Programmatic Strategies for Real-Time Strategy Games

Julian R. H. Mariño,[1] Rubens O. Moraes,[2] Tassiana C. Oliveira,[2] Claudio Toledo,[1] Levi H. S. Lelis[3]

[1] Departamento de Sistemas de Computação, ICMC, Universidade de São Paulo, Brazil
[2] Departamento de Informática, Universidade Federal de Viçosa, Brazil
[3] Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
julianmarino@usp.br, rubens.moraes@ufv.br, tassiana.rios@ufv.br, claudio@icmc.usp.br, levi.lelis@ualberta.ca

Abstract

Search-based systems have shown to be effective for planning in zero-sum games. However, search-based approaches have important disadvantages. First, the decisions of search algorithms are mostly non-interpretable, which is problematic in domains where predictability and trust are desired such as commercial games. Second, the computational complexity of search-based algorithms might limit their applicability, especially in contexts where resources are shared among other tasks such as graphic rendering. In this work we introduce a system for synthesizing programmatic strategies for a real-time strategy (RTS) game. In contrast with search algorithms, programmatic strategies are more amenable to explanations and tend to be efficient, once the program is synthesized. Our system uses a novel algorithm for simplifying domain-specific languages (DSLs) and a local search algorithm that synthesizes programs with self play. We performed a user study where we enlisted four professional programmers to develop programmatic strategies for μRTS, a minimalist RTS game. Our results show that the programs synthesized by our approach can outperform search algorithms and be competitive with programs written by the programmers.

Introduction

Search and learning-based algorithms represent the current state-of-the-art approaches for playing zero-sum games, e.g., AlphaZero (Silver et al. 2018) and AlphaStar (Vinyals et al. 2019). One disadvantage of such approaches is that their decisions are often non-interpretable, which can be an issue if the artificial agent is deployed in scenarios where predictability, explainability, and trust are important, such as commercial games. Programmatic strategies, which we refer to as scripts, might not be as strong as strategies derived by search or reinforcement learning algorithms. However, scripts can more easily be interpreted and modified by a domain expert. The computer games industry heavily relies on scripts for controlling artificial agents because game designers and programmers are able to understand, predict, and thus trust the agent behavior in production. In the industry, scripted strategies are written by professional programmers in a trial-and-error process as they try to understand how other agents (including the player) could react to the strategy that is encoded in a script. The process of manually writing scripts can be time consuming. Moreover, strategies written by programmers are fixed and, if a player creates new content for a game (e.g., a new map for playing a real-time strategy game), then the artificial agents might have to play a strategy encoded in a script developed for a different map.

In this paper we introduce Local Search and Language Simplifier (LS2), a system for synthesizing scripts for real-time strategy (RTS) games. The inputs of LS2 are an RTS map for which a strategy is to be synthesized and a domain-specific language (DSL). LS2 returns a script encoding a strategy that is specialized for the map provided as input.

The DSL is designed to be expressive enough to allow for the synthesis of scripts encoding effective strategies, but also restrictive enough to prune from the synthesis space programs encoding weak strategies. Since strategies can change depending on the game map, features that make a DSL effective for a given map might not be necessary for another map. LS2 uses an algorithm we call Domain-Specific Language Simplifier (Lasi) to simplify the DSL according to the map provided as input. Lasi receives as input a trace of the game, i.e., a sequence of state-action pairs starting at the game's start state and finishing at a terminal state. This trace can be generated, for example, by an algorithm playing the game against itself or by a human demonstrator. Then, Lasi greedily chooses the features from the input DSL that allows one to synthesize a program that reproduces the trace provided as input. The features selected by this greedy approach define the simplified DSL. The intuition behind Lasi's procedure is that the DSL should contain only the features deemed as important by the agent who generated the game trace, thus allowing the synthesis to search on a more promising program space. Once the DSL is simplified LS2 uses a local search algorithm with self play to search over the simplified program space.

We evaluate LS2 on four maps of μRTS, a real-time strategy (RTS) game designed for research purposes. We enlisted four professional programmers to write scripts for all four maps used in our experiment. In addition to the scripts written by programmers, we also compare the LS2's scripts with search algorithms and other scripts from the μRTS codebase in a tournament-style experiment. A script LS2 synthesized obtained the highest winning rate in two of the maps tested and a script written by one of the programmers obtained the highest winning rate in the other two maps. In addition to this quantitative analysis, we also performed a qualitative analysis of the interpretability of the LS2 scripts, showing that the scripts synthesized in our experiments can be interpretable. Our analysis also showed that LS2 was able to synthesize scripts encoding strategies similar to those created by the programmers, but with important optimizations that would be difficult for a human to discover by themself.

Related Work

Our work is related to the fields of program synthesis, programmatic reinforcement learning, automatic generation of scripts, and planning in real-time zero-sum domains. We review relevant works in each of these areas in this section.

In program synthesis one has to synthesize a program that satisfies a given specification (Gulwani, Polozov, and Singh 2017). Approaches for program synthesis include brute-force search (Alur et al. 2013), constraint satisfaction (Jha et al. 2010), genetic programming (Koza 1992), machine learning (Balog et al. 2017), and hybrid systems that combine search with a learned function (Liang, Jordan, and Klein 2010; Menon et al. 2013; Murali, Chaudhuri, and Jermaine 2017). All these works assume some form of supervision, which could be a logical formula that needs to be satisfied or a set of input-output training pairs. Our problem is not supervised, the synthesis algorithm has access to the model of the game and it has to synthesize a program encoding a strategy for playing the game.

