Artificially and (Hopefully) Intelligently Modeling Program Synthesis: Planning in a Large, Strange State Space

Gregory Yauney

15 April 2015

Contents

1 Abstract

The set of possible programs is incredibly large. As such, the problem of generating programs is incredibly difficult. Current research focuses on generating a program that adheres to a program correctness specification as given by input and output examples, but it does not seek to explicitly understand the state space of possible programs. I present a novel approach to modeling program generation as a search and planning problem. LambdaJS, a reduced core of JavaScript, is modeled textually as a Markov Decision Process, with states corresponding to the text of a potential program and actions corresponding to adding a single character to the end of a state's text. In order to take advantage of the fact that the structure of the programming language should guide the search, the MDP incorporates options which encode the tokens allowed in the language. When this approach did not as successfully guide the search as hoped, I implemented an Earley Parser for LambdaJS, the chart of which has now been added to the MDP's state so that only actions that contribute to predicted terminals can be taken from any particular state. This falls very much in line with existing research on goal-based action priors but is domain-specific instead of goal-specific. Finally, I compare the number of states expanded when reproducing LambdaJS programs through breadth-first search by the basic MDP, the option-enhanced MDP, and the parser-enhanced MDP, showing that the parser-enhanced version is a substantial improvement over the others.

2 Introduction

The set of all possible configurations of text in a buffer is incredibly large. The set of all possible programs is incredibly large. Even if we just limit ourselves to all possible programs written in a single programming language, the set is still incredibly large. And the set of all programs that are syntactically correct is *still* incredibly large. But let's say that we want to explore these spaces anyway because a solution to the problem of synthesizing programs would have a number of applications. As mentioned in the abstract, I have tackled the problem from an artificial intelligence and machine learning perspective, modeling a programming language–a subset of JavaScript in this case, for reasons I go into below–as a Markov Decision Process. This is, at its most basic, quite similar to how a human (or in this case a human without any understanding of programming) might approach the problem. By first modeling all possible textual configurations for a buffer of a given length, I am able to expand that model to search through the set of all possible syntactically correct programs.

Being able to synthesize programs would have enormous benefits. It would be able to help you code, or it would take a description of a program and actually create that program for you. Current state of the art research synthesizes programs from examples of input and output. It does not approach it as an artificial intelligence problem. Modeling program generation as an artificial intelligence problem is similar to the states that programs go through as humans actually program them.

After recounting the related work, I describe each stage of this project and ultimately evaluate how each stage performs in a breadth-first search for particular small programs. This project primarily contributes a new model for program generation as well as the beginning of an evaluation and subsequent justification for that model.

3 Related Work

Program synthesis goes by many names, such as automatic program generation, and is a problem that has indeed had some research done on it, but surprisingly little of that research has cast it as an artificial intelligence problem. The way in which I tackled the problem has, as far as I know, no immediate precedents in the literature and relies on a very disparate background knowledge.

3.1 Syntax-Guided Synthesis

This is the only other approach to automatic program generation I could find. Syntax-Guided Synthesis is the latest in a long line of work called "Programming by Example", which attempts to create a program that accurately transform an input set into an output training set [12]. SyGuS, as the authors call it, generates a program that meets a certain correctness specification, which is provided in the form of a logical expression and in the terms of producing the correct output from given input. After the user specifies input and output examples, their algorithm then uses a provided syntactical template (hence the "syntax-guided" appellation) to construct a loop-free program from the deductive proof of its existence [3]. More in-depth information regarding Programming by Example can be found in the textbook of a similar name and its sequel [4], [8]. I do not delve further into specifics for this approach because it is so different from my own.

3.2 Markov Decision Processes

Markov Decision Processes, the classic reinforcement learning problem formulation, are treated extensively in Chapter 17 of Russell and Norvig, the classic artificial intelligence textbook [11], as well as in Chapter 3 of Sutton and Bartow, the classic reinforcement learning textbook [14]. As a bit of an overview of the necessary aspects, MDPs are are sequential decision processes that model some aspect of the world and adhere to the Markov assumption. There is an agent interacting with an environment. The environment is discretized into states, each encoding a particular configuration of the environment. The agent, then, is said to be in a particular state. They are fully observable and can be stochastic; the result of an action is always known, but it is not necessarily deterministic. If an agent takes an action from a particular state, then the state it ends up in is not always the same as the last time it took that action from that state. Transitions operate under a Markov assumption (hence the name): when transitioning, only the current state influences the next state. In other, more mathematical, words: the transition probability is only conditioned upon the current state. History is irrelevant. A solution to an MDP is a policy, a function which takes in a state and returns the optimal action from that state.

