Learning compositional programs with arguments and sampling

Anonymous Author(s)
Affiliation
Address
email

Abstract

One of the most challenging goals in designing intelligent systems is empowering them with the ability to synthesize programs from data. Namely, given specific requirements in the form of input/output pairs, the goal is to train a machine learning model to discover a program that satisfies those requirements. A recent class of methods exploits combinatorial search procedures and deep learning to learn compositional programs. However, they usually generate only toy programs using a domain-specific language that does not provide any high-level feature, such as function arguments, which reduces their applicability in real-world settings. We extend upon a state of the art model, AlphaNPI, by learning to generate functions that can accept arguments. This improvement will enable us to move closer to real computer programs. We showcase the potential of our approach by learning the Quicksort algorithm, showing how the ability to deal with arguments is crucial for learning and generalization.

1 Introduction

The ability to autonomously synthesize programs from data is one of the main goals in developing intelligent systems. Namely, given specific requirements, such as input/output pairs or more formal specifications, the model must learn a program that meets those demands. Several approaches to try to solve this task are based on the neural controller - interface framework, in which a neural network interacts with an external structured environment by using primitives, such as read/write heads or other atomic functions [1, 2, 3, 4, 5, 6]. Neural Turing Machines [3], and Differentiable Neural Computers [4] can learn simple procedures by training on input/output pairs. These models use external memory, and they are differentiable, which means that they can be trained end-to-end by gradient descent. However, they are essentially black boxes since it is non-trivial to inspect them to understand their discovered algorithm. More importantly, supervision is provided at the level of full program outputs only, making training strongly susceptible to overfitting as shown for the priority sort task of [3].

Neural-Programmer interpreters (NPI) [2] use a reinforcement learning paradigm to learn highly compositional programs by exploiting execution traces. An execution trace is an ordered list of commands that, if executed, solve a given task. Execution traces provide finer-grain supervision during training, encouraging convergence towards more accurate programs. NPI assumes to have at hand a large dataset of execution traces for the given problem, substantially limiting its applicability in real-world scenarios, where an oracle capable of providing such a dataset could be unavailable. A recent model, named AlphaNPI, shows how to train NPI without execution traces. AlphaNPI can solve both discrete and continuous control problems ([5, 6]), and it works by combining NPI with Monte Carlo Tree Search (MCTS), which is used to discover the execution traces. This approach showed outstanding results, both in terms of generalization and interpretability of the final models.

Submitted to 35th Conference on Neural Information Processing Systems (aiplans 2021). Do not distribute.

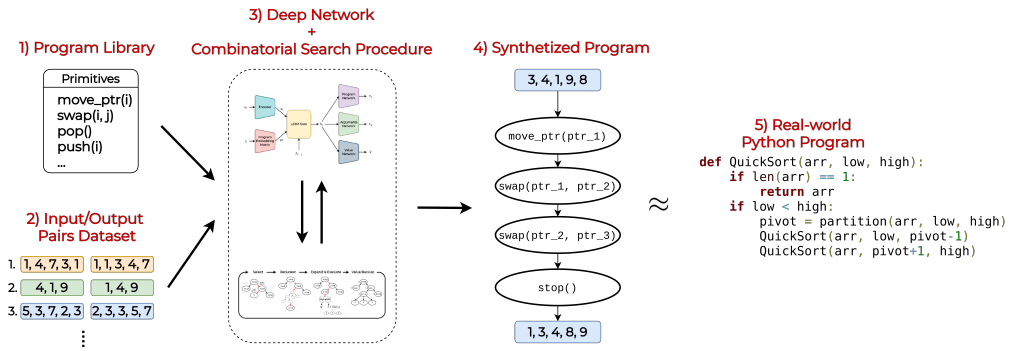

Figure 1: Program Synthesis Pipeline. The final model is able to synthetize a program that resembles a real-world algorithm.