In programmatic reinforcement learning (PRL) one represents policies as computer programs (Bastani, Pu, and Solar-Lezama 2018; Verma et al. 2018, 2019), which might be more easily interpreted and formally verified than policies encoded in black-box representations. Instead of learning policies as was done in PRL, in our work we learn strategies for playing games. Moreover, it is possible that the idea we introduce for simplifying DSLs might also be applicable to speed up the program synthesis process in PRL.

Dynamic Scripting (DS) is a reinforcement-learning-based technique for synthesizing scripts for zero-sum role-playing games. DS allows for the generation of scripts by extracting rules from an expert-designed rule base according to a learned policy (Spronck, Sprinkhuizen-Kuyper, and Postma 2004). DS has been applied to RTS games, but the synthesized script is limited to a fixed number of consecutive if-then clauses. Our method is more expressive as it searches in the space of scripts defined by a DSL that allows for other program structures (e.g., for loops).

Program synthesis has been applied to zero-sum games in the context of synthesizing scripts to work as evaluation functions (Benbassat and Sipper 2011) and as pruning policies (Benbassat and Sipper 2012) for tree search algorithms. By contrast, in this paper we investigate the use of script synthesis for deriving complete strategies for zero-sum games.

Program synthesis has also been applied to other non-zero-sum games. Canaan et al. (2018) use an evolutionary approach for generating strategies to play a cooperative game. Similarly to DS, Canaan et al.'s method generates sequences of if-then rules to play the game. De Freitas, de Souza, and Bernardino (2018) use a genetic-programming approach for evolving controllers for a Mario AI simulator. Butler, Torlak, and Popović (2017) present a method for synthesizing strategies for the puzzle game nonograms. De Freitas, de Souza, and Bernardino, and Butler, Torlak, and Popović's approaches are different than our method because they deal with single-agent problems, while we deal with two-players zero-sum games.

The Script Synthesis Problem

Let $\mathcal{G}$ be a zero-sum game, i and $-i$ the pair of players for $\mathcal{G}$, $\mathcal{S}$ the set of states of the game, s_{init} the start state and $\mathcal{A}_i(s)$ the set of actions player i can perform at state s. A decision point for player i is a state $s \in \mathcal{S}$ where i can act. A strategy is a function $\sigma_i : \mathcal{S} \rightarrow \mathcal{A}_i$ for player i, mapping a state s to an action a, for every decision point of player i in $\mathcal{G}$. A script is a strategy denoted as a function $p(s)$ that returns a legal action $a \in \mathcal{A}_i$ for player i at state s. The value of the game rooted at state s is denoted by $\mathcal{V}(s, p_i, p_{-i})$, which indicates player i's utility if i and $-i$ follow the strategies given by p_i and p_{-i}, respectively. Since $\mathcal{G}$ is a zero-sum game, the utility of player $-i$ is given by $-\mathcal{V}(s, p_i, p_{-i})$.

A Domain-Specific Language (DSL) is a declarative language that defines a space of programs in a particular domain (Van Deursen, Klint, and Visser 2000). Let $[\![D]\!]$ be the set of scripts that can be synthesized with a DSL D. We are interested in synthesizing a script $p_i \in [\![D]\!]$ for player i that maximizes the value of the game while player $-i$ plays the game with a script p_{-i} from $[\![D]\!]$ that minimizes the value of the game. We formulate the script synthesis problem as,

$$\max_{p_i \in [\![D]\!]} \min_{p_{-i} \in [\![D]\!]} \mathcal{V}(s_{init}, p_i, p_{-i}) \, . \qquad (1)$$

The strategies encoded by scripts p_i and p_{-i} that solve the Equation 1 define a Nash equilibrium profile if one considers only strategies encoded by scripts in $[\![D]\!]$ as valid strategies. Searching for a Nash profile can be computationally intractable, specially if D allows for the synthesis of a large set of scripts $[\![D]\!]$. LS2 uses a self play procedure with local search for approximating a solution to Equation 1.

Domain-Specific Languages

We define a DSL to synthesize scripts for zero-sum games through a context-free grammar (CFG) $G = (V, \Sigma, R, S)$, where V is a finite set of non-terminals, Σ is a finite set of terminals, R is a finite set of relations corresponding to the grammar production rules, and S is the start symbol.

As an example, the grammar G_1 below defines a DSL.

$$\begin{aligned} S &\rightarrow C \mid \text{if}(B) \text{ then } C \text{ else } C \mid C \, \text{if}(B) \text{ then } C \\ C &\rightarrow c_1 \mid c_2 \mid c_3 \\ B &\rightarrow b_1 \mid b_2 \mid b_3 \end{aligned}$$

Here, $V = \{S, C, B\}$, $\Sigma = \{c_1, c_2, c_3, b_1, b_2, b_3,$ if, then, else$\}$, R is the set of relations, (e.g., $C \rightarrow c_1$ and $B \rightarrow b_1$), and S is the start symbol. G_1 allows scripts with a single command (c_1, c_2, or c_3), scripts with an if-then-else, or scripts with a single command followed by an if-then. We represent the scripts as derivation trees, where the root

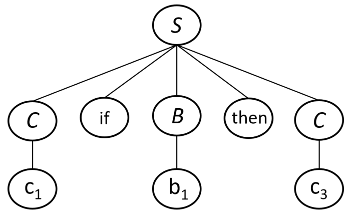

Figure 1: Derivation tree for "`c1 if (b1) then c3`".

node in the tree is the start symbol S, the internal nodes are non-terminals from V, and leaf nodes are terminals from Σ. Figure 1 shows an example of a derivation tree.