Mathematically, an MDP is the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} >$ [11]. In the following notation, s is the current state, a is the action taken from the current state, and s' is a successor state. $\mathcal{S}$ is the set of possible states, the state space. $\mathcal{A}(s)$ is the set of actions that are available to the agent and is a function of the current state because different actions may be able to be taken from different states. $\mathcal{T}$ is the transition function: $\mathcal{T}(s'|s, a)$, denoting the probability of transitioning to state s' after taking action a from state s. $\mathcal{R}$ is the reward function: $\mathcal{R}(s, a, s')$, denoting the numerical reward received by the agent for transitioning to state s' after taking action a from state s.

3.3 OOMDPs and BURLAP

In an Object-Oriented Markov Decision Process, the states, as the name implies, are collections of what are called objects. Each object has one or more attributes, and states are distinguished by the objects they contain and by these objects' attributes [5].

BURLAP, the Brown-UMBC Reinforcement Learning and Planning Java code library, is–just as the name implies–a large collection of planning and reinforcement learning algorithms that operate on OOMDPs. I programmed this in BURLAP in order to take advantage of its many planners and learning agents that can be trivially applied to new problems [9].

3.4 Options and Affordances

Affordances and Options are extensions to the typical MDP reinforcement learning problem described above. Options, as suggested by their original, less colloquial, name of Temporally-Extended Actions, are an addition to the Actions of an MDP. They are a way of temporally abstracting the MDP: instead of transitioning directly from timestep t to timestep t+1, something can happen in between. In other words, all of the things that happen in between the timesteps are abstracted out into a single option that takes the system from the state at timestep t to the state at timestep t+1. Theoretically, options are “closed-loop policies for taking action over a period of time.” When an agent takes an option, that option's policy is executed and the state ultimately transitions to the state that is reached when the option's policy terminates. The option's policy can be deterministic or non-deterministic. When it is deterministic, the option is called a “macro-action” because it is the same set of actions always executed in the same order. In practice, options can be treated like actions by the MDP [15]. Options are a subclass of actions in BURLAP and thus are implemented as direct extensions of actions [9].

Affordances, or goal-based action priors, prune away actions that are irrelevant for acheiving a certain goal. They can be specified by an expert or learned through prior experience on smaller problems [1]. I do not strictly use the affordance framework as formulated in that paper. Instead, I prune actions from a state with respect to the domain, not with respect to the goal.

3.5 Earley Parsing

The Earley Parser, named after its inventor, is a universal parsing algorithm for parsing any context-free grammar. It is top down, meaning that it begins at the highest level of the parse tree and descends downward to the branches. While slow, with its cubic complexity with regards to the length of the input string, I am interested in it because, due to it's top-down approach, of the way it predicts possible next terminals at each step in the parsing. The following description is based off of my understanding of the algorithm as detailed in [2], [6].

There is an input string in a specific language. An Earley Parser is a state machine, with a set of states associated with each position in the input string. Each state is a dotted rule paired with a start index into the original string. A dotted rule consists of a production from the grammar, and a dot, which is an index into the right hand side of the production. Everything to the left of the dot has already been parsed, and everything to the right still has to be parsed. Dotted rules are important because they allow us to easily describe the parsing process in terms of three processes that each operate on a different type of dotted rule. More terminology: non-terminals and terminals refer to elements of the grammar that can be further replaced by another production in the grammar and those that cannot, respectively.

There are three phases of Earley Parsing: prediction, scanning, completion. Each is performed in that sequence until none of them can be performed any more. If, at the end, the chart contains a completed rule with the start state on the left hand side, then the parser accepts the string. The output of an Earley Parser is called its chart, which can be used to construct a parse tree.