However, the spectrum of tasks it can learn is still somewhat limited since it generates programs that use an ad-hoc language that does not present any high-level construct of a formal programming language such as loops, variables or function arguments. This work presents some promising results towards generalized neural interpreters and discusses additional future challenges that still need to be addressed. The main contribution is the following: *we propose a new version of the AlphaNPI model, which brings us closer to learning real-world programs by learning functions that also accept arguments.*

2 Problem Statement and Original Setting

We consider the problem of learning a complex algorithm, such as Quicksort, by having an agent interacting with an environment e by choosing actions p. The initial actions are called *atomic actions*. In the Quicksort case, the environment is a list of integers, and the atomic actions are element swaps and one-step pointer moves. The agent learns to leverage and to combine these atomic actions to produce higher-level programs. These learnt programs are then added to the collection of available actions. Therefore, the agent can benefit from both atomic and more complex actions to produce advanced algorithmic behaviour. Each action has assigned a *level*. The level 0 actions are the atomic actions that do not need to be learned. Discovered programs have positive levels. Each program can only call lower-level action or lower-level learned programs. In addition, each program and atomic action has associated *pre-conditions*, in the form of boolean constraints, conditioned on the environment state. Pre-conditions ensures we are consistently generating programs that are correct and feasible. For instance, the pre-condition of the Quicksort program is that all pointers must be at the beginning of the list. Pre-conditions are not learned, but they are provided beforehand. The objective is to learn a library of programs organized into a hierarchy. In our case, each task corresponds to learning a single program (e.g., the `partition` function of Quicksort). The reward function returns 1 if the program is correct, 0 otherwise. The goal of the agent is to maximize the expected reward for all the tasks. This goal configures as a multi-task reinforcement learning and as a discrete search problem. The search space is sparse since there exist many possible programs but very few viable ones. It also makes the reward function sparse since we get a positive reward and thus learn if and only if we obtain the correct program.

3 Novel Setting

We change the original setting by making actions and programs accept *arguments*, a. Each action/program can take an ordered list of at most three arguments. In our environment, these arguments are the pointers used to manipulate the list. For example, the `quicksort` program accepts two arguments, the pointer placed at the beginning of the list and the pointer placed at the end of the list. Additionally, an action/program can also accept no arguments, such as the `stop` action, which terminates the execution of the simulation. As for the pre-conditions, the maximum number of arguments that a program can accept is given apriori. We also extend the pre-conditions to consider the arguments to decide if a program is feasible. For example, the `swap` pointer action cannot be called by giving as argument the same pointer twice.

3.1 Architecture

We start from the architecture developed by [5], and adapt it to the novel setting by introducing an additional module, $f_{arguments}$. Thanks to this new module, we learn compact programs that can adapt their behaviour by accepting arguments instead of learning a separate program for each possible value of the arguments. For example, we can learn a `move` action to shift pointers that can operate on multiple pointers at the same time, `move(pointer_1,pointer_3)`. With the original method, we would have had to learn two separate actions, `move_pointer_1` and `move_pointer_3`.

The architecture consists of six modules: an encoder (f_{enc}), a programs matrix (M_{prog}), an LSTM core (f_{lstm}), a program network ($f_{program}$), an arguments network ($f_{arguments}$) and a value network (f_{value}). The encoder takes as input an environment e_t and encodes it as a set of features s_t. The program matrix takes as input an index i_t and returns the corresponding program embedding p_t. The LSTM core executes programs while conditioning on p_t, the feature set s_t and its internal state h_{t-1}. The program and arguments networks take the LSTM output, h_t, and return probabilities over the action space, π_p, and argument space, π_a, respectively. Lastly, the value network takes instead the LSTM output to estimate the value function V. The equation of the architecture are the following:

$$s_t = f_{enc}(e_t) \qquad p_t = M_{prog}[i_t] \qquad h_t = f_{lstm}(s_t, p_t, h_{t-1}) \tag{1}$$

$$\pi_p = f_{program}(h_t) \quad \pi_a = f_{arguments}(h_t) \qquad V = f_{value}(h_t) \tag{2}$$

4 Training

During a training iteration, the agent selects a program p_t to learn. It executes n_{ep} episodes by exploring the search space with MCTS. The data gathered during the episodes are aggregated to construct the tree policy vectors for both programs and arguments, π_p^{mcts} and π_a^{mcts}. After each successful episode, we collect the observations, the hidden states, the task indexes, the rewards, π_p^{mcts} and π_a^{mcts} into an execution trace. More formally, an execution trace for a given program p_t is a tuple $(e_I, p_t, h_t, e_O, r_t, \pi_p^{mcts}, \pi_a^{mcts})$. An execution trace gives the exact sequence of actions that, if applied to the input e_I, produce the output e_O. The final discovered execution traces are stored in a replay buffer. Then, a mini-batch is sampled from the replay buffer, and the agent is trained on this data to minimize this adjusted loss function:

$$l = \sum_{batch} \underbrace{-(\pi_p^{mcts})^T \log \pi_p}_{l_{policy}} \underbrace{-(\pi_a^{mcts})^T \log \pi_a}_{l_{arguments}} + \underbrace{(V - r)^2}_{l_{value}} \tag{3}$$