Domain-Specific Language Simplifier

The size of the space of possible programs to solve the script synthesis problem can grow quickly with the size of the CFG defining the DSL (i.e., the number of relations and symbols). The search for effective strategies can become infeasible for large grammars. On the other hand, if the grammar is too constrained, then synthesized programs might not be expressive enough to encode strong game strategies. In this section we introduce Lasi, a method that simplifies DSLs while balancing the language size and expressiveness for a given task. LS2 uses Lasi before running a self play procedure with local search to approximate a solution to Equation 1.

Lasi receives a game $\mathcal{G}$, a grammar $G = (V, \Sigma, R, S)$ defining a DSL, and a sequence of state-action pairs, $T = \{(s_{init}, a_1), (s_2, a_2), \cdots, (s_n, a_n)\}$, which we call a trace, from s_{init} to a terminal state $s_{n+1} \in \mathcal{F}$. The trace can be generated by an agent playing the game (e.g., tree search algorithm or human demonstrator). Lasi selects a subset of Σ to define a grammar G' that is more restrictive than G, but that still allows a program synthesized with a DSL defined by G' to reproduce the trace T received as input. Intuitively, the system designer should provide an expressive grammar G as input and Lasi automatically produces a more restrictive grammar G'. We describe how Lasi defines grammar G' in the next section.

The DSL Simplification Problem

It is common in a program synthesis task that the system designer defines a set of high-level functions to be part of the DSL. We will call these functions domain-specific functions (DSFs). In our application domain, a DSF returns an action for a given state of the game. For example, if grammar G_1 is used in a game played in a grid-world, then command c_1 could encode the knowledge to allow the agent to walk out of a room through the exit door. That is, for any state of the game, c_1 returns an action that takes the agent closer to leaving the room through the exit door. Similarly, terminal symbols representing Boolean expressions can be defined as domain-specific Boolean functions (DSBs). DSBs return a Boolean value for a given state of the game. For example, in G_1 the terminal symbol b_1 could be a DSB that returns true if the agent is waiting by the exit door and false otherwise. We formulate the DSL Simplification Problem as follows,

Definition 1 (DSL Simplification Problem) *Let $[\![G]\!]$ be the set of programs that can be synthesized with grammar $G = (V, \Sigma, R, S)$ and T be a trace with state-action pairs. In the DSL simplification problem one has to find a grammar $G' = (V, \Sigma', R', S)$ with the smallest set Σ', such that $\Sigma' \subseteq \Sigma$ and $R' \subseteq R$, for which there exists a script $p \in [\![G']\!]$ such that $p(s_m) = a_m$ for all $(s_m, a_m) \in T$.*

In a DSL Simplification Problem we assume that all terminals removed from Σ (i.e., $\Sigma \setminus \Sigma'$) are either DSFs or DSBs. That is, the grammar can include non-DSFs and non-DSBs terminals, but they are not be considered for removal in the simplification task. We also assume that G is expressive enough so that there exists a script $p \in [\![G]\!]$ with $p(s_m) = a_m$ for all $(s_m, a_m) \in T$, otherwise the simplification task does not have a solution.

Simplification Problem as Set Cover

Subset R' of R can be trivially computed once subset Σ' of Σ is defined. This is because terminal symbols appear only on the righthand side of the relations and, for that, subset R' is the set R with the relations involving symbols $\Sigma \setminus \Sigma'$ removed. Thus, the task of simplifying G into G' is equivalent to selecting a subset Σ' of Σ.

Removing DSFs We start with the selection of terminal symbols defined by DSFs. Each terminal symbol represented as a DSF in the grammar G provided as input to Lasi returns an action $a \in \mathcal{A}(s)$ for any state s in the trace T. The problem of selecting DSFs can be seen as a set cover problem, where each DSF represents a subset of the actions in the trace T and one needs to find the smallest subset of DSFs that covers all actions in T.

While the set cover problem is NP-hard (Garey and Johnson 1979), a polynomial-time greedy algorithm provides a good approximation. Let Q be the set of state-action pairs (s, a) initialized with all (s, a) in T. Let Σ' be the set of DSFs selected by the greedy algorithm, which is initially empty. One iteratively adds to Σ' a DSF o that covers the largest number of actions in Q, i.e., the DSF that maximizes $|\{(s, a)|(s, a) \in Q \wedge o(s) = a\}|$. We remove from Q the state-action pairs in $\{(s, a)|(s, a) \in Q \wedge o(s) = a\}$ for the selected o. This procedure is repeated until all actions in T are covered by a DSF in Σ'.

In the worst case all symbols in Σ are DSFs and we select all of them to ensure action coverage, i.e., $\Sigma = \Sigma'$. In this case the algorithm's time complexity is $O(|\Sigma|^2 \cdot |T|)$. This is because each iteration of the algorithm has complexity of $O(|\Sigma| \cdot |T|)$ (each DSF in Σ is tested for its coverage) and the algorithm performs $|\Sigma|$ iterations in the worst case. The performance ratio of the solution encountered by the greedy algorithm is $\ln(|\Sigma|) - \ln(\ln(|\Sigma|)) - \Theta(1)$ (Slavík 1996).

