default to 1. The intuition behind this feature is that less conjuncts tend to correspond to a less constrained formula. Further, reducing the number of conjuncts along an edge often means that sub-goals have been achieved. (We consider several further syntactic features such as the number of disjuncts, the height of the syntax tree, or the number of temporal operators, which all follow similar ideas.)

Trueness Since this has proven to be a solid heuristic on its own, we again incorporate it as a feature.

System Control This feature (and variations thereof) incorporate the information of the variable partitioning by approximating how much impact the choice of the system variables has on the truth value of the formula. Intuitively, a higher system control is desirable. Further, this feature also counteracts false positives of, e.g., trueness, as high values of trueness are worth much less if the system has no control on whether one of the many satisfying assignments is played.

Obligation Set This group of features is based on the idea of *obligation sets* as introduced by [29]. In essence, an obligation set for a formula φ is an assignment that, if played indefinitely, satisfies the formula. Using the inductive definition of [29], we can compute a formula φ' whose satisfying assignments are exactly the obligation sets of φ, see [23, Appendix A.2]. Using this new formula, we can obtain numerous new features by applying other base features to φ'. In particular, we are interested in the new formulas trueness as this indicates how many obligation sets exist. Further, we are interested in its system control, as a higher value makes it more likely that the system can enforce at least one obligation set.

In addition to the base features, we define the following edge-specific features:

Priority As priorities are crucial for winning a play, it is only natural to incorporate that information in our features. However, as SVMs struggle with parity information, we reorder the priorities by how beneficial they are for the system and map them to $[-1, 1]$ (similar to [22]). In particular, the smallest odd priority gets mapped to 1 and the smallest even priority to -1. For this normalization, we use an a-priori upper bound provided by the underlying automaton construction.

Progress This feature is rather similar to [22]'s progress feature. We compute the percentage of already succeeded sub-goals of a monitor (instead of

their trueness) and aggregate by weighted average (rather than maximum). Additionally, we introduce punishments for failing monitors. Intuitively, this encourages long-term progress for temporal goals.

One Step Here, the idea is to recommend an assignment that is to be played in the current state by traversing the syntax tree and propagating recommendations upwards, which is inspired by message passing in graph neural networks. For example, if we see $a \wedge b$ we strongly recommend playing a and b, if we see $\mathbf{F}(a \wedge b)$ we take the previous recommendation and tune it down, since $\mathbf{F}$ is "less urgent". The feature value is obtained by measuring how well the valuation of an edge aligns with the recommended assignment.

5.2 Pair Classification by Support Vector Machines

To instantiate our pair classification function f, we opt for support vector machines. In principle, one could employ any binary classifier, which is why we also experimented with other models such as decision trees, random forests or gradient boosted trees. However, SVMs proved to perform best, which we attribute to their great ability to generalize due to their margin maximizing nature [30]. Additionally, SVMs are rather simple (compared to our other options) and provide us with extra information known as *confidence*. Given by the distance of the predicted sample to the decision hyperplane, its magnitude can be interpreted as how confident the SVM is in its prediction. We denote the confidence of a pair (e_1, e_2) by $c(e_1, e_2)$ and use it to slightly alter the greedy ranking algorithm from literature. To rank the edges of a vertex v, each edge $e \in E_v$ gets assigned a score $s(e) = \sum_{e' \in E_v, e' \neq e} c(e, e')$. Recall that if we predict $e \prec_v e'$, the confidence is negative. Finally, we rank the edges according to their score, where a higher score corresponds to a better edge, and the recommended strategy is obtained by playing the highest ranked edge for each state.

5.3 Further Notes on Implementation

In addition to the feature extraction, there are several other engineering aspects, which are crucial for the final performance. In this section, we comment on the three most important ones.