Prediction works on rules where the dot immediately precedes a non-terminal. All productions that have that particular non-terminal on the left hand side and any terminal on the right hand side are added to the state set with a dot position at the very beginning of the right hand side. Rules added through this process tell us what the possible next terminals are.

Scanning is very straightforward: read in the next terminal from the input string and for every state in the set at this index that has a dot immediately before the terminal that was just read, copy that rule over to the next state set and advance the position of the dot by one. Scanning is the phase responsible for actually advancing along the input string.

Completion looks at rules where the dot is at the very right hand side of a singular production, where it is one non-terminal becoming a terminal. It finds states in the previous state set where the dot immediately precedes the non-terminal, copies that rule to the current state set, and advances the dot past the non-terminal.

In this project, parsing is important because it provides what is called syntactic analysis of the programs as they are being generated. It is important to note that this is entirely distinct from whether or not the program can actually be compiled. It merely assures syntactical correctness of the program. If the program fails to run, then there is a problem with something else. I am interested in using an Earley Parser to inform planning by limiting my MDP to only actions that preserve syntactical correctness of the generated program.

3.6 λ_{JS}

Why on earth did I choose to model JavaScript, a notoriously convoluted 258-page specification? I am in fact instead modeling λ_{JS} (pronounced "LambdaJS", a condensed alternative to full-blown JavaScript, produced right here at Brown in the programming languages group. λ_{JS} is a self-contained core within the very messy language that is JavaScript, so condensed that it's operational semantics fits on a comparatively mere three pages. By way of a reduction semantics, nearly all programs in Javascript can be converted to λ_{JS} programs. λ_{JS} has been shown to be completely equivalent to JavaScript in full [7].

But still, why JavaScript? The choice of programming language is ultimately to some extent arbitrary. The benefits of using λ_{JS} are twofold. Beyond the cool-factor of being able to say that I am working on JavaScript, a widely used programming language, using λ_{JS} allows me to mine training examples from existing JavaScript code in a way that using a less-widely adopted language would make difficult while still providing the strong backbone of a simpler language that has been rigorously studied.

4 Approach

When this project began, I had only the vaguest of roadmaps for how to tackle this problem. Therefore, I think it is important to describe the different stages of this project as it progressed and changed.

4.1 MDP formalization of λ_{JS}

I chose to textually model λ_{JS} at the level of individual keystrokes. The analogy here is a person typing on a keyboard, pressing one key at a time, into a text editor. At this basic level, the structure of the programming language is not taken into account; any sort of nonsense can be generated in this model because it has no knowledge of what qualifies as a correct program.

The state, then, is very simply whatever text has been generated so far. The initial state is empty of text.

Any given action adds a single character to the end of the current state's text buffer. There are as many actions as there are allowable characters, where each action corresponds to adding one, and only one, character.

Transitions are entirely deterministic in this MDP. The transition model, then is very simple: $p(s'|s,a) = 1$ where s is the current text buffer, a is adding a single letter to the text buffer, and s' is the resulting state with a text buffer that is identical to that of s but with a's letter appended. When a and the one-character difference between s and s' do not match, the probability of a transition from s to s' given a is zero. States transition one character at a time, deterministically.

The reward function is slightly more difficult to define. First of all, what is our goal? Let's say that we want to look at a single program, like `return false;`. Several different measures could be used. Does the output of our program match the output of the sample program? In this case, that would mean any MDP state that, when run, spits out `false` would be considered a goal state. This is more similar to how Programming By Example operates. Or does the MDP's state exactly match the text of the sample program? Here only one state could possibly be the goal state, and in the above example, that state is the one that says `return false;`. I have chosen to use this metric because the training dataset is then much simpler. Instead of requiring texts of programs and their corresponding output, we can just use the texts of the programs. Because of this, we know exactly which program we've found once we've reached a goal state. In the above example, we would know that we would have found `return false;` as opposed to any number of longer, more complex programs that also output false, like `if (3 > 2)  return false;` or `var t = 3; return false;`.

To go back to the mathematical formulation of MDPs, $\mathcal{S}$ is the set of all possible arrangements of text in a buffer. This is a theoretically infinite state space because the MDP is not restricted to buffers of a certain size. $\mathcal{A}$ is the set of all characters and is uniform across all states. $\mathcal{T}(s'|s,a) = 1$ when s and s' differ only by the character that a adds and is equal to zero otherwise. $\mathcal{R}$ is positive for the goal state and zero for all others.