Removing DSBs Lasi adds to Σ' all DSBs from Σ but those that return either true or false for all states in T, i.e., it does not add the DSBs in $\{b|b \in \Sigma \wedge b(s) = \text{false}\ \forall s \in T\}$ nor the DSBs in $\{b|b \in \Sigma \wedge b(s) = \text{true}\ \forall s \in T\}$. Let b be a DSB that returns false for all states in T. Any program using b can be rewritten without b. That is, the commands `if(b){c1}`, `if(not b){c1}`, `while(b){c1}`, `while(not b){c1}`, and `var` $\leftarrow$ `b` can be replaced by ϵ, `c1`, ϵ, `while(True){c1}`, and `var` $\leftarrow$ `false`, respectively, where ϵ is an empty string. Similarly, all programs using DSBs that always return true can be replaced by equivalent programs that do not use the DSBs. The DSL only needs DSBs b for which $b(s)$ returns true for some states and false for other states in the trace T to be able to synthesize a program that reproduces T. Lasi runs in $O(|\Sigma| \cdot |T|)$ for removing DSBs because each terminal in Σ might be a DSB that needs to be verified against all states in T.

Synthesizing Scripts with Self Play

Once Lasi simplifies the DSL, LS2 uses the self play procedure described in Algorithm 1 to synthesize a script for game $\mathcal{G}$. Algorithm 1 starts by generating a random script from the DSL D (see line 1). This is achieved by starting from the D's initial symbol and selecting rules uniformly at random to be applied to non-terminal symbols; a script is returned once it contains no non-terminal symbols.

Algorithm 1 iteratively improves the initial random solution p with a local search algorithm. The local search receives the game $\mathcal{G}$, the DSL D, and current script p, and returns an approximated best response for p, denoted as p_{BR}. The script p_{BR} is then attributed to p if p_{BR} defeats p. In the context of μRTS, to ensure fairness, p and p_{BR} play two matches, one with p as player 1 and another with p_{BR} as player 1. We consider that p_{BR} defeats p if it either wins both matches or if it wins one match and draws the other. In the next iteration, the local search will approximate a best response to the current's iteration best response.

In our implementation of Algorithm 1 we use a local search algorithm that creates m mutated versions of the script p provided as input and generates i other scripts at random. All $m + i$ newly generated scripts are evaluated with two matches against p, where we vary which script assumes the role of player 1 for fairness; the best performing script is returned as an approximated best response to p. The best performing script is determined by a score function that attributes the value of 1 to a victory, 0 to a loss, and 0.5 to a draw; we break ties at random. A script p is mutated by randomly selecting a node representing a non-terminal symbol in the derivation tree representing p and replacing the sub-tree rooted at the selected node by a sub-tree randomly generated according to the rules of the CFG describing the DSL. For example, if node B in the tree shown in Figure 1 is selected for mutation, then the sub-tree b_1 would be replaced by another sub-tree (b_1, b_2, or b_3 in our example). If the root of the tree is selected, then the entire tree is replaced.

Empirical Methodology

We evaluate LS2's scripts in μRTS (Ontañón 2017) by comparing them with tree search algorithms and scripts written by programmers. We are primarily interested in evaluating the effect of Lasi in LS2 and in comparing the scripts written by professional designers with those LS2 synthesizes. We present a quantitative evaluation of all approaches in terms of strength of play, and a qualitative evaluation of the interpretability of the synthesized scripts. Our implementation of LS2 is available online.[1]

Algorithm 1 Self play with local search

```
Require: Game $\mathcal{G}$, DSL $D$, number of steps $n$.
Ensure: Script $p$ for playing the game $\mathcal{G}$.
1: $p \leftarrow$random-script($D$)
2: for $k = 1$ to $n$ do
3:     $p_{BR} \leftarrow$ local-search($\mathcal{G}$, $D$, $p$)
4:     if $p_{BR}$ defeats $p$ in $\mathcal{G}$ then
5:         $p \leftarrow p_{BR}$
6: return $p$
```

Problem Domain: μRTS

We chose to use μRTS in our experiments because it has an active research community with competitions being organized (Ontañón et al. 2018), with all competing algorithms available in a single codebase.[2] Moreover, in 2017 a script won two tracks of the μRTS competition, cf. Table 1 of (Ontañón et al. 2018), demonstrating that scripts can outperform other approaches for this type of game. Finally, μRTS can offer a diverse set of challenges because the agents can be easily evaluated on different maps of the game.

Most μRTS matches start with each player controlling a set of units known as workers on a gridded map. Workers can be used to collect resources, build structures, and battle the opponent. In some of the maps, players also start with a structure called base, which is used to train workers and store resources. In addition to the base, workers can build a barracks, which can be used to train the following units: light, ranged, and heavy. Units differ in how much damage they can take before being removed from the game, how much damage they can inflict to other units, and how close they need to be from an opponent unit to be able to attack it. A player wins a match if they are able to remove all the other player's units from the game. Every algorithm is allowed 100 milliseconds for planning before deciding the units actions. We call a unit everything that can issue an action, including bases and barracks.

We used four μRTS maps in our evaluations, with a map of size 18×8 being novel. This map contains no worker units neither resources to be harvested, and has three heavy, four ranged units, and a base. The ranged units are initially placed in front of the heavy units and closer to the opponent units, which we hypothesized to be suboptimal as the ranged units are able to inflict damage to enemy units from far, but they are weaker than heavy units and can be quickly eliminated from the game. We expected to see strategies in which ranged and heavy units switch positions so that the latter protect the former. In the second map, of size 8×8, each player starts with a base and one worker. The third map is a map of size 9×8 where players start with a base, a worker, and the resources are placed as a wall separating the players. The fourth map, of size 24×24, is divided in the middle by a wall, and players start with two bases and four

[1]https://github.com/julianmarino/LS2

[2]https://github.com/santiontanon/microrts

workers, one base placed on each side of the map. The maps 8×8, 9×8, and 24×24 were used in μRTS competitions.