Statewise Feature Normalization. Before passing the features to the model, we proceed to normalize them. Due to possible future applications in on-the-fly solvers, we only consider feature values of edges from the same state for this normalization. The crucial observation is that this already introduces comparative information in the features. A normalized trueness value of 1, for example, means this edge has the best trueness among all other edges from their state although it does not tell us anything about its absolute value. While the latter might also be important in theory, we observed that in practice the statewise normalized value is more important with only a few exceptions.

State Classification. We observed several significantly different behaviours required in different states. For example, in some states we need to exclusively focus on the master formula, while in others only the monitors play a role. This also relates to the underlying principles of the automaton construction. It is very difficult, especially for a simple model like an SVM, to switch between different behaviours. We divide states into three groups which approximate the different classes, and train separate models for each class. The three classes we suggest are (i) states without monitors, (ii) states where the master formula does not change in any successor, (iii) and states that fall into neither category. In addition to having the separate models learn separate behaviours, we can also provide them with separate feature sets that only include relevant information. For example, the first class only requires features of the master formula, whereas these can be neglected in the second one.

Complement Construction. The underlying automaton construction uses the fact that the system being able to enforce satisfaction of a formula φ is equivalent to the environment being able to enforce falsification of $\neg\varphi$. In other words, solving the game for the negated formula and swapped roles yields the same result. However, in the game obtained for $\neg\varphi$ the role of "system", the player who choses second and for which we learnt the recommendation, i.e. for transitions from states (p, v) to q, now corresponds to the original environment. This drastically changes the meaning of features. For example, a trueness of 0 suddenly is very desirable. We tackle this by training separate models for both cases. Together with state classification, this yields a total of 6 different models that we assemble for our heuristic.

5.4 Training the Model

With these ideas at hand, we conclude this section by discussing our dataset, in particular how we preprocess it, and how we train our model.

Dataset and Preprocessing. As one of our goals is to exploit human bias in writing LTL formulae, the foundation of our dataset is given by the LTL benchmarks of SYNTCOMP.[2]. To further augment the data, we mutate these formulae by randomly replacing temporal operators. This yields new (random) samples that syntactically resemble the original, human-written structure. For practical reasons, we only consider formulae which can be converted to a DPA within 10 min. Ultimately, this leaves us with 405 original and 514 mutated formulae, of which we use 60% each for training, 20% for validation, and 20% for evaluation.

Obtaining the edge pairs for training requires several further steps. First of all, we exclude trivial cases that can easily be detected by simple rules (see Sect. 4.3), allowing our model to focus on complicated cases. Further, we exclude pairs where the ground truth value happens to be equal, as it is unclear which edge the model should predict. In particular, we exclude all edges originating in losing

[2] Available on GitHub https://github.com/SYNTCOMP/benchmarks.

states (since there is no sensible action to recommend). Finally, we only include a limited amount of pairs per game in the training set: Pairs of the same game tend to look similar, thus a few disproportionately large games would result in a very unbalanced dataset. All remaining edge pairs are added in both orders, i.e. $((e_1, e_2), y)$ and $((e_2, e_1), -y)$, where $y \in \{1, -1\}$ determines which edge is better, in order to prioritize teaching symmetry to the model.

Training. For each of the 6 models, we first compute mean and standard deviation of the respective training set and use them to standardize the input to $\mathcal{N}(0, 1)$. Further, we perform recursive feature elimination for each state class individually, adapted to features appearing twice (once for each input edge). For each state class, we ended up with 30–40 features.

For the actual training process, we performed an extensive grid search for several model types (decision trees, random forests, etc., see Sect. 5.2) in order to determine suitable values for the hyper-parameters. As mentioned earlier, we ultimately opted for the SVMs due to their simplicity and generalization abilities.

6 Experimental Evaluation

In this section, we present experimental evaluation of our tool `SemML`. The model was learnt by communicating the relevant data to a Python process running `scikit-learn` [35]. We then extracted the learnt weights and, based on them, implemented the recommendation procedure in Java, on top of `Owl` [24]. The artifact can be found at [1], which references a slightly improved version from the one we submitted to the artifact evaluation [2].