At the level of implementation, BURLAP deals with OOMDPs. There is one text buffer object in the state, and this text buffer has a string attribute that contains the text as it has so far been generated. Thus the OO part of the OOMDP is rather simple; there is only one object, and everything interesting happens in the manipulation of that buffer object's string attribute.

4.2 Options

It would take an incredibly long time to search through the space of possible programs at the rate of one single character per action, even if we're only trying to find a very small program. Options were the first extension I added to this simple MDP framework in order to take advantage of at least a small part of the structure of λ_{JS}. There are several different ways to do this.

The first options I added might make more sense to think of as macro-actions, which are a family of options. Macro-actions are deterministic and are have the same effect as many actions applied all at once [15]. These are not fancy options with non-deterministic policies all their own. The macro-actions I created correspond to adding an entire reserved keyword from λ_{JS} to the end of the state. In the case of the keyword "return", a macro-action of this type would add to the MDP's state the entirety of the keyword as a single action instead of individually having to take the r, the e, the t, the u, the r, and then finally the n actions in a row. How convenient!

As an example, think about the very short program `return 0;`. Without these options, the minimum number of actions to get to the goal is nine: one for each character. With these actions, the minumum number of actions to get to the goal is 4: one for the return token and one each for the three remaining characters.

I also added options that correspond to the context-free grammar of λ_{JS}. Although this would have encoded even more Options in BURLAP, at least at the time I implemented these options, had to be entirely defined. This means that if this approach were to work out, I would have had to roll out options until expanding one all the way to the full program.

Options are treated just like actions. A planner is at a state particular state in the MDP and it can either take an action or an option. But it is important to note the inherent tradeoff with options. Options on the one hand reduce the total number of actions needed to reach the goal, but on the other they increase the total number of possible actions from a state. Without a way to intelligently prune options, a planner might take longer to run with options than without, depending on the branching factor.

The formalization is the same as in the previous basic version of the MDP, but $\mathcal{A}$ has been extended to include the options because they are treated just like regular actions. $\mathcal{T}$ and $\mathcal{R}$ have been similarly expanded but are fundamentally the same. $\mathcal{S}$ is still just configurations of text in a buffer.

I ultimately threw out the context-free grammar options because of the problems detailed above, but the reserved keyword options have remained.

4.3 Earley Parser

What good could there be in adding an Earley Parser to the MDP's state? Well, what information does an Earley Parser provide? In addition to providing a parse tree for a given string, the parser fills in its chart. Remember the three phases of Earley Parsing: prediction, scanning, and completion. The form of dotted rules generated by the prediction phase is a dot followed by a terminal. Looking at all of the dotted rules generated by the prediction phase after the most recent scanning phase and extracting the terminal that immediately follows the dot's position gives us the set of possible terminals that can come next. In other words, all of the terminals that have been predicted since the last terminal was added are the only possible syntactically correct terminals that can be appended to this state. This means that any action that adds at least part of a syntactically-allowed terminal should be chosen in order to maintain syntactical correctness. All other actions can be pruned from this state so that only actions that lead to syntactical correctness are considered.

This is very similar to recent work on goal-aware action priors. The difference here, as far as I can tell, is that this action-pruning is not dependent on the goal but is instead dependent on the domain in general [1]. That is, the function that governs whether an action or an option can be taken is . (insert piecewise binary function)

Further implementing this parser within BURLAP proved to be immensely frustrating. States in BURLAP are uniquely identified by the objects in them and by the attributes of those objects. Java objects are not the same as these abstract BURLAP objects. This meant entirely rewriting my parser, trying to break it apart into pieces that would fit inside BURLAP's integer and string attributes. The chart is really what matters here since, in the form of rules that have been generated by prediction phases of the parser, . I was, of course. So what did I do? I smushed a GPL Early Parser into BURLAP. Whenever a new state is reached by a planner, the parser is run on that state's text buffer and the parser returns the list of terminals that were generated during any of the parser's prediction steps. This workaround makes it so that, as long as we ignore time complexity, the parser's chart has essentially been embedded inside the MDP's state.