Generating Trace for Simplifying the DSL

In our experiments we use A3N (Moraes et al. 2018) to generate Lasi's required demonstration trace by having it play a match with itself. We chose A3N because Moraes et al. showed that it performs well in a variety of maps. We could have also used other search algorithms such as Naive Guided Sample (GNS) (Yang and Ontanón 2019) or human demonstrations to produce the trace.

A3N requires a set of scripts as input and, for a subset of the units controlled by the player (known as restricted units), it considers only the actions returned by the scripts; all other actions are disregarded during search. A3N accounts for all legal actions of the remaining units (known as unrestricted units). A3N uses a policy to decide which units are unrestricted in each state of the game. Since we would like to explore the action space, we use a policy that randomly chooses three unrestricted units in each state. The size of the unrestricted set was chosen empirically in preliminary experiments.

Since A3N is being used in the context of script synthesis, we do not provide expert-designed scripts as input to A3N, as is described in its original paper (Moraes et al. 2018). Instead, we generate a random script that obeys our DSL and provide it as input to A3N. This randomly synthesized script defines the action of the restricted units. Naturally, the quality of the actions returned by A3N depends on the quality of the set of scripts provided as input and the use of a randomly synthesized script reduces its strength of play. We compensate for this reduction in quality by allowing A3N more planning time. Instead of 100 ms, we allow A3N 500 ms of planning time in the self play match that generates T.

Competing Agents

We use the following search algorithms: Portfolio Greedy Search (PGS) (Churchill and Buro 2013), Stratified Strategy Selection (SSS) (Lelis 2017), the MCTS version of Puppet Search (PS) (Barriga, Stanescu, and Buro 2017b), Strategy Tactics (STT) (Barriga, Stanescu, and Buro 2017a), NaïveMCTS (NS) (Ontañón 2017), A3N (Moraes et al. 2018), and Naive Guided Sample (GNS) (Yang and Ontanón 2019). We also use the scripts Worker Rush (WR), Light Rush (LR), Heavy Rush (HR), and Ranged Rush (RR) (Stanescu et al. 2016). These scripts train one worker to collect resources and then continuously train one type of unit that is sent to battle to opponent. WR, LR, HR, and RR train workers, light, heavy, and ranged units, respectively. Although simple, these scripts were shown to perform well in a wide range of maps. We also experiment with a baseline of LS2, denoted as LS, that does not employ Lasi. That is, LS uses the self play algorithm with local search to synthesize a script while using the original DSL provided as input.

μLanguage: a Domain-Specific Language for μRTS

We have developed a DSL for μRTS, which we name μLanguage, to allow for the development of scripts in μRTS. Both users in our study and synthesizers use the same DSL. The CFG below summarizes the μLanguage.

$$\begin{aligned}
S_1 &\to C\, S_1 \mid S_2\, S_1 \mid S_3\, S_1 \mid \epsilon \\
S_2 &\to \text{if}(B) \text{ then } \{C\} \mid \text{if}(B) \text{ then } \{C\} \text{ else } \{C\} \\
S_3 &\to \text{for (each unit } u) \, \{S_4\} \\
S_4 &\to C\, S_4 \mid S_2\, S_4 \mid \epsilon \\
B &\to b_1 \mid b_2 \mid \cdots \mid b_m \\
C &\to c_1\, C \mid c_2\, C \mid \cdots \mid c_n\, C \mid c_1 \mid c_2 \mid \cdots \mid c_n \mid \epsilon
\end{aligned}$$

S_1 is the initial symbol, ϵ is an empty string, and symbols $c_1, c_2, \cdots, c_n$ are DSFs and $b_1, b_2, \cdots, b_n$ are DSBs. μLanguage does not allow nested conditionals and nested loops, but it allows programs with if-clauses inside and outside for-loops and infinitely large sequences of DSFs.

Scripts Written by Programmers

We have enlisted four professional programmers, who are not involved in this research, to write one script for each of the four maps used in our experiments. All programmers used the same DSL we provide as input to LS2, so both system and programmers had access to the same space of programs. The experiment was carried out online, advertised by email, with the participation being anonymous. Each participant watched a 10-minute video explaining the rules of μRTS and showing a few examples of μLanguage, and also had access to a manual describing each DSF and DSB implemented in the μLanguage. The participants were allowed as much time as they wanted to develop their scripts. After the participants developed the four scripts they answered a questionnaire. Our study had 3 males and 1 female participants, with an average age of 26.5 years. The average years of programming experience was 5.5 years. All participants had taken at least one course on artificial intelligence and three participants had played an RTS game at least one time, the average minutes spent to write all four scripts was 66.25 minutes. We will refer to the scripts written by the programmers as P_1, P_2, P_3, and P_4 in our table of results.

Empirical Results

Evaluation of Strength of Play

Our experiment consists in performing ten independent runs of LS and LS2 for each map. We use $n = 400$ for Algorithm 1, and $m = 70$ and $i = 20$ for our local search algorithm. Each run takes approximately 30 minutes for maps of sizes 18×8, 8×8, and 9×8, and 7 hours for the 24×24 map. We use one machine with 56 cores for our experiments.