6.1 Evaluation Goals

Our primary goal in this work is to show that our approach, enabled by our new ground truth, can be used to solve more complicated instances than the approach of [22], in particular formulae going beyond pure (co-)safety. Thus, our first evaluation goal is the following:

> *Research Question 1:* How much does our model based on SVM and the game tree ground truth outperform the trueness-based initial strategy recommendation approach of [22]?

We refer to the trueness-based initial strategy of [22] as `TrueSI`.

Although not the focus of this work, we ultimately want to improve synthesis through meaningful exploration guidance, in particular, by suggesting likely winning edges. Thus, we are interested how our prototype performs in a real-world scenario.

> *Research Question 2:* How do initial strategies recommended by our approach synergize with state-of-the-art synthesis tools?

We address both questions separately.

6.2 RQ1: Quality of Initial Strategy

Datasets. To fairly compare to [22], we consider the same dataset, i.e. randomly generated LTL formulae, split into three categories: "(Co-)Safety", "Near (Co-)Safety", and "Parity". See [22] for details on how these are obtained. In essence, the tool `randltl` [7] is used to generate random formulae with different biases. Then, we filter out formulae which need more than 10 min to be translated to a parity automaton. As a second dataset, we also use some (original and mutated) SYNTCOMP formulae (the test set described in Sect. 5.4). We only consider formulae where the corresponding game can be won by system. We do this simply because we can only recommend on games which are winning – otherwise there is no preference on edges since every action is losing by definition. In total, this leaves 262 randomly generated formulae and 123 from SYNTCOMP.

Metrics. We consider two metrics for our comparison. Firstly, similar to [22], we consider the fraction of *immediately solved* games, i.e. games where following actions recommended by `SemML` or `TrueSI` directly yields a winning strategy. In light of our motivation to augment SI solvers, we want to measure how "close" the recommended strategy is to being correct in case is not immediately winning. To this end, we feed it to (a modified version of) the parity game solver `Oink` [6] and compute the *(relative) distance* of the obtained strategy, as follows. We count the number of (reachable) states in which the winning strategy determined by `Oink` differs from the recommended one, i.e. how many "wrong" choices were recommended, and divide it by the total amount of (reachable) states. We note that this unfortunately induces a slight bias that we cannot measure: `Oink` may potentially change winning decisions because of internal details of the algorithm. Ideally, we would want to obtain the minimal distance over all winning strategies; however this quantity is intractable to compute due to the exponential size of the strategy space. Nevertheless, we believe that this measure strongly correlates with the quality of the strategy.

We argue that simply measuring the number of iterations required by strategy iteration to converge is a too crude metric: On the one hand, even a "very wrong" strategy can be changed to a winning strategy in a single iteration by changing the choice in every single state. On the other hand, even a nearly correct strategy, requiring only a hand full of changes, may need as many iterations. Moreover, this additionally induces the same bias as above.

Table 1. Summary of our comparison between `TrueSI`, the approach of [22], and our tool `SemML`. We first list the fraction of immediately winning strategies (larger is better), followed by the geometric mean of the relative distance, i.e. the fraction of states in which the decision was adapted by `Oink` to obtain a winning strategy (smaller is better). For the first comparison, we also consider random initialization as a baseline. For this second comparison to be fair, we only consider games where neither tool yielded an immediately winning strategy.

Tool	(Co-)Safety	Near (Co-)Safety	Parity	SYNTCOMP
Immediately Solving				
`TrueSI`	100%	85%	66%	44%
`SemML`	99%	95%	88%	85%
Random	7%	2%	5%	3%
Relative Distance				
`TrueSI`	–	75%	45%	29%
`SemML`	–	52%	28%	16%
Ratio of both	–	1.4	1.6	1.8