This represents a significant change to the MDP formalization. $\mathcal{S}$ must now be expanded to include the Earley Parser's chart, which is a list of dotted rules, i.e. indices into a grammar and indices into rules of that grammar, and indices into the input string. $\mathcal{A}(s)$ now depends highly on s because different states will syntactically allow different succeeding states according to the grammar of the language. The set of available actions is only those that will contribute. $\mathcal{T}$ and $\mathcal{R}$ remain essentially unchanged. All of the work has gone into changing the state space so that the action space is now variable(wording?).

5 Results and Evaluation

The extensions I made after defining the basic MDP have dealt with increasing and decreasing the available actions at any particular state. Therefore, I am interested in how many states are expanded when searching for a specific program. As a means of comparing the project at each of its three stages, I examine the differences in the number of states expanded during breadth first search. The goal here is to reproduce a λ_{JS} program character-for-character, so the goal state is reached when the planner has found a state with a text buffer string attribute that exactly matches the program it is trying to reproduce.

I compare three different stages of this project. The first stage is: the initial MDP with only the single-character action set; the second is: the MDP with the token options; and the third is: the MDP with the the parser additionally embedded in the state.

As the chart and the graphs show, the parser-enhanced MDP expands far, far fewer nodes than the MDPs from the first two stages of the project and can even find programs that the first two have no hope of finding in a reasonable amount of time. Estimates were used for the basic MDP on the medium and

Stage	Small	Medium	Large
Basic	1265233	1.4×10^{23}	1.9×10^{46}
Options	15822	20478	9.7×10^{23}
Parser	38	17	7818

Table 1: A comparison of the number of nodes expanded in a breadth-first search by all three stages of the project in reproducing a small, a medium, and a large program. All entries with scientific notation are estimates.

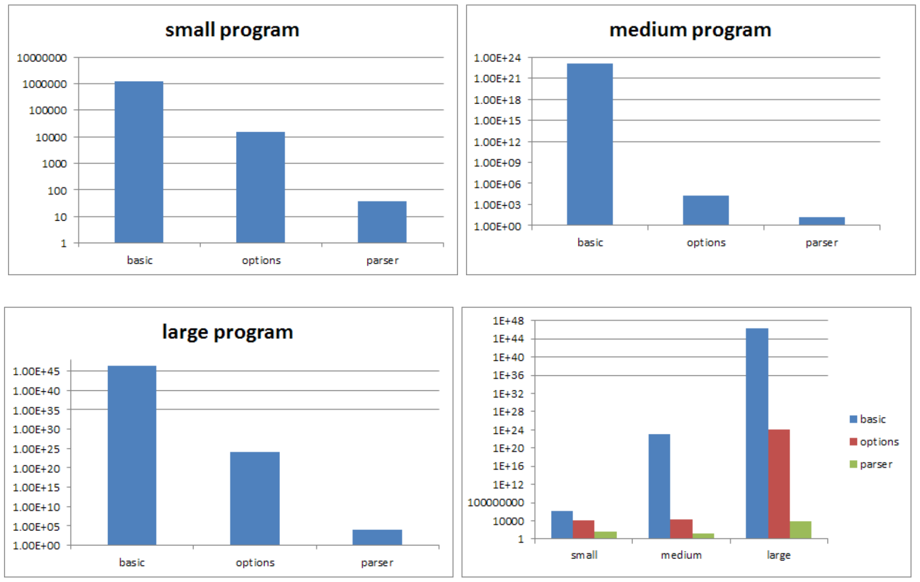

Figure 1: The same data in chart form. Note the logarithmic vertical axis.

large programs and for the options-enhanced MDP on the large programs . The small program was `i++;`, the medium program was `return false;`, and the large program was `if (x == false) return true;`. Of course, the labels are relative because all of these are just incredibly small snippets of programs, but no comparison can be made between all three versions with larger programs.

These results tell us, at least for these small programs, that in all cases, the parser MDP searches through a much smaller state space than the other two. How can this be if they all have the same underlying domain? The number of available actions greatly determines just how many states are reachable from any state. From the start state, the only possible choices for the parser MDP are characters that are the beginning of something syntactically correct, where as the options MDP allows characters that begin any reserved keyword regardless of whether the placement of that word could lead to a correct program , and the basic MDP allows any character to be chosen. This results, when carried through to each character position, in a much smaller set of text to search through for the parser MDP.

