# AlphaSMT: A Reinforcement Learning Guided SMT Solver

by

Zhengyang Lu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my readers.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Satisfiability Modulo Theories (SMT) solvers are programs that decide whether a first-order logic formula is satisfiable. Over the last two decades, these solvers have become central to many methods and tools in fields as diverse as software engineering, verification, security, and Artificial Intelligence. Most modern SMT solvers provide user-controllable *strategies*, i.e., users can define a strategy that customizes a solving algorithm for their own problems by combining individual *tactics* as building blocks. A tactic is a well-defined and implemented reasoning step provided by the SMT solver, which either simplifies, transforms, or solves the given input SMT formula. The flexibility of customizing a strategy to a specialized type of formula is important since no existing strategy is considered optimal for all instances. However, finding a good customized strategy is challenging even for experts.

In this thesis we present a novel class of reinforcement-learning (RL) guided methods, implemented in the `Z3` SMT solver and that we refer to as `AlphaSMT`, which adaptively constructs the expected best strategy for any given input SMT formula. Briefly, the `AlphaSMT` RL framework combines deep Monte-Carlo Tree Search (MCTS) and logical reasoning in a unique way in order to enable the RL agent to learn the best combination of tactics for a given class of formulas. In more detail, a deep neural network serves as both the value function and the policy, evaluating state-action pairs and making the decision of which tactic to choose at each step. The neural network is trained toward the optimal policy by learning from self-exploring sampling solving processes. MCTS is used as a lookahead planning step for each decision made in the sampling processes.

We evaluated the performance of `AlphaSMT` on benchmark sets from three SMT logics, namely, quantifier-free non-linear real arithmetic (QF_NRA), quantifier-free non-linear integer arithmetic (QF_NIA), and quantifier-free bit-vector (QF_BV). In all these logics, `AlphaSMT` outperforms its base solver, `Z3`, by solving up to 80.5% more instances in a testing set. Evaluation results also show that a reasonably longer tactic timeout helps solve more instances and a pre-solver contributes significantly to the speedup.

**Acknowledgements**

First of all, I want to express my sincere gratitude to my supervisor, Dr. Vijay Ganesh. His enthusiastic teaching and insightful mentoring lead me to this exciting research field of advancing artificial intelligence by combining mathematical logic and machine learning. Only with his guidance, encouragement, and support, this MASc thesis research is possible.

My colleague Piyush Jha also contributes greatly. His expertise in machine learning helps shape and advance this research. Joseph Scott, from the research group, also contributes ideas. I also benefit from the String Constraint Solver work with my collaborators, Stefan Siemer, Dr. Florin Manea, Dr. Mitja Kulczynski, and Dr. Joel Day. I am happy working with you guys.

Thanks to my readers, Dr. Arie Gurfinkel and Dr. Mark Crowley.

Many thanks to the Writing and Communication Advisor, Jane Russwurm. So much help and encouragement through all these years, and it is always lovely to see a friend from old times.

Thank you Dr. Liping Fu for all the support as always.

My deepest gratitude to my parents. Without your support, this journey of a restart in academia in a different field would be impossible.

I am grateful that computer science has presented its beauty to me, and I have the courage and luck to take on this MASc journey. Hope that the following Ph.D. studies will continue to be a rewarding adventure!

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background

First-order logic (FOL) is a formal system used to reason about objects, their properties, and relationships. It provides a logical framework for expressing statements and making inferences based on logical connectives, quantifiers, variables, functions, and predicates. First-order theories, which are built upon the FOL framework, reason about specific domains by defining their own sets of symbols (functions, predicates, constants) and axioms. For example, the theory of arithmetic reasons about numbers, arithmetic operations, and their properties. The statement "Infinitely many prime numbers exist" can be expressed in the integer arithmetic theory as "$\forall q \exists p \forall x, y[(p > q) \wedge (x, y > 1 \implies xy \neq p)]$". By employing first-order theories, one can focus on reasoning within specific domains using the expressive power of FOL.

SMT solvers, which stand for Satisfiability Modulo Theories solvers, are automated reasoning tools that determine whether a formula is satisfiable/valid. The input formula is expressed in FOL, usually within a specific theory. Some common SMT theories are linear arithmetic, bit vectors, arrays, pointer logic, and strings. High-performance SMT solvers, which can efficiently solve satisfiability problems in these theories play increasingly important roles in the fields of hardware and software verification, automated theorem proving, compiler optimization, security, artificial intelligence, and beyond.

SMT can be seen as an extension of the Boolean satisfiability problem (SAT). SMT solvers utilize SAT-solving algorithms as a foundation to process the input formula's Boolean skeleton and incorporate specialized decision procedures for different theories to

handle theory-specific constraints. Since the SAT problem is already NP-complete, SMT problems are generally intractable or undecidable. Although most SMT solvers adopt a high-level DPLL(T) framework [24], different solvers vary significantly in terms of preprocessing steps, theory-specific decision procedures, implementation details, etc. It is widely believed that there is no single algorithm that always performs the best on all instances of a hard problem. This statement certainly matches the observations in the SMT-solving field: one solver which performs well in one type of instance may perform very poorly in other types. Thus, there is naturally an algorithm selection problem for SMT practitioners: which solver to choose for one's own instances? An algorithm portfolio, which selects multiple algorithms and runs them in sequence or in parallel to solve a single instance, provides a flexible solution to this algorithm selection problem.

However, for SMT problems, there is another level of flexibility: in addition to choosing from a pool of start-to-end algorithms, one can build an algorithm by selecting and combining tactics. In this thesis, we adopt the concepts of tactic and strategy introduced in [7]. A *tactic* is an algorithmic proof method or reasoning step within the whole solving process, and each can be viewed as a building block of a decision procedure. Tactics are usually well-defined and implemented in SMT solvers. Some tactics simplify formulas, e.g., constant folding; some tactics transform formulas, e.g., bit-blasting; some tactics solve formulas, e.g., Simplex. A *strategy* refers to the heuristics of how tactics are orchestrated to form a decision procedure. Modern SMT solvers, e.g., `Z3`, usually offer user-controllable strategies. In other words, solver users can build their own solving strategies by connecting certain tactics sequentially, conditionally, or iteratively, for different instances. Strategies play a key role in solver performance: one strategy may solve a hard instance in seconds, but another strategy may not be able to solve the same instance at all. In this thesis, the problem of crafting a specialized strategy for a given instance is called the "tactic selection problem".

## 1.2 Research Problem and Objectives

This research aims to answer the SMT tactic selection problem: how to construct the best strategy by sequencing individual tactics for a specific problem instance? Crafting a suitable strategy often relies on human expertise and extensive trial and error. However, such expensive efforts are still slow to adapt to the newly emerging instance types and are unable to capture complex patterns. Machine learning (ML), on the other hand, can automatically explore and learn from experience and data, and shows promise in improving the tactic selection heuristics.

ML methods have already been contributing to enhancing heuristics in logic solvers [10, 9]. Liang et al. [19] successfully used reinforcement learning (RL), a type of ML technique, to improve branching heuristics within the CDCL framework [21] for SAT problems. Following works, such as `NeuroCore` [32] and `GQSAT` [15] also used different ML techniques for better branching decisions. ML has also been applied to improve other aspects of the solver heuristics, e.g., how to select a preprocessor [5], how to select a restart/reset strategy [23, 20, 16], how to choose learned clauses to delete [36], and how to choose solver parameters [13]. Moreover, other than improving heuristics within a solver, ML has been used to select among different solvers (the algorithm selection problem). `SATzilla` [42] and `MachSMT` [30] built machine learning models to predict solver performance via offline training, and used such models to select the best solver for each instance. `MedleySolver` [25] is an online SMT algorithm selection framework, requiring no prior training. For every query, it selects a sequence of solvers with an assigned timeout. The algorithm selection problem is modeled as a Multi-Armed Bandit (MAB) problem, and the tool learns to make a decision via trial and error. Scott et al. [31] introduced the concept of "meta solver", defined as a tool containing a set of subsolvers that get adaptively called, in a sequence, based on online and offline performances. `Goose`, a solver for the Verification of Neural Network (VNN) problem is built based on this concept. See Section 2.4 for a broader and deeper literature review on this topic.

In this research, we propose to use ML, specifically, RL, to tackle the tactic selection problem. Learning to solve this problem would be more comprehensive than improving one specific solver heuristic since different tactics encode different aspects of solver heuristics. The tactic selection problem is also more granular than the algorithm selection problem. This increased flexibility offers great richness in how a decision procedure can be constructed. This richness also brings difficulties in finding an optimal chain of tactics to apply: considering the large number of tactics and their combinations, the search space is vast. `FastSMT` [1] is one early attempt to learn how to select SMT tactics for a specialized benchmark set. Its learning is twofold: it first uses RL methods to find a set of fixed candidate strategies that each works well for certain instances, and then synthesizes these strategies into one combined strategy based on the performance data of exhaustively testing all candidate strategies on all training instances. This combined strategy would be considered a good solution to all the instances in the particular benchmark set. We argue that there is room for improvement in terms of strategy adaptiveness and training efficiency of the learning method.