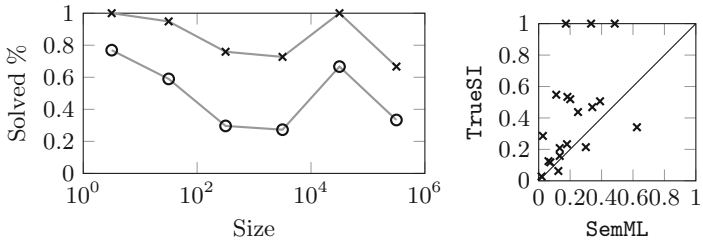

Fig. 5. A detailed comparison on SYNTCOMP formulae. The left plot compares how many games were immediately solved, grouped by size and considering the (arithmetic) mean in each group. `SemML`'s values are displayed by crosses, `TrueSI` by circles. The right plot compares the relative distance of `SemML`'s and `TrueSI`'s solutions.

Expectations. Since our approach incorporates trueness as one of its many features, we expect that our approach should be at least on par with the previous one of [22]. As we also consider long-term temporal information beyond trueness, we particularly expect to outperform `TrueSI` on larger, more complicated instances.

Results. We ran this evaluation on consumer hardware (Intel Core i7-8565U with 16GB RAM). We summarize our findings in Table 1. Clearly, our approach vastly outperforms the previous one. In particular, while `TrueSI` perfectly handles (co-)safety formulae, its performance quickly drops when going to more complicated formulae. In comparison, the `SemML` solves the vast majority of formulae immediately, even on the quite complicated SYNTCOMP dataset. We note that these findings are not "absolute" (as to be expected from machine

learning approaches). There are few instances where the previous approach does perform better. Our baseline comparison to a random initialization approach validates that both approaches indeed solve a non-trivial problem.

Since we are particularly interested in complex, "human written" formulae, we investigate the SYNTCOMP dataset more closely. In Fig. 5, we provide a more detailed view on our two metrics. First, we investigate how the "immediately solving" performance evolves in comparison to the size of the game, which intuitively correlates with the difficulty of the synthesis question. We observe that `SemML` solves practically all smaller games and still performs well on larger games, compared to `TrueSI`, which quickly falls off. The second plot displays the relative distances for each instance which neither recommendation solved immediately. We clearly see that the strategies recommended by `SemML` are better in almost all cases.

This positively answers our first question. Aside from the direct comparison to the previous approach, the significant percentage of immediately solved games gives us an interesting implication: If `SemML` solves many games immediately, we can use `SemML` as a best-effort guidance tool for reactive synthesis questions which are intractably large to solve. Moreover, `SemML` thus presents us with a constant size representation of a winning strategy for many games, effectively described by approximately a few hundred SVM weights compared to a decision table for thousands of states in *each* game.

6.3 RQ2: On-the-fly SemML

In our second experiment, we evaluate the suitability of `SemML` for real-world parity game solving by using it as guidance tool for the state-of-the-art reactive synthesis tool `Strix` [33].

Strix' Anatomy. We first briefly describe how `Strix` works and how it uses guidance heuristics. In essence, `Strix` builds the parity game on-the-fly, i.e. iteratively constructs parts of the game it deems important. Then, two strategy improvements are running in parallel, one for either player. Not yet explored states are treated as losing for both. In this way, if we find a winning strategy for either player on the constructed part of the game, it is winning for the complete game. Otherwise, we need to explore further. Here, a key ingredient for practical efficiency is a heuristic to decide which states should be explored first: If we explore states reachable under the "smallest" winning strategy, we naturally find this strategy as quickly as possible. In its current form, `Strix` employs trueness for this guidance and selects an *automaton* edge with the *globally* highest trueness for exploration. (Dually, edges with the lowest trueness are also followed, since these are "promising" for the environment.)