This simple uninformed search in fact tells us a lot about the nature of each of the state spaces. It should be noted that breadth first search is sufficient to do this because each of the basic actions, that of adding a character or token to the end of the text buffer, has the same cost. The reason I use breadth-first search is to try to understand the size of each of these state spaces, and it is possible to conduct such a search on the MDP at each stage of the project. Reducing the number of nodes expanded has been the goal at each stage of the project. In other words: the goal has been to craft more intelligent state spaces for the problem, ones that limit the search through text only to states that are more likely to be (or in this case capable of being in the first place) the goal state. Because breadth-first search always expands the shallowest search node, the number of nodes expanded while breadth-first searching is a rough measure of the branching factor for each state space [11]. But only roughly: this can be seen in a comparison between the basic MDP and the second MDP with token options. While each can eventually find the same programs, the options MDP expands fewer nodes (as long as it can use some of its options) even though it has a larger branching factor because its options result in a search tree with a smaller depth.

The numbers of nodes expanded can be better understood by mathematically examining the branching factors of each of the three MDPs. In each, the branching factor is the number of successors each node can generate, as determined by the number of actions that can be taken from that node. The total number of nodes generated is on the order of b^d, where b is the branching factor and d is the depth of the search tree, which in this case is the number of characters in the program that is being searched for. In the case of the basic MDP, each node always has the same number of actions, which is the number of possible characters. For the MDP with options, the branching factor is even larger because of all the newly available options in addition to all of the same actions as the first MDP. The MDP with the parser has the lowest branching factor of all three stages because the parser prunes away actions and options that do not preserve syntactical correctness. This one's factor is much more difficult to calculate because it varies from state to state. This mathematical understanding was used to provide the estimates in the above figures.

The number of nodes expanded is also an indicator of how well-suited each state space is for the problem. The intuition goes that the less the agent knows about the structure of the state space, the more nodes it will have to expand. A smaller branching factor indicates that the state space is more suited to the problem we are trying to solve. The first MDP, when it searches through all configurations of text instead of just programs, proves to be too general; it does not have as much information about the problem as the other two MDPs. That is, the search through the same underlying state space is better guided by the parser's action pruning.

More results in a similar vein are forthcoming in the final *final* draft.

6 Conclusions

6.1 Limitations and Future Work

While much more impressive than the initial baseline implementation, there are of course many limitations to my final implementation. This project can branch off in many different directions, and the current limitations can and should motivate which direction it goes in.

First of all, it requires knowledge of the underlying context-free grammar of the language. The parser MDP only does so well because I have provided it with so much information in the form of the λ_{JS} grammar. It would be entirely possible to use this framework for learning a grammar that governs the syntax of a set of training examples that could work by modeling a probabalistic parser instead and learn a probabalistic context-free grammar [13], [10].

The major limitation to my approach right now is that it still searches through all possible syntactically correct programs. This is an incredible improvement, as shown in the above section, over searching through the set of all possible configurations of text without regard to syntactical correctness, to be sure, but this is still not as good as it could be.

There are a couple ways to approach this problem. One would be to work with the MDP as it is now defined and to try to use reinforcement learning algorithms to find optimal policies for generating programs. Q-Learning, R-Max, and even policy gradient could be used to learn the optimal policies.

But that does not address the still incredibly large state space. Another direction would be to add in subgoals, which use many different smaller goal states as opposed to one final goal state. Instead of the single goal test which is only succesfully met once the entire program has been generated, programs would be generated in a scaffolded manner. Adding these in conjunction with affordances, which, as their other name of goal-based action priors implies, are with respect to a goal state would significantly reduce the number of states expanded. [1] The tricky part is determining what good subgoals are. Would it be better to generate the first half of a program or to generate a basic program that works and then go back and add features one at a time? I do not have an answer to this conundrum yet.

Any further work will, I think, require a more robust reformulation of the goal state. Right now, since the goal condition is the simple check of whether or not the generated program matches, character for character, there is no notion of a correctness specification that has proven so useful in other work. Natural language descriptions of programs are also a pipedream. What I have done is an important first step toward understanding this very complex problem, but it is not enough.