The objective of this thesis is to design and implement a combined learning and logic framework, which enables an agent to learn how to dynamically select a tactic at each step through logic feedback. Guided by this trained agent, an SMT solver will be more effective

and efficient for specialized problem tasks.

## 1.3 Overview of AlphaSMT

For each input SMT formula that is solvable within a specified timeout, there must exist (at least) one optimal sequence of tactics which solves this formula in the shortest time. Therefore, the ideal solution to the tactic selection problem is to ask an oracle each time when a tactic decision needs to be made, and the oracle always knows the best tactic to apply given the current formula to solve. Noted that usually multiple sequential tactic decisions need to be made in order to solve a formula, and the formula is rewritten along the process. Thus, the tactic selection problem can be nicely framed as a sequential decision-making problem, as shown in Figure 1.1: an agent sequentially makes decisions on which tactic to apply, given the current solving status, in order to solve the formula in the shortest time. In recent years, reinforcement learning (RL) methods have shown tremendous progress over many sequential decision-making problems [17]. The motivation of this research is that, through reinforcement learning, an agent learns the optimal policy for the tactic selection problem, where policy is a mapping from states (current solving situations) to action (tactic) selections. However, due to the infinite state space (there are infinitely many possible original and intermediate formulas) and limited training resources, it is infeasible to learn an optimal policy in the tabular form, which finds the best action for every possible state. However, we argue that, with (1) lookahead planning methods, such as Monte-Carlo Tree Search (MCTS), and (2) function approximation methods, such as training a Deep Neural Network (DNN) to approximate the policy or value function, reinforcement learning is able to find an increasingly close approximation to the optimal policy, by efficiently exploring and exploiting more promising paths and generalizing such experience to construct an approximation for the entire policy/value function. Therefore, it is feasible to train a good tactic selection policy, which works generally well for a specialized problem set, by learning from the experience of solving formulas in a relatively small subset.

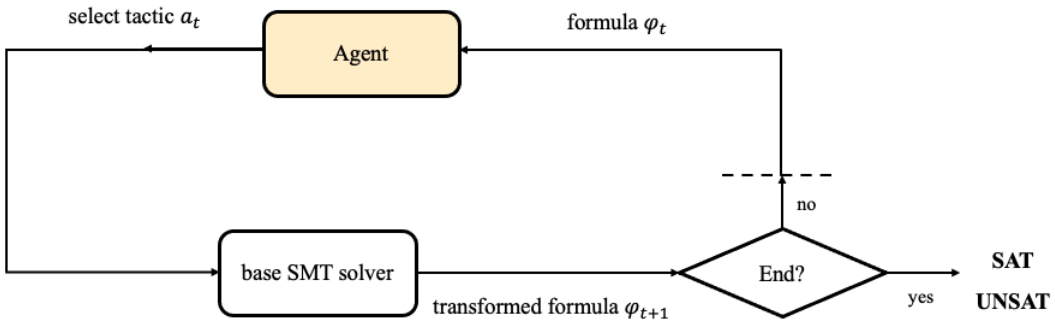



Figure 1.1: A simple illustration of framing the tactic selection problem as a sequential decision problem

Based on this motivation, we present `AlphaSMT`, an adaptive RL-based SMT solver. `AlphaSMT` uses an RL framework that combines MCTS and DNNs to train a tactic selection agent. This deep MCTS framework is first introduced in `AlphaZero` [34]. `AlphaZero` is an RL-based computer Go program, introduced by DeepMind in 2017. It showed superior performance over all other human and computer players in the board game GO, which is considered the most challenging classical game for artificial intelligence. The deep MCTS framework uses one deep neural network for both policy and value function approximation, and uses MCTS for lookahead searches inside the neural network training loop, resulting in rapid improvement and stable learning for problems with vast state and search spaces. `AlphaSMT` adapts this framework to the tactic selection problem, with key changes being made in terms of logic feedback, feature engineering, usage of MCTS, neural network architecture, etc.

`AlphaSMT` is currently built upon an established SMT solver `Z3` [6]. It selects from `Z3` built-in tactics, and uses `Z3` as its base solver to execute selected tactics. `AlphaSMT` is able to make good tactic choices for a specialized problem set after training on a small representative subset. `AlphaSMT`'s decision-making unit is a deep neural network, which maps solving states to the recommended tactic selection probabilities. The neural network incorporates techniques such as transformer [40] and batch normalization [11].

## 1.4 Contributions

The contributions of this thesis are as follows:

1. Present the tactic selection problem and formally frame it as a full reinforcement problem. In the framing, an agent iteratively selects a tactic from candidates, observes how the formula is rewritten, and selects again, for the purpose of solving the input formula the most efficiently. By formalizing the tactic selection problem as a sequential decision problem, we can apply RL methods to the problem.

2. `AlphaSMT`, an adaptive SMT solver, is built based on an RL framework that combines deep MCTS and logical reasoning. The RL agent in `AlphaSMT`, in the form of a deep neural network, learns the best sequence of tactics for a given class of formulas, by actively attempting to solve sampling instances from a training benchmark set. The sampling solving processes are guided by MCTS at each step to ensure a steady policy improvement over the training loop. Logical reasoning, i.e., a base SMT solver, provides feedback for the learning process.

3. We performed an experimental analysis of `AlphaSMT` on three benchmark sets of three SMT logics, i.e., quantifier-free non-linear integer arithmetic (QF_NIA), quantifier-free non-linear real arithmetic (QF_NRA), and quantifier-free bit-vector (QF_BV). The tested `AlphaSMT` solvers were built with various configurations of tactic timeout and pre-solver time. We found that `AlphaSMT` solved about 14.4% more QF_NIA instances, 80.5% more QF_NRA instances, and 3.6% more QF_BV instances than its base solver `Z3`, with a 5-minute time limit for each instance. The proposed RL framework greatly improves the solver's ability to solve challenging instances, and with proper configuration, it expedites the solving processes.

## 1.5 Structure of this Thesis

Chapter 1 introduces the research problem and outlines our objectives and contributions. The remainder of this thesis is organized as follows:

- Chapter 2 begins by establishing the current state of knowledge in the fields of reinforcement learning and SMT solvers. By examining their respective concepts and paths, we then conduct an in-depth review on their intersection: how do learning methods help build better logic solvers? This literature review not only walks

through the existing works on the topic but also provides a critical analysis of the strengths and limitations of previous research. Our work is motivated by the existing research gap.

- Chapter 3 demonstrates our proposed RL framework for the SMT tactic selection problem. We first rigorously formalize the tactic selection problem as an MDP and then introduce the framework that combines deep MCTS and logical reasoning to tackle this problem.

- Chapter 4 presents the evaluative experimental designs and results for `AlphaSMT`. This Chapter also provides analyses and discussions of the result data, in light of the research objectives.

- In Chapter 5, we present a summary of the key research findings, provide a thoughtful reflection on the limitations of the study, and propose potential avenues for future research.

# Chapter 2

# Literature Review

## 2.1 Reinforcement Learning

Reinforcement learning (RL) is learning what expected actions to take according to the current situation (state) in order to maximize a quantified reward, via a trial-and-error search. Commonly in RL problems, one action does not only affect the immediate next-step reward but also all the subsequent situations (states) and rewards [37]. Such reinforcement learning problems can be abstractly framed as a mathematically idealized form of Markov decision processes, or MDPs. As illustrated in Figure 2.1, MDP abstracts a process of optimization through agent-environment interaction. The agent, which is the learner and decision maker, interacts with the outside environment. The interaction happens iteratively by the agent selecting actions, and the environment generating new states and providing numerical rewards, in response to the agent's actions. The agent aims to learn an optimal policy, which tells what is the best action in each state, through the iterative interaction. [37].

Board games usually fit perfectly into the MDP framework: a player needs to make decisions on what is the next move based on the current board situation (state) in order to win the game (reward); all moves contribute to the game dynamics and final results in a complicated and entangled way. RL has already shown its strong power in board game applications: in 2016, `AlphaGo` [35], an RL-based computer program developed by DeepMind, defeated Lee Sedol, the world champion Go player, in a high-profile match. Go is known as, by one measure, the most challenging classical game for artificial intelligence because of its complexity; a computer program defeating a Go world champion is definitely a remarkable event in the history of artificial intelligence. The searching in `AlphaGo` uses deep neural

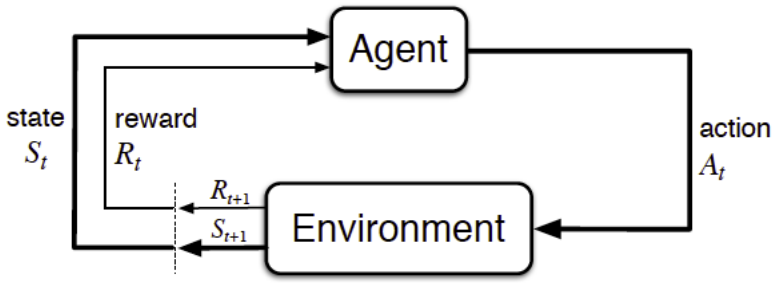



Figure 2.1: The agent-environment interaction in a Markov decision process [37]

networks to evaluate positions and select the next moves. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. In 2017, DeepMind introduced a new version called `AlphaZero` [34], which learns solely from self-play, without human guidance or domain knowledge. `AlphaZero` won 100-0 against the previous champion-defeating version of `AlphaGo`.