We want to push the network to reproduce the execution traces found by jointly minimizing the cross-entropy between the policies discovered by MCTS and the policies generated by the network. l_{policy} minimizes the cross-entropy between the program policies, π_p^{mcts} and π_p. $l_{arguments}$ minimizes the same cross-entropy but between the two arguments policies, π_a^{mcts} and π_a. Instead, l_{value} pushes the network to generate the correct value function V. We also employ curriculum learning [7] to focus on learning programs by following the hierarchy, such as to learn lower-level programs first. Moreover, curriculum learning ensures choosing the next program to learn by looking at the success rate of the single programs. This rule means that programs that fail often are picked more often for learning. Given the environment state, the agent emits a program policy π_p and an argument policy π_a. The policies are such that $\sum_M \pi(i) = 1$, where M is the total number of available programs or arguments. The best program/arguments pair (p^*, a^*) is given by $p^* = \arg\max(\pi_p)$ and $a^* = \arg\max(\pi_a)$.

5 Experiments

We tested our approach on a sorting task. It is a familiar setting used by many others [2, 5, 3]. We focused on learning the hierarchy of programs needed to perform the Quicksort algorithm. Unlike the Bubblesort algorithm used in previous works [5], Quicksort is more refined and it cannot be easily written without functions that can accept arguments. Therefore, we choose it as the target for our experiments. We consider an environment composed of three main components: a list of n integers, three pointers that can reference the list's elements and a stack that can save and retrieve the pointers' current positions. The programs can accept at most three arguments. Each argument

Table 1: Generalization accuracy of the learned programs by varying the length of the list. Each model was tested on 50 different random lists for each given length. The accuracy gives the fraction of lists correctly sorted. The bold values indicate the best results for that given list length.

Program	**List Length**	$M_{alphanpi}$	$M^s_{alphanpi}$	$M_{alphaargs}$	$M^s_{alphaargs}$
partition_update	5	**1.00**	**1.00**	**1.00**	0.96
	10	**1.00**	**1.00**	**1.00**	0.98
	20	**1.00**	0.98	**1.00**	0.96
	40	**1.00**	**1.00**	**1.00**	0.94
	60	**1.00**	**1.00**	**1.00**	0.96
partition	5	**1.00**	0.98	**1.00**	0.96
	10	**1.00**	0.78	0.98	0.78
	20	**1.00**	0.80	**1.00**	0.70
	40	**1.00**	0.80	**1.00**	0.82
	60	**1.00**	0.76	0.98	0.80
quicksort_update	5	0.72	0.58	**1.00**	0.60
	10	0.64	0.42	**0.96**	0.48
	20	0.56	0.34	**0.90**	0.56
	40	0.36	0.30	**0.98**	0.40
	60	0.38	0.26	**0.94**	0.44
quicksort	5	0.10	0.02	**1.00**	0.06
	10	0	0	**0.66**	0
	20	0	0	**0.42**	0
	40	0	0	**0.22**	0
	60	0	0	**0.02**	0

represents a reference to one of the three pointers available in the environment. We also added an empty value to signal arguments that the given function has to ignore. Additionally, as in regular programming languages, the arguments order count (e.g., *function(a,b,c)* has a different meaning from *function(b,c,a)*). We trained the architecture with lists from 2 to 7 elements. The list's elements were integers, and we constrained them between 0 and 10. We trained four different models. $M_{alphanpi}$ is the baseline, and it uses the vanilla architecture from [5]. $M^s_{alphanpi}$ is again a baseline model which uses an approximate MCTS instead (A-MCTS). More details about A-MCTS can be found in the Appendix. $M_{alphaargs}$ and $M^s_{alphaargs}$ employ our custom architecture presented in the previous sections. The former uses the standard MCTS, and the latter uses A-MCTS instead. The higher-level programs that compose our hierarchy and which will be learned are: `partition_update`, `partition`, `quicksort_update` and `quicksort`. They correspond to certain pieces of the original Quicksort algorithm. The complete lists and descriptions of learnable programs and atomic actions can be found in the Appendix. The $M_{alphanpi}$ and $M^s_{alphanpi}$ models learn functions that do not accept arguments. Therefore, we provided them with an augmented set of atomic actions, which we built by taking the cartesian product between the atomic actions and the possible arguments they can accept. As further evaluation studies, we validated the trained models on lists of up to 60 elements to investigate the generalization capabilities. Table 1 presents the validation results for all the learnable programs. From the results, the $M_{alphaargs}$ is the only model which can learn the Quicksort algorithm correctly, whereas the other models fail. It also shows some generalization capacities since it can sort lists with a different length than those used for training. We can also see how the various models struggled when learning the `quicksort_update` function.