We determine the script returned by LS and LS2 by performing a round-robin tournament among the scripts returned in each run of LS and LS2. The purpose of the tournament is to select the best solution encountered by LS and LS2 in the ten independent runs. In our round-robin tournaments, each strategy plays against all the other strategies ten times, five as player 1 and five as player 2. This is to ensure fairness in μRTS. The time required to generate one trace with A3N is negligible compared to the time required to perform the 10 independent runs of the local search algorithm and the round-robin tournament to select the best

Map 8 $\times$ 8

	WR	GNS	P_3	P_1	P_4	P_2	LS	LS2	Avg
STT	30.0	0.0	15.0	55.0	30.0	20.0	35.0	6.0	54.1
NS	20.0	30.0	40.0	20.0	10.0	10.0	51.0	6.0	56.4
A3N	30.0	10.0	70.0	50.0	30.0	15.0	69.0	30.0	62.8
WR	-	30.0	100.0	50.0	50.0	75.0	60.0	25.0	75.6
GNS	70.0	-	65.0	60.0	30.0	90.0	67.0	69.0	81.9
P_3	0.0	35.0	-	25.0	25.0	50.0	45.0	11.0	55.1
P_1	50.0	40.0	75.0	-	50.0	0.0	5.0	0.0	62.2
P_4	50.0	70.0	75.0	50.0	-	50.0	65.0	20.0	75.6
P_2	25.0	10.0	50.0	100.0	50.0	-	70.0	64.0	76.5
LS	40.0	33.0	55.0	95.0	35.0	30.0	-	26.0	66.2
LS2	75.0	31.0	89.0	100.0	80.0	36.0	74.0	-	83.9

Map 9 $\times$ 8

	A3N	RR	P_4	P_1	P_3	P_2	LS	LS2	Avg
LR	0.0	0.0	100.0	0.0	0.0	0.0	74.0	59.0	46.1
PS	0.0	0.0	100.0	35.0	0.0	0.0	82.0	73.0	54.4
GNS	60.0	60.0	85.0	65.0	20.0	0.0	86.0	81.0	70.4
A3N	-	20.0	100.0	80.0	40.0	10.0	100.0	92.0	79.5
RR	80.0	-	100.0	75.0	0.0	0.0	100.0	95.0	80.6
P_4	0.0	0.0	-	0.0	0.0	0.0	36.0	40.0	16.9
P_1	20.0	25.0	100.0	-	15.0	0.0	83.0	47.0	62.8
P_3	60.0	100.0	100.0	85.0	-	0.0	100.0	99.0	89.0
P_2	90.0	100.0	100.0	100.0	100.0	-	100.0	100.0	99.4
LS	0.0	0.0	64.0	17.0	0.0	0.0	-	41.0	27.9
LS2	8.0	5.0	60.0	53.0	1.0	0.0	59.0	-	39.4

Map 18 $\times$ 8

	NS	A3N	P_1	P_4	P_3	P_2	LS	LS2	Avg
HR	40.0	45.0	50.0	50.0	0.0	0.0	20.0	10.0	37.8
PS	20.0	35.0	50.0	50.0	50.0	50.0	25.0	20.0	42.8
STT	50.0	30.0	65.0	45.0	75.0	40.0	49.0	56.0	52.5
GNS	25.0	45.0	65.0	55.0	55.0	25.0	53.0	58.0	53.8
NS	-	65.0	70.0	70.0	65.0	65.0	68.0	70.0	68.6
A3N	35.0	-	75.0	90.0	85.0	50.0	82.0	75.0	70.4
P_1	30.0	25.0	-	50.0	0.0	0.0	20.0	10.0	34.7
P_4	30.0	10.0	50.0	-	0.0	0.0	20.0	10.0	35.6
P_3	35.0	15.0	100.0	100.0	-	25.0	80.0	70.0	71.6
P_2	35.0	50.0	100.0	100.0	75.0	-	75.0	65.0	80.3
LS	32.0	18.0	80.0	80.0	20.0	25.0	-	10.0	57.4
LS2	30.0	25.0	90.0	90.0	30.0	35.0	90.0	-	68.5

Map 24 $\times$ 24

	WR	GNS	P_3	P_1	P_2	P_4	LS	LS2	Avg
STT	50.0	0.0	95.0	90.0	35.0	20.0	47.0	6.0	50.2
A3N	15.0	0.0	80.0	50.0	30.0	35.0	40.0	17.0	52.6
PS	45.0	35.0	80.0	65.0	50.0	45.0	44.0	32.0	54.8
WR	-	90.0	0.0	100.0	100.0	0.0	45.0	20.0	69.1
GNS	10.0	-	100.0	5.0	65.0	85.0	39.0	21.0	69.4
P_3	100.0	0.0	-	100.0	0.0	20.0	62.0	50.0	38.9
P_1	0.0	95.0	0.0	-	65.0	0.0	0.0	0.0	45.9
P_2	0.0	35.0	100.0	35.0	-	50.0	42.0	11.0	63.0
P_4	100.0	15.0	80.0	100.0	50.0	-	50.0	62.0	74.8
LS	55.0	61.0	38.0	100.0	58.0	50.0	-	50.0	67.4
LS2	80.0	79.0	50.0	100.0	89.0	38.0	50.0	-	82.6

Table 1: Average winning rate of the row player against the column player. The average winning rate of row players was computed considering matches played with all methods used in our experiment, and not only those shown in the table.

synthesized script. Once scripts of LS and LS2 are determined, we perform another round robin tournament with the synthesized scripts, tree search algorithms, and the scripts written by the programmers in our study.

We execute the experiment described above ten times and calculate the average winning rate considering the ten executions. The results are presented in Table 1, where each number shows the average winning rate of the row method against the column method (e.g., LS2 wins 89% of its matches against P_3 on the 8×8 map). The last column shows the average winning rate of the row methods. The table presents LS2, LS and the scripts the programmers wrote. In the interest of space, we present the best five and the best two methods among the remaining ones, in terms of average winning rate, in the rows and in the columns of the table.