`AlphaZero` uses a novel reinforcement learning algorithm, as shown in Figure 2.2, to train a deep neural network $f_\theta$ with parameters $\theta$. This neural network serves both to approximate the policy (mapping from boards to the probability distribution over actions for the next move) and the state value function (estimating the expected winning probability starting in a certain state and following a particular policy afterwards). The neural network is trained via games of self-play, and in each position of a self-play game, a Monte-Carlo Tree Search (MCTS) [3] is performed as a lookahead search. The MCTS is guided by the neural network to generate a probability $\pi$ for each move. This probability is believed to be a stronger policy than the raw move probabilities $p$ from the input neural network; thus the MCTS step can be considered as a policy improvement step. The self-play games, which select moves according to the improved MCTS policy and use the delayed game winner as a sample return, can be viewed as a policy evaluation step. This reinforcement learning algorithm also fits into the Generalized Policy Iteration (GPI) framework [37]: a policy improvement step, which greedily improves the policy with respect to the current value function, and a policy-evaluation step, which updates the value function towards the improved policy, interact to achieve an optimal policy after iterations. This GPI framework is one main idea of reinforcement learning. Within each iteration, `AlphaZero` uses the improved policy and sample returns as the neural network training labels. Ideally, the policy and value function approach the optimal ones through iterations.

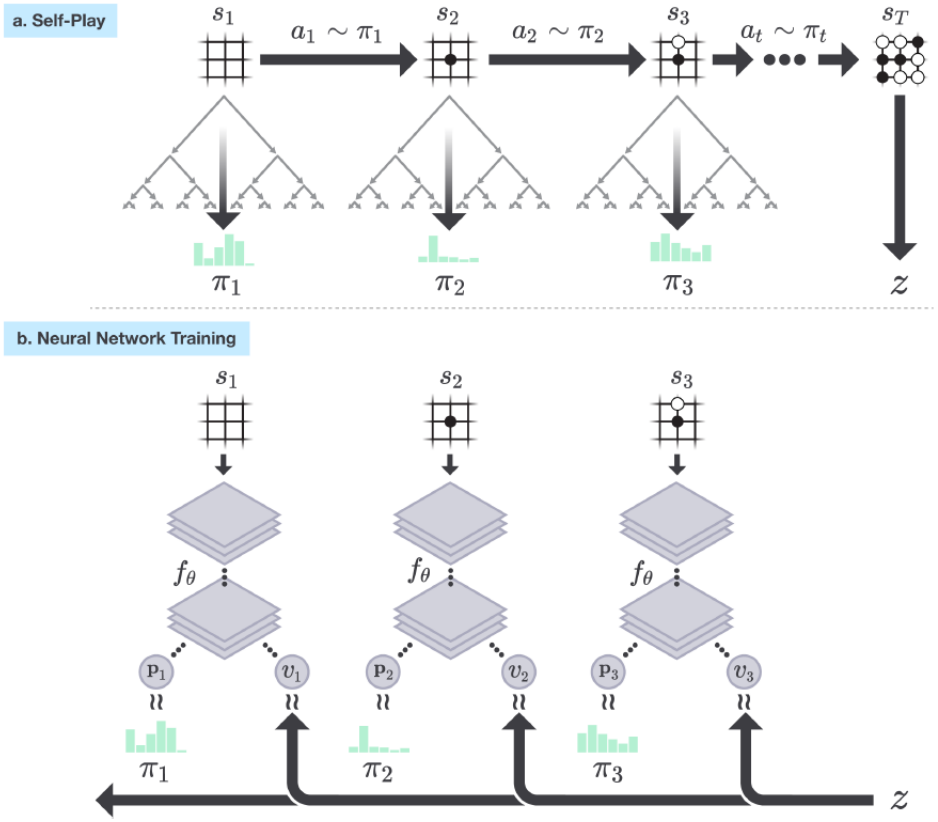



Figure 2.2: The self-play reinforcement learning in `AlphaZero` [34]

As mentioned in the previous paragraph, MCTS executes a lookahead search at each position of the self-play games inside the training looping. It takes a neural network $f_\theta$ and a root position $s$ as inputs and outputs a probability distribution over moves, which is regarded as an improved policy at $s$. Specifically, through simulations starting from $s$, MCTS incrementally builds a search tree rooted at $s$, which stores statistics for state-action pairs that are likely to be reached in a few steps. Each simulation starts from the root $s$, walks through a path in the tree, and exits at a leaf node. In each simulation, the path within the tree is selected to maximize an upper confidence bound (UCB) value at each step. The UCB considers both the move probabilities predicted by the neural network and the statistics collected from the simulations and keeps a balance between exploitation and exploration. The simulation ends when reaching a leaf node and the leaf node is expanded and evaluated just once by the neural network. The evaluations are then

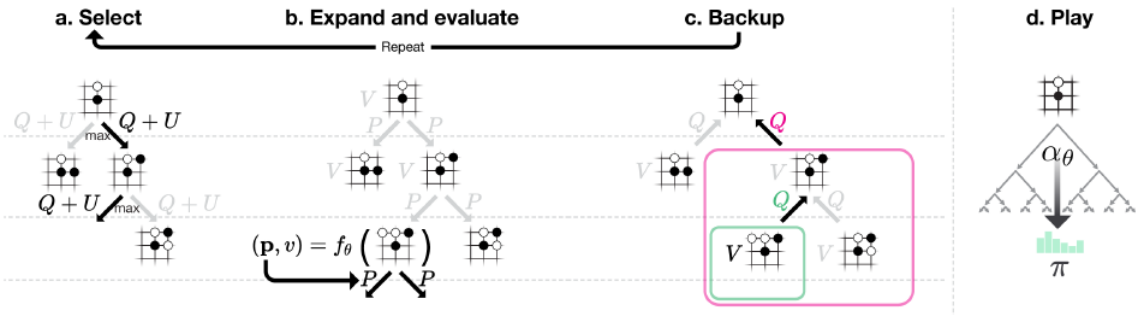



Figure 2.3: Monte-Carlo Tree Search in `AlphaZero` [34]

backed up to update the action values for the state-action pairs along the path. As in any Monte Carlo methods, the action value for one state-action pair is estimated as the average of all simulated sample returns. MCTS continues this four-step process (selection-expansion-evaluation-backup) until it reaches a time or resource threshold. Then, finally, a probability distribution proportional to the visiting counts for all actions at the root $s$, $\pi = \alpha_\theta(s)$, is returned. This probability distribution is considered a recommended policy at $s$ and a target for the neural network to learn. Figure 2.3 also explains how the MCTS works in `AlphaZero`.

Neural network training samples are generated from numerous self-play episodes. In each self-play episode, MCTS, guided by the neural network from previous iterations, is used to play each move. The episode ends when the Go game ends, and the episode receives a final reward, $r_t$, regarding the winner of the game. One training sample will be created for one step in the episode: $(s_t, \pi_t, z_t)$. $s_t$ is the state representation at step $t$, $\pi_t$ is the MCTS output at the step, and $z_t = \pm r_t$ is the final reward with respect to each player. The neural network is trained from the dataset of $(s_t, \pi_t, z_t)$ sampled uniformly among all time steps of all self-play episodes in the iteration. The goal of the training is to minimize a loss function that measures the errors in both move probabilities and value estimations between the neural network predictions and the self-play sample data.

This neural network training process will be iterated multiple times. Each neural network trained in iteration $i$, $f_{\theta_i}$, will be evaluated against the previous best network, $f_{\theta_*}$; only if $f_{\theta_i}$ beats $f_{\theta_*}$, $f_{\theta_i}$ will become the incumbent best network and be used subsequently for self-play guidance and performance comparison. Such a mechanism guarantees a steady improvement.

## 2.2 SMT Solver

Satisfiability Modulo Theories, or SMT for short, is the field of determining the satisfiability of formulas in first-order logic. First-order logic, or predicate logic, can be seen as an extension of propositional logic. First-order logic introduces additional symbols of quantifiers, functions, and predicates, enlarging the expressiveness of the logic language. For example, Fermat's Last Theorem can be expressed in first-order logic ($\forall a, b, c, n[(a, b, c > 0 \wedge n > 2) \rightarrow a^n + b^n \neq c^n]$), but not in propositional logic. Satisfiability refers to whether there exists an assignment of the variables in the formula under which the formula evaluates to True: if yes, the formula is satisfiable (SAT); if not, the formula is unsatisfiable (UNSAT).