Integration. We integrate `SemML` with `Strix` as follows. Suppose we are asked to compute a global score for an automaton edge $e = (p, q)$ (recall that `SemML` gives *local* advice on edges in the *game*). We explicitly build up the game between the automaton states p and q, i.e. all choices of the environment in p followed by the

respective system choices. For each occurring system state s, we compute the SemML ranking score as explained in Sect. 5.2, i.e. the confidence based score. This only gives us local information: the magnitude of our score only reflects the preference relative to actions available in the system state $s = (p, v)$. Since the previously used trueness proved to be a good indicator for global progress, we multiply our local score by this global value. Finally, to obtain a value for the automaton edge, we take the minimal value of all arising system states, since the environment chooses first. We additionally apply straightforward rules such as assigning values of 0 and 1 values to ff and tt states, respectively. Finally, Strix by default employs a decomposition approach, which does not build a single DPA. Therefore, SemML would not be applicable, and we disable it for the purpose of evaluation.

Dataset. We considered 188 randomly selected formulae of SYNTCOMP (which were not used in the training of the model), also including unrealizable ones.

Metrics. We evaluate the total required time to solve the game and compare to Strix in its normal configuration. Since we expect the unoptimized computation of SemML's advice to take considerable time, we separately measure the required time and additionally perform a comparison with this time subtracted. Since our scoring function is a straightforward SVM, we strongly believe that by tailoring the evaluation to Strix' requirements, it can be significantly sped up. In particular, our advice computation re-constructs information which is computed during the exploration of the automaton but difficult to access without significant changes to both Strix and Owl.

Expectations. We do not expect this approach to work to its full potential because Strix architecture does not exactly fit our approach (recall that our primary motivation was to compare to [22]). We discuss these differences and possible ways to address them later. Moreover, as we construct the intermediate game states for every recommendation and evaluate the recommender SVM several times, we expect that significant time is spent computing the advice of SemML.

Results. We conducted our experiments on a server with an Intel Xeon E5-2630 v4 processor with 256GiB of RAM and employed a 10 min timeout per execution. We summarize our findings in Fig. 6. Strikingly, our approach already performs favourably, despite the differences in architecture, hardly optimized advice computation, and no specific re-training for the task at hand. Excluding the time spent for advice computation, our approach performs significantly better in practically all instances. This answers our second question positively, too.

Adapting SemML to Strix In order to adapt our underlying approach, we require several non-trivial changes to SemML. We discuss the "mismatches" between the current approach and how they could be addressed. First, Strix selects a globally optimal edge to explore while SemML suggest actions locally. In particular,

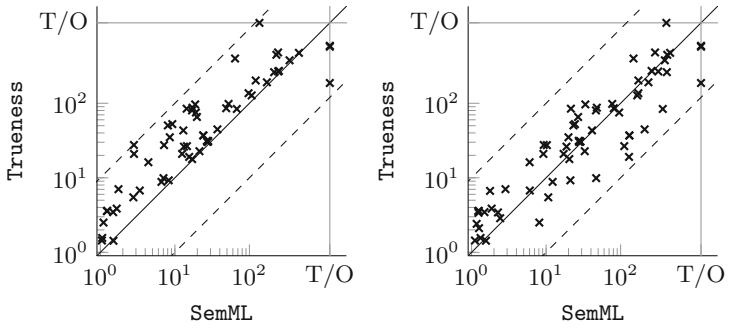

Fig. 6. Scatter plot comparing `Strix` with guidance provided by `SemML` and the default `Trueness`. On the left, we depict the total runtime excluding time spent for computing the guidance, and on the right we show the total time. We plot all models for which at least one method produced a result and count timeouts as 20 min (twice the timeout of 10 min). Note that the plot is logarithmic. The dashed lines denote a 10x difference.