6 Conclusion

This work presents some initial results to improve program synthesis by generating more human-like programs starting from input/output examples. We propose an improved AlphaNPI architecture to learn programs that can accept arguments to diversify their behaviour. The support for arguments enables our revised architecture to produce richer programs and learn more complex algorithms with respect to the original work. It also generates code that is more similar to the one produced with current high-level programming languages. We benchmarked our method by learning a well-known sorting algorithm, Quicksort, showing how our approach manages to learn it effectively while AlphaNPI fails to converge. We also show how the learned program generalizes when sorting lists of increasing length. Many open issues are still to be addressed. For instance, in this work only atomic actions can accept arguments, whereas higher-level programs can accept only the "empty" argument. This behaviour happens because all methods operate in the same environment, and there is no proper program scope. Additional studies are required to add this notion of program space.

References

[1] Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 421–429, New York, New York, USA, 20–22 Jun 2016. PMLR.

[2] Scott Reed and Nando de Freitas. Neural programmer-interpreters, 2016.

[3] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[4] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016.

[5] Thomas Pierrot, Guillaume Ligner, Scott Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning, 2019.

[6] Thomas Pierrot, Nicolas Perrin, Feryal Behbahani, Alexandre Laterre, Olivier Sigaud, Karim Beguir, and Nando de Freitas. Learning compositional neural programs for continuous control, 2020.

[7] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 166–175. PMLR, 06–11 Aug 2017.

[8] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, page 72–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[10] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[11] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 2021. (Accelerated article preview).

[12] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: A review of recent modifications and applications. *CoRR*, abs/2103.04931, 2021.

[13] Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Remi Munos. Monte-Carlo tree search as regularized policy optimization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3769–3778. PMLR, 13–18 Jul 2020.

[14] Chenjun Xiao, Jincheng Mei, and Martin Müller. Memory-augmented monte carlo tree search. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

A Appendix

A.1 Code for the experiments

The anonymized repository with the code used for the experiments can be found at this link `https://anonymous.4open.science/r/learning_programs_with_arguments-AIPLANS-2021/README.md`. The code is released to ensure better peer review and reproducibility.

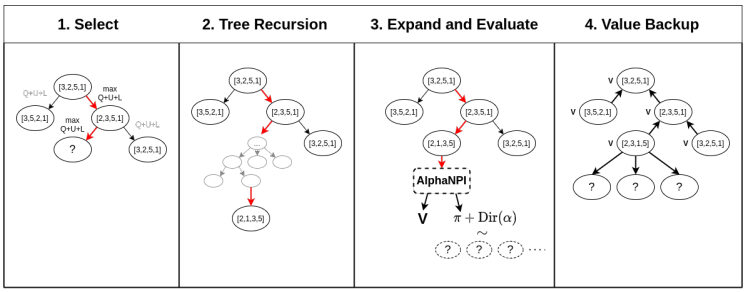

Figure 2: 1. Each simulation explores the tree by selecting those actions that maximize a target objective. 2. If we find a new node that is not an atomic function, we run A-MCTS recursively by resetting to zero the LSTM state. 3. Given the expanded leaf node, we compute the program and actions policies (π_p^{mcts} and π_a^{mcts}). We sample from the two policies to add n new child nodes ($n << M$ where M is the total possible child nodes). 4. We return the value V, and we propagate it back in the tree.

A.2 Re-train over failed environments

We noticed that a small fraction of training environments are substantially harder to solve and are thus mostly neglected when optimizing average performance during training. The result is that sub-optimal programs are learned, and the performance degradation becomes apparent when programs are evaluated on the longer lists used for testing. To tackle this problem and improve robustness and generalization, we implemented a function $g(p_i)$ to re-train over previously failed environments.