SMT is usually concerned with the formulas of an arbitrary theory or combinations thereof. Every theory is defined over a set of symbols (functions, predicates, constants), and the formulas of the theory are restricted to the symbols of the theory. Some common first-order theories are linear arithmetic, bit vectors, arrays, pointer logic, and strings. The satisfiability problem for these theories and their combinations is critical to the field of formal verification, which is a subfield of software engineering dedicated to proving the correctness of computer programs with respect to a given formal specification [14]. Thus, SMT solvers, which implement algorithms for solving satisfiability problems of first-order formulas, are widely used in fields of software and hardware verification, theorem proving, compiler optimization, etc. Modern SMT solvers mainly adopt a DPLL(T) [24] architecture. DPLL(T) is a theoretical framework that abstractly defines how satellite theory-specific solvers (e.g., arithmetic, bit-vector, array) and the SAT solver are integrated to solve an SMT problem in a specific theory. This abstract framework still leaves a large room for different heuristic designs. Strategies describe the recipe for heuristics in the SMT solver.

A *strategy* refers to a solving algorithm that is constructed by combining individual *tactics* as building blocks of the said algorithm [7]. A *tactic* is a well-defined and implemented reasoning step in an SMT solver. For example, in the SMT solver `Z3` [6], the tactic `simplify` is a preprocessing step that simplifies input formulas; the tactic `solver-eqs` eliminates variables using Gaussian elimination; the tactic `smt` wraps the core DPLL(T) solver as a tactic. Every tactic rewrites the input formula, so the whole solving process can be viewed as sequentially applying rewrite rules (tactics) until the rewritten formula is trivially satisfiable or unsatisfiable (see an example in Figure 2.4). A strategy uses combinators, such as `then`, `repeat`, `or-else`, to group tactics into a decision procedure. For example, the strategy `(then simplify solver-eqs smt)` describes a decision procedure of sequentially applying the tactics `simplify`, `solver-eqs`, and `smt`.

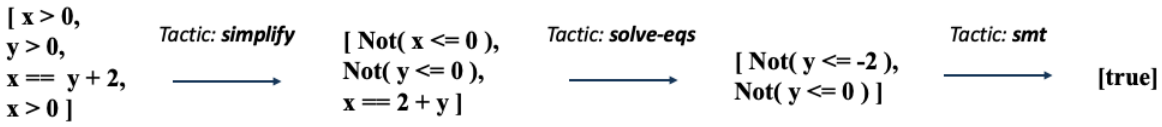



Figure 2.4: One example of solving an input formula by sequentially applying tactics in Z3

Strategies are crucial to the effectiveness and efficiency of an SMT solver: a change in strategy can make an originally unsolvable instance to be solved in seconds. Default strategies in solvers are usually handcrafted by solver developers for known and limited classes of problems. Such default strategies may perform poorly on new or specific classes of problems. These days, more SMT solvers are offering user-controllable strategies, allowing users to exert fine-grained control over the heuristic aspects of the solver. However, the work of finding the appropriate strategy for specific problems requires both expert knowledge and extensive effort.

## 2.3 Algorithm Selection and Tactic Selection

In most logic solving fields, no single solver prevails for all instances. This, practitioners are usually facing an "algorithm selection problem" [27]: how to choose a solver that best suits one's particular needs. One common solution is to choose the solver with the best average performance on a representative benchmark set. For example, solver users can choose the winning solver from the SAT and SMT competitions [8, 41] for all their instances. However, this "winner-take-all" approach does not leverage the strength of solvers which may not perform well on average but have a significant edge on particular types of instances.

The ideal solution would be to have an oracle that knows how long it takes for each candidate solver to solve a particular instance and selects the solver with the best performance predicted by the oracle. Despite the absence of such an oracle, a learned approximate runner predictor, in place of the oracle, performs fairly well to select one algorithm in many settings [42, 30]. Other than choosing one solver for each instance, one can also build a portfolio solver combining multiple solvers. Broadly speaking, a portfolio solver refers to any solver that utilizes multiple different solver algorithms to solve a single instance. A portfolio solver can choose solvers in the portfolio to be executed in sequence or in parallel. Additionally, the decision of which solver algorithm to be executed can be made on the

fly, leveraging information collected along the solving process. Portfolio solvers provide flexibility to the algorithm selection problem.

In SMT, customizable strategies provide another level of flexibility. Each tactic is a building block of the solving algorithms. The solver users can not only choose which start-to-end algorithm to use, but also control how individual reasoning steps are combined to build an algorithm. For example, in `Z3`, one can attempt to solve a nonlinear arithmetic formula using the strategy `(then simplify smt)`. Alternatively, she can also add another preprocessing step, changing the strategy into `(then simplify solve-eqs smt)`; or replaces the general solving tactic `smt` with one particular solving tactic `qfnra-nlsat`, resulting in the strategy `(then simplify qfnra-nlsat)`. The strategy `(then nla2bv bit_blast sat)` shows another option to solve a nonlinear arithmetic formula: first converting it to a bit-vector formula, and then solving the resultant formula using a bit-blasting based method. In this thesis, the problem of constructing an SMT strategy by sequencing candidate tactics to solve a particular instance is called the "tactic selection problem". It can be viewed as a generalization of the "algorithm selection problem", with more granularity.

## 2.4 Learning to Solve Logic Formulas

Machine learning (ML) now plays a key role in many logical reasoning tools, helping them become more powerful and efficient. The functionality of ML in solvers ranges from directly predicting satisfiability, and optimizing solver heuristics, to algorithm selection [9, 10]. The tactic selection problem is most related to the last category; however, it also shares many similarities with the second type of problem, e.g., requiring step-wise decisions during one solving process. Therefore, in this section, we first provide a concise review of relevant research that uses learning to optimize certain solver heuristics, and then walk through the works that apply learning methods to algorithm/tactic selection problems in greater detail. Although we do not directly discuss the ML application in satisfiability prediction, this technique is leveraged in some research that attempts to improve solver heuristics. One such example [32] is covered in the review.

### 2.4.1 Learning Solver Heuristics

Conflict-Driven Clause Learning (CDCL) [21] is the dominant framework for modern SAT solvers, and numerous attempts have been made to improve various aspects of CDCL

solver heuristics [22, 43]. Liang et al. [19] first successfully implemented an RL-guided SAT solver, `MapleSAT`, which learns to improve branching heuristics. Specifically, they modeled the branching variable selection problem as an online multi-armed bandit problem, where the optimization objective is to maximize the learning rate, defined as the propensity for variables to generate learned clauses. A novel branching heuristic called `learning rate branching` (LRB) is developed based on a multi-armed bandit algorithm exponential recency weighted average. The SAT solver, built upon LRB, won 1st in the SAT Competition 2016 Application category. `NeuroCore` [32] also aims to improve the branching heuristics, but in a very different way. `NeuroSAT` [33] is a neural-network-based tool that predicts propositional formula properties. `NeuroCore` is built based on a trained `NeuroSAT` model that predicts the unsatisfiable core. It incorporates this prediction into its branching heuristics. `GQSAT` [15] is another SAT solver that applies RL to improve branching heuristics. It models the SAT problem as a full reinforcement learning problem and applies Deep Q-Learning (DQN) algorithm to train a branching decision agent. Moreover, there are works that use ML to learn how to select a preprocessor [5], how to select a restart/reset strategy [23, 20, 16], how to choose learned clauses to delete [36], and how to select solver parameters [13].

### 2.4.2 Learning to Select Algorithms

Now let us shift our focus to the algorithm selection problem. `SATzilla` [42] is one of the first algorithm selection tools for logic solvers. It automatically constructs a per-instance algorithm portfolio for SAT. The choice of algorithm, or solver, is based on an Empirical Hardness Model (EHM), which is trained by the performance of solvers on the benchmark. `SATzilla` won five medals in both the 2007 and 2009 SAT Competition, proving the tool's effectiveness.

`MachSMT` [30] is a algorithm selection tool for SMT solvers. Upon facing an instance, it selects an existing solver which is predicted to be the best algorithm for solving this particular instance. The prediction is based on both empirical hardness models (EHMs) and pairwise ranking comparators (PWCs). These models are trained upon samples of applying various solver algorithms to SMT instances. Experimental results show that `MachSMT` outperforms the SMT-COMP 2019 and 2020 competition winners in 54 out of 85 divisions, notably in important logics, e.g., BV, LIA, NRA.

`FastSMT` [1] is one prominent previous work of applying learning algorithms to optimize SMT solving strategy. It learns an efficient synthesized SMT strategy for a specialized benchmark set. The learning process is twofold: (1) train a policy that searches for the

best strategies for each formula and create a dataset of strategies for the benchmark; (2) synthesize one combined strategy, which captures the trained policy in a language interpretable to the solver (in their case, `Z3`), from the strategy dataset. In the first step, `FastSMT` explores training various types of policy models (e.g., bilinear model, neural network model) using the `DAgger` framework [29]. In the second step, a combined strategy is synthesized from the set of candidate strategies using decision tree learning in a top-down manner. `FastSMT`'s experiment results show that their synthesized strategy solves 17% more formulas and is up to 100× faster than the default `Z3` strategy, based on tests on five benchmarks of varying complexity across three SMT logics (i.e., QF_NRA, QF_BV, and QF_NIA).