our scoring is not trained to compare edges of two different states. While trueness seems to be a good compromise for the time being, we believe that (through significant engineering effort) `Strix` can be modified to accommodate local recommendations, or, alternatively, a more sophisticated indicator of a state's global relevance can be learnt. Second, `Strix` performs two searches, one for the environment and one for the system player. However, the parity games we deal with are not entirely symmetric – environment always moves first. Thus, we cannot directly apply `SemML`'s ranking to environment states, as they have a different structure. Here, we believe that the best solution is to train a separate model for the environment (or rather, six further models). Thirdly, `Strix` only constructs the automaton explicitly and computes the game implicitly. As such, `Strix` requests scoring information only for edges in the automaton and not in the game. This can be addressed by closely integrating the scoring computation with the exploration of the automaton – instead of rebuilding the game for each edge (p, q), we can compute all scores for all outgoing edges of p at once. Finally, as we mentioned, `Strix` by default applies a decomposition approach which builds several sub-automata. These also are equipped with semantic labelling, however with a different meaning – enough to create a significant hurdle for our learning approach. We note that `Strix` actually builds automata by communicating with `Owl` through a highly optimized interface between Java and C++, significantly complicating passing information back and forth between the processes.

7 Conclusion

We demonstrated that semantic labelling can be exploited for practical gains in LTL synthesis. Our experimental evaluation shows that we vastly outperform the simple approach of [22], the first step in this direction. Moreover, despite several

mismatches, our approach shows promising results for real world applications of this idea, i.e. when combined with the state-of-the-art tool `Strix`.

Future Work. As discussed above, the main point for future work is a tight, tailored integration with `Strix`. In particular, we want to modify our approach to be applicable to the decomposition methods of `Strix`, modify `Strix` to consider local guidance, and actually learn for the precise task required by `Strix`.

Aside from this, we believe that there might be further interesting features (hand-crafted or learnt) which could provide us with additional insights. In particular, we want to employ automated feature extraction, through more sophisticated model architectures such as *transformers* or *graph neural networks*.

References

1. Artifact for “Guessing Winning Policies in LTL Synthesis by Semantic Learning". Zenodo (2023). https://doi.org/10.5281/zenodo.7876095
2. Artifact for “Guessing Winning Policies in LTL Synthesis by Semantic Learning". Zenodo (2023). https://doi.org/10.5281/zenodo.7876096
3. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. RAIRO Theor. Inform. Appl. **36**(3), 261–275 (2002). https://doi.org/10.1051/ita:2002013
4. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasi-polynomial time. SIAM J. Comput. **51**(2), 17–152 (2022). https://doi.org/10.1137/17m1145288
5. Cosler, M., Schmitt, F., Hahn, C., Finkbeiner, B.: Iterative circuit repair against formal specifications. arXiv preprint arXiv:2303.01158 (2023)
6. Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 291–308. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_16
7. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
8. Esparza, J., Křetínský, J.: From LTL to deterministic automata: a safraless compositional approach. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 192–208. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_13
9. Esparza, J., Křetínský, J., Raskin, J.-F., Sickert, S.: From LTL and limit-deterministic büchi automata to deterministic parity automata. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 426–442. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_25
10. Esparza, J., Kretínský, J., Raskin, J., Sickert, S.: From linear temporal logic and limit-deterministic büchi automata to deterministic parity automata. Int. J. Softw. Tools Technol. Transf. **24**(4), 635–659 (2022). https://doi.org/10.1007/s10009-022-00663-1
11. Esparza, J., Kretínský, J., Sickert, S.: One theorem to rule them all: a unified translation of LTL into ω-automata. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, 09–12 July 2018, pp. 384–393. ACM (2018). https://doi.org/10.1145/3209108.3209161