6.2 Uniqueness of Approach

I want to emphasize just how different my approach is from existing program synthesis techniques. By modeling a programming language as an MDP, I am able to treat program generation as a search and planning problem in artificial intelligence. The significance of this project lies, I strongly believe, in its novel approach. The MDPs from different stages of the project have enabled an exploration of how the search through the textual space of programs can be guided and refined.

6.3 Last Words

Trying to tackle program synthesis as a reinforcement learning problem is in the end–at or at least in *this* particular end–both simpler and more complex than I had initially anticipated.

It is much more complex in all sorts of ways than I initially anticipated. As discussed in the above limitations section, the state space of the parser-enhanced MDP is still so enormous that it takes incredibly long to reproduce anything but the smallest of programs.

But on the other hand, the way I have framed the problem might be throwing too much power in the wrong places. The goal of model-based reinforcement learning algorithms is to estimate the transition and reward functions. Since this MDP's transition function is entirely deterministic, there is very little to learn with Q-Learning and perhaps even other reinforcement learning approaches.

One final question remains: if programs can be found by searching through an MDP, are programmers now out of a job? Don't worry. This system is indeed pretty artificial but certainly not (at least not *yet*) very intelligent.

7 Acknowledgments

I would very much like to thank Prof. Stefanie '`stefie10`' Tellex for the initial idea, for providing guidance and encouragement throughout each phase of this project, for funding not just a summer research project but also a winter research project, and for introducing me to the field of AI in the first place.

This project could not have come close to happening without the help of Dr. James McGlashan, the BURLAP wizard himself, who put up with all sort of questions at all sorts of odd times.

Prof. Tom Doeppner is the best adviser a student could ask for. I've never seen such fast email response times or such beautiful underwater fish photos. I've also never received such academic kindness.

Thank you thank you also to Prof. Michael Littman for kindly agreeing to read this project at nearly the last minute.

What's the point of an acknowledgments section if you don't shout out to all your friends? Big thanks go out to Ellis for being a phenomenal partner in research, gym, living, ratty, crime, &c; Anthony for always being willing to talk about more exciting lit'rary matters; Grace for incessant kindness and friendship even as this project stressed me out more and more; Dave for occasional help and constant friendliness; Natasha for late-night conversations; Helen for intense pep talks and for spending so much time around me as I feverishly coded and and and; Jason for talking to me about this project's (and everything else's) criticotheoretical implications over delicious Geoff's sandwiches; Angela for listening with an outsider's ear and for being a much more motivating study buddy to me than I am to her; Leigh for ensuring that no fellow humans were killed or subjugated or otherwise harmed by the robots; Nick, Will, and Michael for putting up with me as year-long roommates; Mahalia and Ardra for being the sunniest possible summer and winter housemates, respectively; my family, of course; and (altogether) for being there.

`gyauney`
St. Johnsville, NY
Providence, RI

References

[1] David Abel, D Ellis Hershkowitz, Gabriel Barth-Maron, Stephen Brawner, Kevin O'Farrell, James MacGlashan, and Stefanie Tellex. Goal-based action priors. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling*, 2015.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–17. IEEE, 2013.

[4] Allan Cypher, Daniel C. Halbert, David Kurlander, Harry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* The MIT Press, Cambridge, MA, 1993.

[5] Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 240–247, New York, NY, USA, 2008. ACM.

[6] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[7] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *ECOOP 2010–Object-Oriented Programming*, pages 126–150. Springer, 2010.

[8] Henry Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software: Programming by Example.* Morgan Kaufmann, San Francisco, CA, 2000.

[9] James MacGlashan. Burlap: Brown-umbc reinforcement learning and planning. `http://burlap.cs.brown.edu`. Accessed: 2015-4-12.

[10] Brian Roark. Probabilistic top-down parsing and language modeling. *Computational linguistics*, 27(2):249–276, 2001.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Third Edition.* Pearson Education, Inc., Upper Saddle River, NJ, 2010.

[12] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example, 2015.

[13] Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational linguistics*, 21(2):165–201, 1995.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

[15] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.