LS2 achieves a much higher average winning rate than its baseline LS and wins more direct matches in all maps tested. This result shows that LS2's simplification scheme given by Lasi allows for a more focused search in the scripts space. LS2 achieves the highest average winning rate in the 8×8 and 24×24 maps. In the 9×8 and 18×8 maps, the highest average value is obtained by P_2. In the 18×8 map, LS2 was also outperformed by P_3 and by the search-based methods A3N and NS. A3N and NS can be very effective in the map 18×8 because they are able to micromanage well their units.

LS2 outperforms A3N in terms of average winning rate on the 8×8 and 24×24 maps. This result suggests that although the synthesis process is being constrained to a region of the scripts space containing strategies similar to the ones played by A3N, the script synthesizer is able to encounter stronger strategies than the ones derived by A3N in its regular setting (i.e., the one described by Moraes et al. (2018), which was used to generate the results of Table 1).

Table 2 shows the average winning rate of the methods evaluated across all four maps. The scripts synthesized by LS2 are outperformed only by programmer P_2, and performs similarly to the tree search algorithm GNS. These results are promising as they show that it is possible to automatically synthesize programmatic strategies for playing non-trivial games that are able to defeat scripts written by programmers and tree search algorithms. The table also highlights the importance of Lasi in the synthesis process as LS obtained only a 54.7% average winning rate across all maps.

Interpretability of Scripts

Table 3 shows the best scripts synthesized by LS2 and the best scripts written by programmers in terms of average winning rate for the 8×8 and 24×24 maps. The objective of this analysis is two-fold. First, we want to verify if the programmatic strategies LS2 synthesizes can be interpretable. Second, we want to perform a qualitative comparison of the LS2 scripts with those written by the programmers.

The script P_2 follows an strategy similar to the WR strategy. P_2 sends one worker to harvest resources (line 19) and the remaining workers are sent to attack the closest enemy unit (line 21). P_2 continuously trains workers (line 20). This strategy has shown to be effective in previous μRTS competitions. The script LS2 synthesized for map of size 8×8 (`LS2-8x8`) iterates through all units the player controls

Average Winning Rate on All Maps																
HR	SSS	PGS	LR	RR	NS	PS	STT	WR	P_4	P_1	LS	P_3	A3N	LS2	GNS	P_2
26.7	28.6	29.6	36.2	38.5	44.1	44.7	47.6	49.9	50.8	51.4	54.7	63.6	66.3	68.6	68.9	79.8

Table 2: Average winning rate across all four maps tested.

```
1  def LS2-8x8()
2      for(each unit u)
3          if(DistanceEnemy(u, worker, 5))
4              attack(u, worker, closest)
5          else
6              harvest(u)
7          train(u, worker, Right)
8          harvest(u)
9
10 def LS2-24x24()
11     for(each unit u)
12         if(HaveQtdUnitsHarvesting(2))
13             attack(u, worker, closest)
14             train(u, worker, right)
15         else
16             harvest(u)
17
18 def  P_2-8x8()
19     harvest(1)
20     train(worker, enemyDirection)
21     attack(worker, closest)
22
23 def  P_4-24x24()
24     train(worker, up)
25     harvest(5)
26     attack(worker, closest)
```

Table 3: Scripts synthesized by LS2 and the best two scripts written by the programmers for maps 18×8 and 24×24.

and, if a unit u is at a distance of 5 or less from an enemy unit, u is sent to attack the closest opponent unit (lines 3–4). If no enemy unit is nearby, the script will send workers to harvest resources (line 6). The strategy continuously train worker units (line 7). LS2's script has a seemingly unnecessary instruction in line 8 ("harvest(u)"). This is because, once we reach line 8, all units will have received an action to be performed in the game. However, if we remove line 8, the winning rate of this specific LS2 script is reduced from 50% to 0% in matches against P_3 (not shown in Table 1). We discovered that this extra instruction optimizes the pathfinding system of μRTS. That is, the pathfinding algorithm might fail to find a path for a unit because the path is momentarily blocked and the unit stays idle despite the script having issued an action for the unit. In these cases it is helpful to add an extra action for the units. In LS2's script, if worker units are unable to attack the enemy, then the units will move toward their resources, as indicated by the instruction "harvest," possibly clearing the path for the next round of actions. This is a case where LS2 is able to synthesize a script that contains a feature that would be difficult for human programmers to discover by themselves.

For the 24×24 map, script P_4 encodes a strategy that is similar to the strategy LS2 synthesized: both strategies train a large number of worker units and send them to attack. A key difference between the two scripts is the number of units used to collect resources. While P_4 uses five units to collect resources (line 25), the LS2 script uses two units (lines 12 and 13). This is also a case where LS2 is able to synthesize a script that contains a feature that might be difficult for human programmers to optimize by themselves.

The scripts synthesized by LS2 can be improved too. For example, in the 9×8 map, LS2 is able to synthesize a strategy that, similarly to P_2, trains ranged units and sends them to attack the opponent. One of the drawbacks of the LS2 strategy is that it also sends worker units to continuously build barracks, which exhaust almost all resources available to the player. The script also trains a large number of workers that clutter the region around the base, thus making it difficult for the workers themselves to move around and collect resources. The result on the map 9×8 suggests that better search algorithms could further improve the quality of the scripts LS2 synthesizes.