Pimpalkhare et al. [25] developed an online algorithm selection framework for SMT, `MedleySolver`. They framed the algorithm selection problem, i.e., choosing an SMT solver for a particular instance, as a Multi-Armed Bandit (MAB) problem. MAB models a class of problems which repeated choices need to be made among multiple actions, and the actions only affect immediate rewards. In their problem framing, the action space is the set of candidate solvers and the reward is the solver performance for each instance. They also extend the MAB problem: instead of choosing one single solver, `MedleySolver` selects a sequence of solvers and assigns each solver in the sequence with a predicted timeout. The tool learns to make better algorithm selection decisions via balancing exploration and exploitation along the solving processes. An empirical evaluation on 2,000 benchmarks shows that `MedleySolver` without pre-training solves 93.9% of the queries solved by the virtual best solver selector.

Scott et al. [31] introduced a novel concept of "meta solver". They defined a meta solver as "a tool containing a set of subsolvers that get adaptively called, in a sequence, based on online and offline information collected about their performance histories on a given input". `Goose`, a meta-solver for the Verification of Neural Network (VNN) problem, is presented in this paper. The solver built its adaptive sequential portfolio using three synergizing techniques, i.e., algorithm selection, probabilistic satisfiability inference, and time-iterative deepening. The authors found a 25.6% improvement over the VNN-COMP'22 winner in PAR-2 score across more than 1500 benchmarks.

### 2.4.3 Summary

All the research above involves augmenting the logical solver with learning techniques. Here, I want to summarize the key similarities and differences among these methods, to help readers have a clearer understanding of the development of the field and the contribution of

this thesis research. Such summaries are also presented in Table 2.1. One major difference among these learning solvers is whether the learning happens in a *supervised learning* or in a *reinforcement learning* manner. Supervised learning is learning from a provided training set of labeled examples, and the system is expected to learn how to predict labels for situations not present in the training set. `SATzilla` and `MachSMT` are supervised learning tools: a dataset of the performance of all solvers running on some instances is given, and a machine learning model learns to predict the running time or the best solver for unseen instances. In other words, supervised learning learns the correct action as instructed by the provided training data. In contrast, reinforcement learning learns through its own active interactions with the environment. For example, `MedleySolver` initially applies different algorithms to instances almost randomly and observes their performance; these observations help the agent gain a growing knowledge and, over time, make better educated guess on which algorithm to use. Moreover, `FastSMT` applies a mixed methodology: it uses reinforcement learning to explore the strategy space, and uses supervised learning to synthesize one combined strategy using a training dataset created from the strategy space. One important feature of reinforcement learning is the *onlineness*. Here, an online solver refers to a solver constantly learning from its solving experiences at runtime. Such a solver, e.g., `MedleySolver` may not require a pre-training stage.

Table 2.1: Summaries of Existing Learning Solvers

| Solver | Problem | Learning Methods | Online | Adaptive | Transferable |
|---|---|---|---|---|---|
| `SATzilla` | algorithm selection | supervised learning | N | N | Y |
| `MapleSAT` | branching | MAB | Y | Y | N |
| `FastSMT` | tactic selection | reinforcement learning & supervised learning | N | N | Y |
| `NeuroCore` | branching | supervised learning | N | Y | Y |
| `GQSAT` | branching | reinforcement learning | N | Y | Y |
| `MachSMT` | algorithm selection | supervised learning | N | N | Y |
| `MedleySolver` | algorithm selection | MAB | Y | N | Y |
| `Goose` | algorithm selection | supervised learning | N | Y | Y |
| `AlphaSMT` | tactic selection | reinforcement learning | N | Y | Y |

**Note**: **online**: learning happens at solver runtime; **adaptive**: solving heuristics are adjusted according to the status quo situation during the solving process of one input formula; **transferable**: the model learned from the experience of solving certain formulas can be applied to other formulas.

Different tools listed above apply learning techniques to tackle different decision problems. For CDCL solvers, we have listed research that applies ML to improve heuristics about branching, preprocessor selection, restart strategy, learned clauses deletion, and solver parameter selection. For algorithm/tactic selection problems, the agent learns to predict the best algorithm for each instance. However, the granularity differs: while `SATzilla`, `MachSMT`, and `Goose` select from start-to-end algorithms, `FastSMT` and `AlphaSMT` build an algorithm from tactics.

Furthermore, some learning happens within the solving process of one single instance, while some happen across instances. In the former scenarios, there are usually many

decisions to make to solve one formula, and the agent is observing and learning from early decisions and their effects. For example, there are tons of branching decisions to make in the CDCL algorithm [21] for SAT problems. `MapleSAT` learns to make better branching decisions from early experience of exploring different decision options. We call a solver *adaptive* if it has this property of adjusting solving heuristics during the process of solving one particular instance. Also, some solvers learn more general knowledge of what strategies may work better for certain types of instances, and apply this knowledge to future unseen formulas. We define a learning solver to be *transferable* if what it learns can be applied across instances. `SATzilla` is a transferable learning solver. It learns which algorithm performs the best for different types of formulas from a training set, and achieved remarkable performance when applying this knowledge to formulas unseen in the training set. Noted that adaptiveness and transferability are not contrasting with each other. For example, `MachSMT` is both adaptive and transferable.

In this research, our goal is to build an RL-guided SMT solver, `AlphaSMT`, which adaptively selects tactics during the solving process. Because different tactics encode different aspects of solver heuristics, learning to solve the tactic selection problem is more comprehensive than the research problems that improve one specific solver heuristic (as listed in Section 2.4.1. The tactic selection problem is also more granular than the algorithm selection problem (listed in Section 2.4.2). One early work in the field of learning to select SMT tactics is `FastSMT`. As mentioned earlier, its learning is twofold: it first uses RL methods to find a set of fixed candidate strategies that each works well for certain instances, and then synthesizes these strategies into one combined strategy based on the performance data of exhaustively testing all candidate strategies on all training instances. This combined strategy is their solution to a particular benchmark set. We argue that `AlphaSMT` is more adaptive since it constructs strategies dynamically instead of choosing from a set of fixed strategies. Moreover, `AlphaSMT` is more efficient in training as it uses a deep MCTS framework that actively explores more promising paths instead of exhaustive searching and testing.

# Chapter 3

# Reinforcement Learning for SMT Tactic Selection

In this chapter, we present a novel reinforcement learning framework for the adaptive tactic selection problem for SMT solvers. Section 3.1 thoroughly examines how the tactic selection problem is formalized as a Markov Decision Process (MDP) and what are the characteristics of this formalization, and Section 3.2 proposed a reinforcement learning framework for this formalized problem.

## 3.1 Framing the Tactic Selection Problem as a Markov Decision Process (MDP)

### 3.1.1 Formalization

The Markov Decision Process (MDP) is a mathematical framework that formalizes the process of an agent learning from interactions with the outside environment, in order to achieve a goal. The MDP framework is abstract and flexible, leaving much design room to accommodate various problems. This section explains how the tactic selection problem is formalized as an MDP, mapping the elements in the tactic selection problem to MDP components. A high-level illustration of the formalization is shown in Figure 3.1.

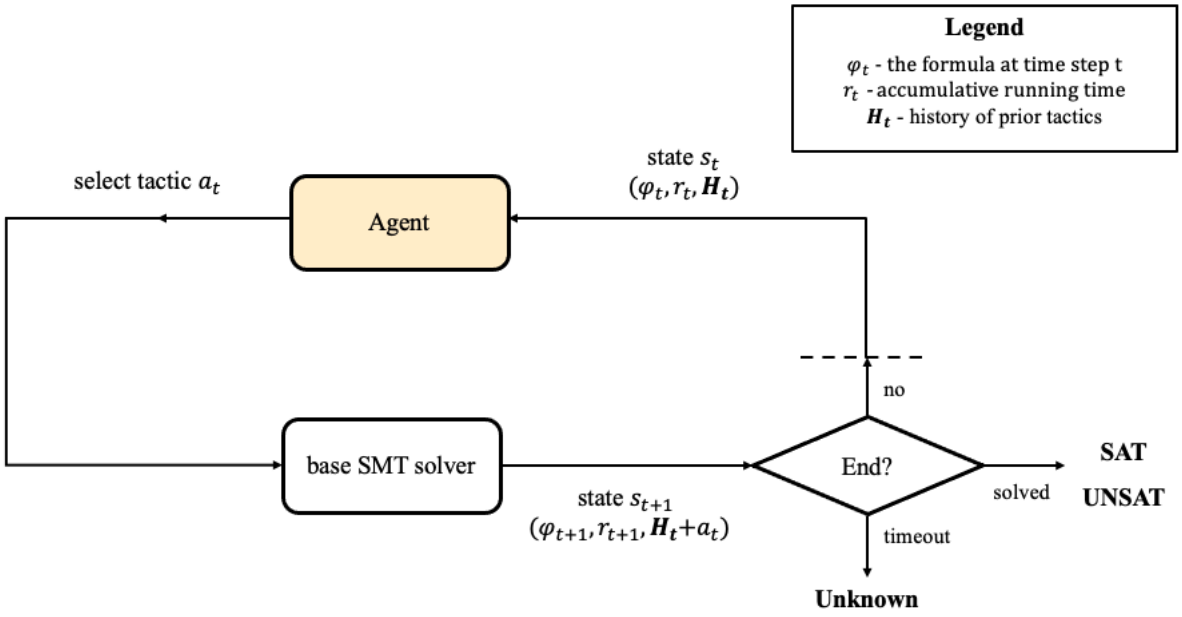



Figure 3.1: Formalization of the tactic selection problem

### Agent

The agent is both the learner and decision-maker in the MDP framework. In the tactic selection context, the agent is supposed to make the tactic decision $a_t$ given the current solving situation at each step $t$, and learn to make better choices from the previous solving experiences. Usually, we express the agent's decision-making scheme in the form of a policy. A policy is a mapping from states to probabilities of selecting each possible action.