12. Esparza, J., Kretínský, J., Sickert, S.: A unified translation of linear temporal logic to ω-automata. J. ACM **67**(6), 33:1-33:61 (2020). https://doi.org/10.1145/3417995
13. Fearnley, J.: Efficient parallel strategy improvement for parity games. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 137–154. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_8
14. Fearnley, J., Jain, S., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, 10–14 July 2017, pp. 112–121. ACM (2017). https://doi.org/10.1145/3092282.3092286
15. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04761-9_15
16. Gaiser, A., Křetínský, J., Esparza, J.: Rabinizer: small deterministic automata for LTL(**F**,**G**). In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 72–76. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_7
17. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 455–459. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_34
18. Jacobs, S., et al.: The reactive synthesis competition (SYNTCOMP): 2018–2021. arXiv preprint arXiv:2206.00251 (2022)
19. Jurdzinski, M.: Deciding the winner in parity games is in UP \cap co-up. Inf. Process. Lett. **68**(3), 119–124 (1998). https://doi.org/10.1016/S0020-0190(98)00150-1
20. Komárková, Z., Křetínský, J.: Rabinizer 3: safraless translation of LTL to small deterministic automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 235–241. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_17
21. Křetínský, J., Garza, R.L.: Rabinizer 2: small Deterministic Automata for LTL/GL. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 446–450. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_32
22. Křetínský, J., Manta, A., Meggendorfer, T.: Semantic labelling and learning for parity game solving in LTL synthesis. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 404–422. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_24
23. Kretinsky, J., Meggendorfer, T., Prokop, M., Rieder, S.: Guessing winning policies in LTL synthesis by semantic learning (2023)
24. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 543–550. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_34
25. Křetínský, J., Meggendorfer, T., Sickert, S., Ziegler, C.: Rabinizer 4: from LTL to your favourite deterministic automaton. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 567–577. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_30
26. Kretínský, J., Meggendorfer, T., Waldmann, C., Weininger, M.: Index appearance record with preorders. Acta Inform. **59**(5), 585–618 (2022). https://doi.org/10.1007/s00236-021-00412-y
27. Kupferman, O., Rosenberg, A.: The blow-up in translating LTL to deterministic automata. In: van der Meyden, R., Smaus, J.-G. (eds.) MoChArt 2010. LNCS (LNAI), vol. 6572, pp. 85–94. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20674-0_6

28. Lehtinen, K., Parys, P., Schewe, S., Wojtczak, D.: A recursive approach to solving parity games in quasipolynomial time. Log. Methods Comput. Sci. **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:8)2022
29. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL satisfiability checking revisited. In: Sánchez, C., Venable, K.B., Zimányi, E. (eds.) 2013 20th International Symposium on Temporal Representation and Reasoning, Pensacola, 26–28 September 2013, pp. 91–98. IEEE Computer Society (2013). https://doi.org/10.1109/TIME.2013.19
30. Liu, T.: Learning to rank for information retrieval. Found. Trends Inf. Retr. **3**(3), 225–331 (2009). https://doi.org/10.1561/1500000016
31. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Inform. 3–36 (2019). https://doi.org/10.1007/s00236-019-00349-3
32. Meyer, P.J., Luttenberger, M.: Solving mean-payoff games on the GPU. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 262–267. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_17
33. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: explicit reactive synthesis strikes back! In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 578–586. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_31
34. Osborne, M.J.: An introduction to game theory (2004)
35. Pedregosa, F., et al.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
36. Piterman, N.: From nondeterministic buchi and streett automata to deterministic parity automata. In: Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12–15 August 2006, Seattle, pp. 255–264. IEEE Computer Society (2006). https://doi.org/10.1109/LICS.2006.28
37. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
38. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ausiello, G., Dezani-Ciancaglini, M., Della Rocca, S.R. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0035790
39. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, 24–26 October 1988, pp. 319–327. IEEE Computer Society (1988). https://doi.org/10.1109/SFCS.1988.21948
40. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: de Alfaro, L. (ed.) FoSSaCS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00596-1_13
41. Schmitt, F., Hahn, C., Rabe, M.N., Finkbeiner, B.: Neural circuit synthesis from specification patterns. In: Ranzato, M., Beygelzimer, A., Dauphin, Y.N., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, 6–14 December 2021, Virtual, pp. 15408–15420 (2021). https://proceedings.neurips.cc/paper/2021/hash/8230bea7d54bcdf99cdfe85cb07313d5-Abstract.html
42. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17

43. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the Symposium on Logic in Computer Science (LICS 1986), Cambridge, Massachusetts, 16–18 June 1986, pp. 332–344. IEEE Computer Society (1986)