Conclusions

In this paper we introduced LS2, a system for synthesizing scripts for RTS games. LS2 employs Lasi, an algorithm for simplifying DSLs for RTS games. Lasi uses A3N to play the game against itself to generate a game trace. Then, Lasi greedily chooses the features from the DSL that allows one to synthesize a program that reproduces the trace generated by the search algorithm. The features selected by this greedy procedure define the simplified DSL. LS2 then uses a local search algorithm with self play to search in the space of programs defined by the simplified DSL. We evaluated the scripts synthesized by LS2 on μRTS by comparing the synthesized scripts with tree search algorithms, scripts written by four programmers, and scripts synthesized by a baseline that does not use the DSL simplification step. Our results show that the LS2 scripts were able to outperform the scripts synthesized by the baseline by a large margin. The LS2 scripts obtained the highest winning rate on two of the four tested maps and a script written by a programmer obtained the highest winning rate on the other two maps. A qualitative analysis of the scripts showed that LS2 was able to synthesize scripts encoding strategies similar to those derived by the programmers. The qualitative analysis also showed that LS2 was able to synthesize scripts with important optimizations that would be difficult for a human to discover by themself.

Acknowledgements

This research was partially supported by CNPq, CAPES, and Canada's CIFAR AI Chairs program. The research was carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0), the cluster Jupiter from Universidade Federal de Viçosa, and the clusters from the Compute Canada. We thank the anonymous reviewers and David da Silva Aleixo for great suggestions on an earlier version of this paper.

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-Guided Synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design*, 1–17.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder: Learning to Write Programs. In *Proceedings International Conference on Learning Representations*. OpenReviews.net.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017a. Combining Strategic Learning and Tactical Search in Real-Time Strategy Games. *Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* .

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017b. Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games. *IEEE Transactions on Computational Intelligence and AI in Games* .

Bastani, O.; Pu, Y.; and Solar-Lezama, A. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Advances in Neural Information Processing Systems*, 2499–2509.

Benbassat, A.; and Sipper, M. 2011. Evolving board-game players with genetic programming. In *Genetic and Evolutionary Computation Conference*, 739–742.

Benbassat, A.; and Sipper, M. 2012. Evolving both search and strategy for Reversi players using genetic programming. In *IEEE Conference on Computational Intelligence and Games*, 47–54.

Butler, E.; Torlak, E.; and Popović, Z. 2017. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Canaan, R.; Shen, H.; Torrado, R.; Togelius, J.; Nealen, A.; and Menzel, S. 2018. Evolving agents for the hanabi 2018 cig competition. In *2018 IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Churchill, D.; and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Proceedings of the Conference on Computational Intelligence in Games*, 1–8. IEEE.

De Freitas, J. M.; de Souza, F. R.; and Bernardino, H. S. 2018. Evolving Controllers for Mario AI Using Grammar-based Genetic Programming. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. IEEE.

Garey, M. R.; and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4(1-2): 1–119. ISSN 2325-1107. doi:10.1561/2500000010.

Jha, S.; Gulwani, S.; Seshia, S. A.; and Tiwari, A. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 215–224.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

Lelis, L. H. S. 2017. Stratified Strategy Selection for Unit Control in Real-Time Strategy Games. In *International Joint Conference on Artificial Intelligence*, 3735–3741.

Liang, P.; Jordan, M. I.; and Klein, D. 2010. Learning Programs: A Hierarchical Bayesian Approach. In *Proceedings of the International Conference on Machine Learning*, 639–646. Omnipress.

Menon, A.; Tamuz, O.; Gulwani, S.; Lampson, B.; and Kalai, A. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the International Conference on Machine Learning*, 187–195. PMLR.

Moraes, R. O.; Mariño, J. R. H.; Lelis, L. H. S.; and Nascimento, M. A. 2018. Action Abstractions for Combinatorial Multi-Armed Bandit Tree Search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 74–80. AAAI.

Murali, V.; Chaudhuri, S.; and Jermaine, C. 2017. Bayesian Sketch Learning for Program Synthesis. In *Proceedings International Conference on Learning Representations*. OpenReviews.net.

Ontañón, S. 2017. Combinatorial Multi-armed Bandits for Real-Time Strategy Games. *Journal of Artificial Intelligence Research* 58: 665–702.

Ontañón, S.; Barriga, N. A.; Silva, C. R.; Moraes, R. O.; and Lelis, L. H. 2018. The First MicroRTS Artificial Intelligence Competition. *AI Magazine* 39(1).

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419): 1140–1144. ISSN 0036-8075. doi:10.1126/science.aar6404.

Slavík, P. 1996. A tight analysis of the greedy algorithm for set cover. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, 435?441.

Spronck, P.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2004. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation* 3(1): 45–53.

Stanescu, M.; Barriga, N. A.; Hess, A.; and Buro, M. 2016. Evaluating real-time strategy game states using convolutional neural networks. In *Computational Intelligence and Games, 2016 IEEE Conference on*, 1–7. IEEE.

Van Deursen, A.; Klint, P.; and Visser, J. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35(6): 26–36.

Verma, A.; Le, H. M.; Yue, Y.; and Chaudhuri, S. 2019. Imitation-Projected Programmatic Reinforcement Learning. *arXiv preprint arXiv:1907.05431* .

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the International Conference on Machine Learning*, 5052–5061.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J. P.; Jaderberg, M.; Vezhnevets, A. S.; Leblond, R.; Pohlen, T.; Dalibard, V.; Budden, D.; Sulsky, Y.; Molloy, J.; Paine, T. L.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Wu, Y.; Ring, R.; Yogatama, D.; Wünsch, D.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Kavukcuoglu, K.; Hassabis, D.; Apps, C.; and Silver, D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782): 350–354.

Yang, Z.; and Ontanón, S. 2019. Guiding Monte Carlo Tree Search by Scripts in Real-Time Strategy Games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 100–106.