### Environment

In MDP, the environment is a broad and relative concept: anything that cannot be controlled arbitrarily by the agent is considered part of the outside environment. In this context, the environment is basically a base solver, which is able to apply the selected tactic $a$ to the formula $\varphi$ and to collect the results of this application, e.g., formula transformation, consumed time, extra feedback. The environment is also responsible for storing all historical information if needed, e.g., prior tactics, accumulated time.

### Episode

The process of solving one instance, $\varphi$, is called an episode in the tactic selection problem. As with industrial applications and in the SMT-COMP, there is an episode timeout, $C_{eps}$, associated with each solving process: after a time of $C_{eps}$, the solver will give up solving the instance. Consequently, every episode will end at a final time step $T$, either when the instance is solved, or when the episode timeout is reached.

### Action

Naturally, in the tactic selection problem, the actions are the tactics of choice. The action space, i.e., the tactic set $Tactics$, is dependent on the theory/logic to which the benchmarks belong. At each step of the episode, the agent chooses one action/tactic, $a \in Tactics$. The agent can also assign a timeout value, $C_{tactic}$, for each tactic in selection: if the tactic application is not completed after $C_{tactic}$, this tactic attempt would be forcefully given up and the formula remains the same. A timeout scheme called time iterative deepening [31] can be applied. This scheme exponentially increases the wall-clock timeout for tactics throughout the whole solving process. This scheme encourages trying more tactics with short timeouts at early stages, and then applies tactics with increasingly longer timeouts when more information is collected during the solving process. Also, a tactic may not be applicable to a certain formula, e.g., applying the SAT solving engine `sat` to a `QF_NIA` formula. In our modeling, an inapplicable tactic is considered a valid action, and just the formula remains the same after the application.

### State

A state should include all the information that makes a difference in the decision-making. In the tactic selection problem, we identify two main sources of such information: the current (transformed) formula $\varphi_t$ and the solving history statistics. In the specific design, the state at time step $t$ is represented as:

$$s_t = (\varphi_t, r_t, \boldsymbol{H_t})$$

where, $r_t$ is the solver running wall time from the beginning until the current step, and $\boldsymbol{H_t}$ is the history of the prior actions $[a_0, a_1, ..., a_{t-1}]$.

The episode ends when a terminal state is reached. In this formulation, a state is a terminal one when (1) the current formula, $\varphi_t$, is trivially solved, either satisfiable or

unsatisfiable; or (2) the accumulated solving time, $r_t$ exceeds the episode timeout, $T_{eps}$. We call the first case a winning terminal state, and the second case a losing one.

#### Reward and Return

The goal of the tactic selection problem is to sequentially make the best tactic choice so that the instance is solved the most efficiently. This goal should be formalized in terms of maximizing an episode return $G$, which is the sum of step-wise numerical rewards $R_t$. In this formalization, the reward will be given only at the end of the episode. If the instance is solved within the time limit, the agent will receive a positive reward; otherwise, the agent will receive a negative reward:

$$R_t = \begin{cases} 1 - \frac{r_t(1-R_m)}{T_{eps}}, \text{ when } s_t \text{ is a winning terminal state} \\ 0, \text{ when } s_t \text{ is not a terminal state} \\ -1, \text{ when } s_t \text{ is a losing terminal state} \end{cases}$$

where $R_m$ specifies the minimum positive reward in a winning state (in this research we pick 0.5), and $\frac{r_t(1-R_m)}{T_{eps}}$ is a time penalty. Generally, the longer time the agent takes to solve the formula, the less reward it will receive at the end.

In summary, the tactic selection problem is formalized as an agent learning to select the best tactic in each step of the SMT solving process, in order to solve each formula effectively and efficiently. The agent learns by actively trying various tactics on different formulas in different situations.

### 3.1.2 Characteristics of the Tactic Selection Problem

After the tactic selection problem has been clearly framed as an MDP, it can be solved using reinforcement learning techniques. There exist tons of reinforcement learning algorithms, each is effective for certain sorts of problems. This section carefully examines the nature and features of the tactic selection problem, for the purpose of choosing the most appropriate reinforcement learning algorithm.

A list of the problem features is as follows:

- Associativity refers to whether the current situation affects the optimality of actions. The tactic selection problem is an associative task, given that the most basic assumption of the problem is that tactics that work best for one type of instance may not work well for others. The goal of the tactic selection problem is to learn a policy that maps the current solving situation to the best tactic.

- Tactic selection is a sequential decision-making problem, where actions have not only immediate but also long-term consequences. There is a class of problems in reinforcement learning called multi-armed bandits (MAB) problems, where the action only has immediate rewards. It is not suitable for the tactic selection problem to be modeled as an MAB problem, because every tactic in the sequence transforms the formula, and all of them together determine the final solving result and time. It is impossible to separate the influences between steps.

- The tactic selection problem is episodic. The process of solving one query, from the start to the end, is viewed as one episode. Since a total timeout is defined for the whole solving process, all episodes eventually terminate no matter what actions are selected. Moreover, every episode usually deals with a different instance. Thus, we can consider that each episode starts with a sample from a starting state set.

- In the MDP modeling, the state includes the current formula and some prior solving statistics. Since there are infinite possible SMT queries, the state set for the tactic selection problem is infinite.

- The number of relevant tactic options for a formula is relatively large. The action space gets even much larger when we consider various parameter combinations for a tactic. It usually takes multiple steps to solve an instance.

- A base solver, which executes the selected tactic, is an essential part of the environment, and can be used as a model to generate simulations. Depending on the base solver, this model can be stochastic. One key difficulty of solving the tactic selection problem is that the simulation transition from state to state may be slow because the tactic application sometimes takes a very long time.

Overall, the tactic selection problem is a challenging reinforcement learning problem. We cannot model it as a simplified MAB problem, because every tactic has long-term impacts. The decision is associated with the current state and each episode starts with a different starting state, i.e., the target instance. The infinite state space requires the use of function approximation methods, both for prediction and for control. The large state

space and the large action space together create a vast search space, making exhaustive search methods infeasible. Exploring and exploiting more wisely are needed. A model for simulating experiences, i.e., a base SMT solver, is available for the problem. However, the slow transition between states prohibits the use of planning methods at solver run-time. Planning methods can, however, be used in the training phase.

## 3.2 Reinforcement Learning Framework

`AlphaSMT` mainly adopts a reinforcement learning framework that integrates Monte Carlo Tree Search (MCTS) and neural networks. The framework was introduced by DeepMind to train a computer Go in [34]. Key changes have been made to adapt the `AlphaZero` framework to the tactic selection problem. First, logical reasoning, i.e., a base SMT solver, is used to provide feedback, in the means of state transition and final reward. Second, because SMT solving is a time-critical task, MCTS, as the lookahead search, is used in the training stage only, but not in validation or at solver runtime. Third, the feature engineering (e.g., action embedding) and reward mechanism (e.g., winning and losing conditions) have been carefully redesigned to suit the needs of the tactic selection problem. The architecture of the neural network has also been dramatically altered, e.g., the addition of the transformer. Lastly, the framework is modified for the single-agent tactic selection problem instead of the two-player board game, Go.

In the following parts, Section 3.2.1 presents an overview of the proposed RL training algorithm, and Section 3.2.2 and Section 3.2.3 give a more detailed introduction to the two main components of the algorithm, i.e., Monte Carlo Tree Search and Deep Neural Network, respectively. Section 3.2.4 describes how, after training, the agent works at solver runtime.

### 3.2.1 Overview of the Reinforcement Learning Algorithm

The objective of our RL framework is to train an agent which makes good tactic choices at each step of solving a particular instance. The agent is trained over benchmarks that are representative of the future benchmarks of interests. The proposed RL framework for the tactic selection problem is summarized in Figure 3.2.