Namely, given a task T_i to learn, if we get a 0 reward, we will store the used environment e_I into a buffer called e_{failed}. Then, if we happen to try to learn the task T_i again, we will run A-MCTS by sampling with a small probability ϵ an environment from e_{failed} in which the model had failed to solve in previous iterations. In the other case, we sample the next state from a normal distribution over the environment conditioned on the task T.

$$g(p_i) = \begin{cases} e_I \sim e_{failed} & \epsilon \\ e_I \sim \mathcal{N} & 1-\epsilon \end{cases} \tag{4}$$

The buffer e_{failed} implements some curriculum learning since, given task T_i, if we use previously failed states, we will sample the failed states by looking at the success rate. If a state fails more often than others, then it will be sampled more frequently.

A.3 Approximate Monte Carlo Tree Search

Monte Carlo Tree Search [8, 9] is an algorithm to search large combinatorial spaces represented by trees efficiently. It was successfully used to solve many tasks, such as mastering complex games and protein folding [10, 11]. The original method is made by four phases: *selection*, *recursion*, *expansion* and *backup*. Many different flavours of MCTS are available (for instance, [12, 13, 14]). In this work, we build upon the recursive MCTS developed by [5]. We propose an Approximate MCTS (A-MCTS) by extending the *expansion* phase not to expand all the possible nodes available. This improvement is needed to account for larger search spaces that would be prohibitive to explore thoroughly. In our setting, we have discrete arguments, and the original MCTS requires creating a node for each program/argument pair available. If we increase the number of available arguments, the number of program/argument combinations grows significantly.

In our setting, each node of the tree represents the environment's state at time *t* and each transition represents a *program call*. Figure 2 shows a description of the A-MCTS algorithm. Given the policies π_p^{mcts} and π_a^{mcts}, we add Dirichlet noise to foster exploration and we sample n program/arguments pairs, (p_i, a_i), (n is a hyperparameter of the model) and we add them as possible future available actions. The simulation phase will be shorter with fewer nodes. By exploring fewer nodes, we reward immediately good solutions. Thanks to the random perturbations, we will eventually explore all the available configurations if good solutions are sparse.

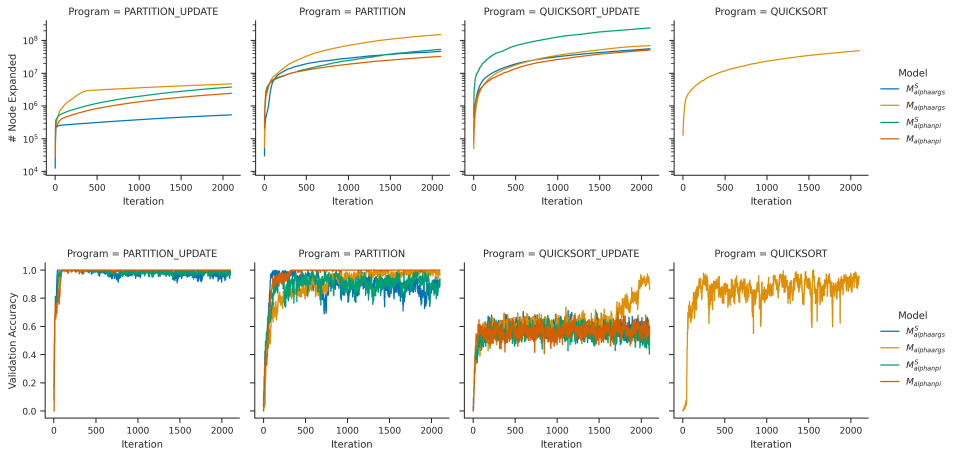

Figure 3: Total node expanded by models (top) and the corresponding accuracy (bottom) while training. The node count is expressed with the log scale. In some cases, the Approximate MTCS shows to enable convergence by exploring fewer nodes. Moreover, once the models learn a program, the node expansion slows down. Interestingly, it seems that the `quicksort_update` function is harder to learn. Unfortunately, only one model is able to learn the complete Quicksort procedure.