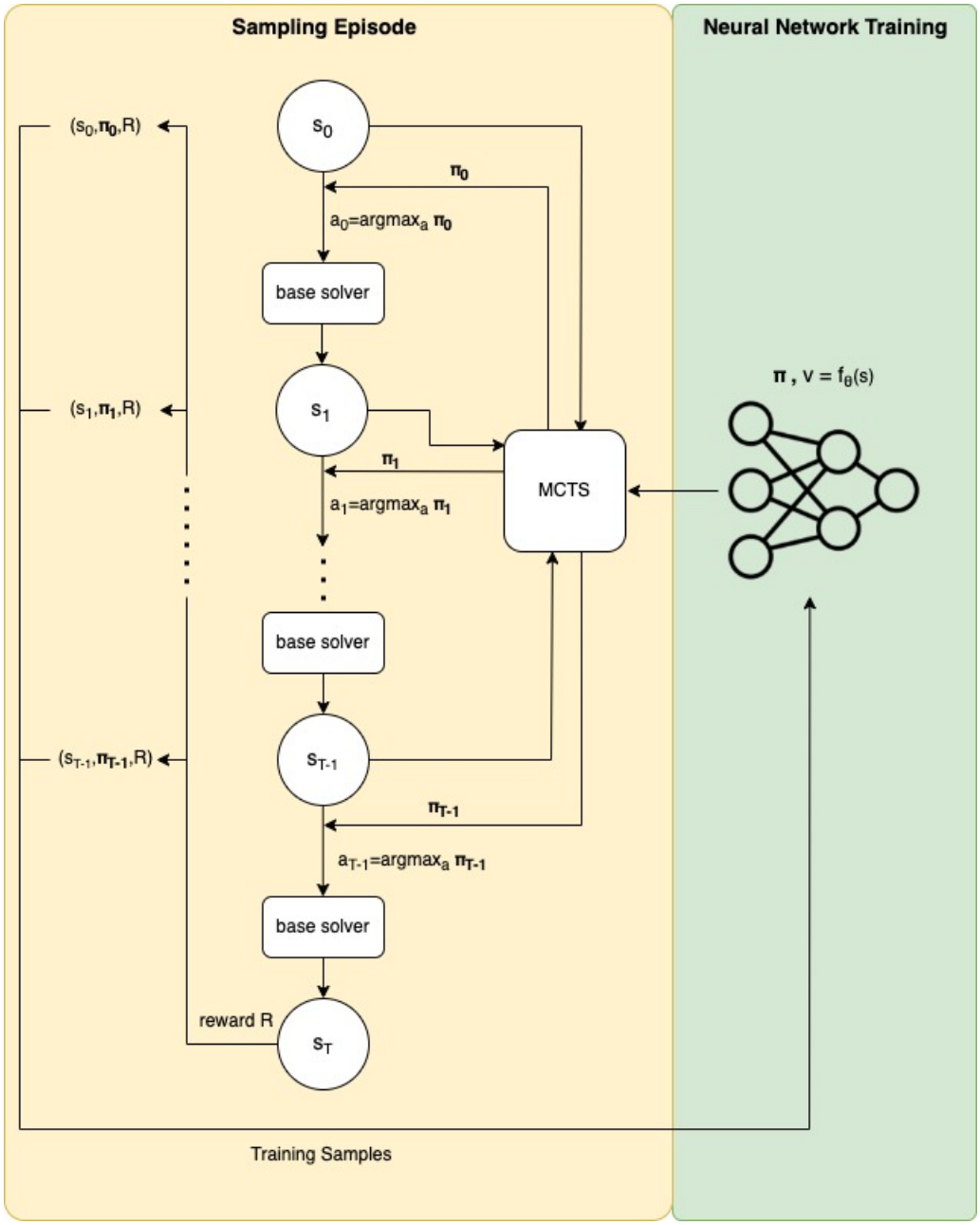



Figure 3.2: Reinforcement learning training framework in `AlphaSMT`

The key to reinforcement learning is a generalized policy iteration loop, which consists

of two processes, i.e., policy evaluation and policy improvement. In each loop, the value function is driven towards an evaluation of the policy, and the policy is then improved based on the value function. Through iteratively learning from experiences of agent-environment interaction, both the policy and the value function stabilize towards the optimal ones. In this framework, the policy and the value function are combined into a neural network with parameters $\theta$, $f_\theta$. The neural network $f_\theta$ is a function approximation of both the policy, a mapping from state to action (tactic) probability distribution, and the value function, a mapping from state to state value estimation. Specifically, $\boldsymbol{p}, v = f_\theta(s)$, where $\boldsymbol{p}$ is the action probability distribution at $s$, and $v$ is the value estimation, both at $s$.

The goal is to train a neural network $f_{\theta^*}$ that approximates the optimal policy and value function. The training happens by iterations. In each iteration, multiple sampling episodes are executed to collect neural network training samples. Each episode works on one formula $\varphi$ picked from the training benchmark set and sequentially applies tactics until the formula is solved, or the episode time exceeds a threshold timeout. Optionally, a time iterative deepening scheme is adopted. The tactic applied at each step is selected by a lookahead search algorithm, MCTS. See Section 3.2.2 for a more detailed description of the MCTS algorithm. The episode receives a final reward $R$ at the end, according to the terminal state status described in Section 3.1.1. This reward would be used as the sample return for all steps in the episode. Specifically, at the end of an episode, for every step $t$ in this episode, one neural network sample $d_t = (s_t, \pi_t, R)$ is generated, where $s_t$ is the state representation, and $\pi_t$ is the tactic probability distribution output by the MCTS. When all sampling episodes of an iteration end, the neural network $f_\theta$ will be trained upon the training samples from all episode steps. After the training, a validation step follows: the neural network $f_{\theta_i}$ from iteration $i$ will compete with the incumbent winner neural network from the previous iterations, over the validation benchmark set. The winner is the one that solves more instances in the validation set, or the one that solves faster if the two policies solve exactly the same number of instances. This criterion is also the one adopted in the SMT-COMP. Only the winning neural network will be used to guide future explorations. The purpose of this validation step is to ensure steady policy progress through learning.

The rationale behind the whole RL framework is as follows: MCTS, which looks ahead and makes better selections, is a policy improvement step; the sampling episodes, which enable the neural network to make better value predictions, are doing the policy evaluation, based on the logic feedback provided the base SMT solver. Thus, each training iteration is considered as one policy iteration loop. Over iterations, the neural network is expected to be closer to the optimal policy and the optimal value function.

### 3.2.2 MCTS Search Algorithm

MCTS is a decision-time planning algorithm, which helps make a better decision at the current environment state through simulations. Simulated trajectories all begin at the current environment state, and action values are estimated by averaging the returns of all traversing trajectories. The accumulated action value estimates from past simulations will guide future simulations toward more rewarding trajectories.

Specifically in this framework, MCTS serves as a lookahead search at each step $t$ of the sampling episodes. MCTS takes two inputs, i.e., the current state $s_t$ and the neural network $f_\theta$ with parameters $\theta$ from the previous iteration. By accumulatively collecting and updating value estimations from simulations starting at $s_t$, MCTS outputs a probability distribution recommending tactics to apply at $s_t$, $\pi(\cdot|s_t) = \alpha_\theta(s_t)$.

MCTS itself may be viewed as a self-play algorithm, exploring and exploiting different combinations of tactics in a search tree. MCTS incrementally builds and expands a tree rooted at $s_t$ through simulation runs. In the search tree, every state node $s$ has edges $(s, a)$ for all legal tactics $a \in A(s)$, and every edge $(s, a)$ stores a set of statistics:

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\},$$

where $N(s,a)$ is the visit count, $W(s,a)$ is the sum of action-values, $Q(s,a)$ is the mean action-value, and $P(s,a)$ is the prior probability of selecting $a$ at $s$. The search tree information accumulates through all steps in a sampling episode, but is not shared between iterations.

Every simulation run consists of four stages: select, expand, evaluate, and backup.

#### Select

Each simulation starts from the root node $s_t$, and traverses a path in the search tree until reaching a leaf node $s_L$ at time-step $L$. At every step $t'$ between $t$ and $L$, the transition is made by a tree policy that chooses the action with the highest upper confidence bound value $Q(s_{t'}, a) + U(s_{t'}, a)$ [28],

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

where $c_{puct}$ is a constant controlling the exploration level. This tree policy initially favors actions with high prior probability $P(s,a)$ and encourages exploration (where visit count

$N(s, a)$ is low), but asymptotically the policy prefers actions with high value estimation $Q(s_{t'}, a)$.

When the simulation trajectory reaches the leaf node $s_L$, the in-tree traverse ends, and $s_L$ is considered to be selected.

### Expand

If the selected leaf node $s_L$ is not a terminal one, the search tree is expanded by adding the edges $(s_L, a)$, for all legal action $a$ at $s_L$.

### Evaluate

Unlike traditional MCTS methods that use a rollout strategy, the proposed algorithm evaluates the expanded edges $(s_L, a)$ using the input neural network $f_\theta$. The neural network will evaluate the selected leaf node $s_L$, and outputs $p, v = f_\theta(s_L)$ (see Section 3.2.3 for more info on the neural network), where $p$ is the prior probability distribution of tactics and $v$ is the estimated value at $s_L$. Then the newly expanded edge $(s_L, a)$ would be initialised to

$$\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a\}$$

### Backup

The neural network output $v$ would be considered as the final return of the simulated trajectory (if the simulation reaches a terminal state, the real final reward $R$ will be used instead). This value would be backed up to update the action value of all the edges that traversed by the simulation path prior to the simulation reaching $s_L$, i.e., $(s'_t, a'_t)$ for $t < t' < L$. Every simulation run is considered as a Monto Carlo trial. Specifically, all edge statistics are updated as:

$$\begin{aligned} N(s'_t, a'_t) &:= N(s'_t, a'_t) + 1 \\ W(s'_t, a'_t) &:= W(s'_t, a'_t) + v \\ Q(s'_t, a'_t) &:= \frac{W(s'_t, a'_t)}{N(s'_t, a'_t)} \end{aligned}$$

Every simulation runs through these four steps, and the MCTS runs a user-defined number of simulations. After all simulations are completed, the MCTS outputs $\pi = \alpha_\theta(s_t)$, a probability distribution of tactics at $s_t$. The probability of each tactic is proportional to its exponentiated visit count:

$$\pi(a|s_t) = \frac{N(s_t, a)^{1/\gamma}}{\sum_b N(s_t, b)^{1/\gamma}}$$

where $\gamma$ is a temperature parameter controlling the level of exploration.