A.4 Evaluation of the Approximate Monte Carlo Tree Search

While training, we also recorded the total number of nodes expanded by the MCTS and A-MCTS procedures after each training step. The idea is to check if we can converge to a solution by doing an approximate exploration of the search space by expanding fewer nodes.

Interestingly, the models trained with the approximate procedure A-MTCS show similar convergence rates to those trained with MCTS. They also show good generalization performances, albeit not reaching the level of the original MCTS. From Figure 3, we can look at the total nodes expanded by both the approximate and exact procedures. For specific programs, such as `partition_update` or `partition`, the A-MCTS can converge to a solution by exploring fewer nodes with respect to the counterpart. However, the A-MCTS behaviour seems to be less stable than MCTS, and further investigations are required to be able to exploit its exploration efficiency without losing robustness.

A.5 Quicksort Program Hierarchies and Pre-conditions

Table 2 and 3 show the complete hierarchy of programs. The former shows programs for the $M_{alphanpi}$ and $M^S_{alphanpi}$ models. The latter shows programs for the $M_{alphaargs}$ and $M^S_{alphaargs}$ models. Table 4 shows the programs preconditions.

Table 2: Hierarchy of the programs for Quicksort. In this case, the programs cannot accept arguments. The level 0 programs are not learned and they are the atomic actions.

Program	**Description**	**Level**
`quicksort`	Sort the list.	5
`quicksort_update`	Execute `partition` call and populate the stack with the next pointers.	4
`partition`	Order a sub-array and return the pivot for the next operation.	2
`partition_update`	Swap the elements pointed by p_3 (pivot) and p_1 if p_3 pointed value is less than p_2 pointed value.	1
`save_ptr_1`	Save pointer 1 position into the temporary registry.	0
`load_ptr_1`	Load the temporary registry value into the pointer 1 position.	0
`push`	Push pointers values inside the stack.	0
`pop`	Pop and restore pointers values from the stack.	0
`swap`	Swap elements pointed by pointer 1 and by pointer 2.	0
`swap_pivot`	Swap elements pointed by pointer 1 and by pointer 3 (pivot).	0
`ptr_i_left`	Move pointer i (where $i \in \{1, 2, 3\}$) one step left.	0
`ptr_i_right`	Move pointer i (where $i \in \{1, 2, 3\}$) one step right.	0

Table 3: Hierarchy of the programs for Quicksort. In this case, the atomic actions can accept arguments. The level 0 programs are not learned and they are the atomic actions.

Program	Description	Arguments	Level
quicksort	Sort the list.	0	5
quicksort_update	Execute partition call and populate the stack with the next pointers.	0	4
partition	Order a sub-array and return the pivot for the next operation.	0	2
partition_update	Swap the elements pointed by p_3 (pivot) and p_1 if p_3 pointed value is less than p_2 pointed value.	0	1
save_ptr	Save a given pointer position into the temporary registry.	1	0
load_ptr	Load the temporary registry value into the given pointer position.	1	0
push	Push pointers values inside the stack.	0	0
pop	Pop and restore pointers values from the stack.	0	0
swap	Given two pointers, swap the corresponding elements	2	0
ptr_left	Move pointer one, two or all three pointers one step left.	3	0
ptr_right	Move pointer one, two or all three pointers one step right.	3	0

Table 4: Pre-conditions associated to each Quicksort program.

Program	**Pre-condition**
quicksort	Pointer 1 and 3 are at the extreme left of the list. Pointer 2 is at the extreme right of the list. The stack is empty.
quicksort_update	The stack is not empty and the temporary registry is empty.
partition	The temporary registry must contain Pointer 1 position.
partition_update	Pointer 1 position must be lower than Pointer 3 position. Pointer 3 position must be lower than Pointer 2 position. The temporary registry must be not empty.
save_ptr_1	No pre-condition.
load_ptr_1	The temporary stack must not be empty.
push	Check custom boolean condition $(p_1 + 1 < p_2) \vee (p_1 - 1 > 0 \wedge p_3 < p_1 - 1)$.
pop	The stack must not be empty.
swap	Pointer 1 position must be different than pointer 2 position.
swap_pivot	Pointer 1 position must be different than pointer 3 position.
ptr_i_left	Pointer i (where $i \in \{1, 2, 3\}$) is not at the extreme left of the list.
ptr_i_right	Pointer i (where $i \in \{1, 2, 3\}$) is not at the extreme right of the list.