The MCTS output $\pi$ is considered an improved policy over the prior distribution $p$, which is predicted by the neural network.

### 3.2.3 Deep Neural Network

The deep neural network $f_\theta$ with parameters $\theta$ takes as an input a state embedding $x$, and outputs a tactic probability distribution $\boldsymbol{p}$ and a value estimation $v$, i.e., $\boldsymbol{p}, v = f_\theta(x)$.

The neural network input $x$ is an embedding of the state $s$. As described in Section 3.1.1, at time step $t$, $s_t = (\varphi_t, r_t, \boldsymbol{H_t})$, where $\varphi_t$ is the formula at step $t$, $r_t$ is the wall time past since the start of the episode, and $\boldsymbol{H_t}$ is the prior action/tactic history. Then, $x_t$ is derived from $s_t$:

$$x_t = (\boldsymbol{M_t}, r_t, \boldsymbol{H_t^f})$$

where, $\boldsymbol{M_t}$ is a vector of feature statistics describing the current formula $\varphi_t$, such as the number of expressions, the number of non-Boolean constants; $\boldsymbol{H_t^f}$ is a fixed-size prior action list, recording the most recent $N$ tactic attempts in sequence, where $N$ can be specified (padding will be applied if the current history is shorter than $N$).

The neural network output $\boldsymbol{p}$ is a probability distribution, representing the probability of selecting each tactic at state $s$; $v$ is a scalar, estimating the expected return from $s$. The neural network is an integrated policy and value function, which maps the state to its tactic selection recommendation and value estimation.

#### Architecture

The numerical part of the input, $\boldsymbol{M_t}$ and $r_t$, are first concatenated and scaled into an input vector $I_1$ with the size of $(|Tactics|+1)$. Then $I_1$ is connected to a hidden layer $L_{1,1}$. For the

embedding of the recent prior action vector $\boldsymbol{H_t^f}$, each action $a$ in $\boldsymbol{H_t^f}$ is encoded as a vector of size $E$, and a Transformer is used to encode the time series relationship among them. Vector embedding is a very popular technique in the natural language processing (NLP) field, which is usually used to encode words. This technique is adopted to encode tactics in this research, in an effort to gain a deeper understanding of the "natural language of tactics" [4, 38]. We also use transformers [40], another technique prevalent in the NLP field, to capture the time series relationship between tactic applications. Transformer is a deep-learning model that processes sequential input data. It uses a self-attention mechanism to deferentially weight each part of the input, in our case, each tactic $a$ in $\boldsymbol{H_t^f}$. Thus, $\boldsymbol{H_t^f}$ is encoded by the action embedding and the transformer into a vector $I_2$ of size $E \times N$, where $E$ is the tactic embedding size and $N$ is the size of $\boldsymbol{H_t^f}$. Subsequently, $I_2$ is connected to a hidden layer $L_{1,2}$. The concatenation of the outputs of $L_{1,1}$ and $L_{1,2}$ is then connected to another hidden layer $L_2$. Every layer applies dropout, batch normalization [12], and ReLU activation.

The output of $L_2$ is passed into two separate "heads" for computing the policy $\boldsymbol{p}$ and the value $v$, respectively. The policy head applies a fully connected linear layer that outputs a vector of size equal to the number of candidate tactics $|Tactics|$ and a `softmax` function. The value head uses a fully connected linear layer to a scalar, and a `tanh` function outputting a value in $[-1, 1]$.

Figure 3.3 shows the architecture of the deep neural network proposed in this research.

### Training Pipeline

As described earlier, in every sampling episode $e$, one training sample, $d_{e,t} = (s_t, \pi_t, R)$ will be recorded at each time step $t$, where $\pi_t$ is the tactic probability distribution output by MCTS, and $R$ is the episode final reward. Thus, in every iteration $i$, the training dataset $D_i$ consists of all $d_{e,t}$, where $e$ is a sampling episode within the iteration $i$, and $t$ is a step within the episode $e$.

Noted that if a formula has never been solved in previous attempts, either in MCTSs or in sampling episodes, the training samples from its associated episodes will not be used for neural network training. The reason is that there is nothing to learn if a formula is unsolvable by any combination of the available tactics. Including samples from unsolvable formulas can discourage potentially good strategies.

In each iteration, the aim of the neural network training is to adjust the neural network parameters $\theta$, so that the differences between the neural network predicted $\boldsymbol{p}$ and the MCTS output $\pi$, and between the predicted $v$ and the recorded $R$, are minimized. Specifically,

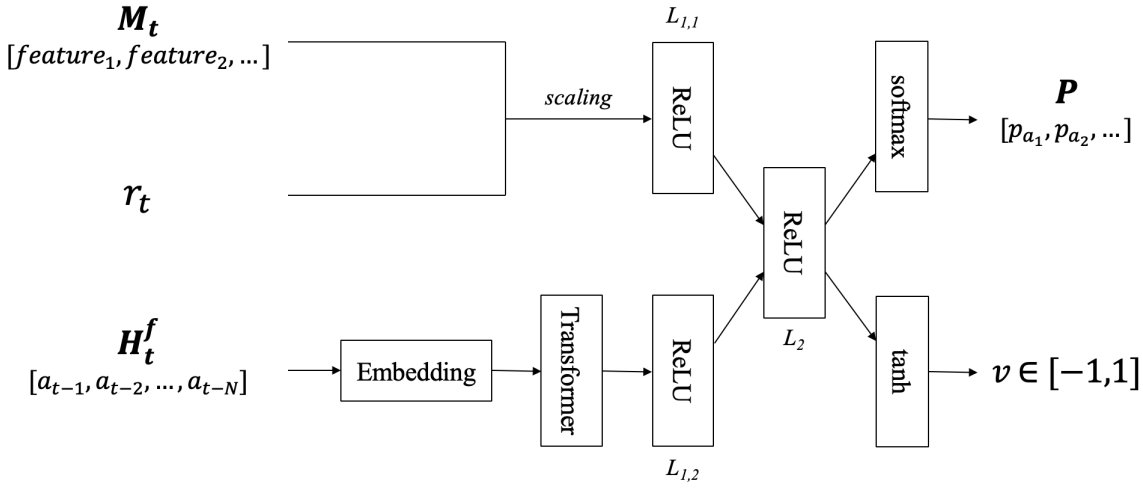



Figure 3.3: Deep neural network architecture

gradient descent is used to tune the parameters $\theta$ to minimize a loss function $l$ over the training dataset. The loss function $l$ is the sum of a mean square error of the value estimation, a cross-entropy of the policy, and a regularization term:

$$l = (z - v)^2 - \boldsymbol{\pi}^\top \log \boldsymbol{p} + c\|\theta\|^2$$

where $c$ is a parameter controlling the level of L2 weight regularization.

### 3.2.4 Runtime Design

Lookahead search is not applied at runtime. At each time step $t$, the trained neural network $f_\theta$ serves as the policy, outputting a tactic probability $\boldsymbol{p}$. The tactic with the highest probability is chosen for $a_t$. This runtime framework is shown in Figure 3.4.

Specifically, upon each formula $\varphi_0$ to solve, the neural network $f_\theta$ iteratively selects the expected best tactic, until either the formula is solved or the whole solving process times out. Starting from the initial state $s_0 = (\varphi_0, 0, ())$, AlphaSMT derives $x_0 = (M_0, 0, \boldsymbol{H_0'})$ from $s_0$, and calculates $\boldsymbol{p}$ as $\boldsymbol{p}, v_{s_0} = f_\theta(x_0)$. The tactic at the first step, $a_0$, is selected as $a_0 = \text{argmax}_{a \in Tactics}\, p_a$. A base solver then applies $a_0$ to $\varphi_0$, causing the state transition

into $s_1 = (\varphi_1, r_1, (a_0))$. If $s_1$ is not terminating, a tactic $a_1$ is then picked by the probability output of $f_\theta(x_1)$, and the state moves into $s_2$. This process of selecting a tactic will repeat until a terminal state, $s_T$, is reached.

Furthermore, there is always an overhead cost to extract the state features and call the neural network. Some easy instances can be quickly solved, regardless of the choice of tactic combinations. Thus, we optionally use a pre-solver scheme, i.e., trying a selected pre-solver on the instance for some short time period, before incurring the neural network. This scheme helps save the overheads for easy instances.

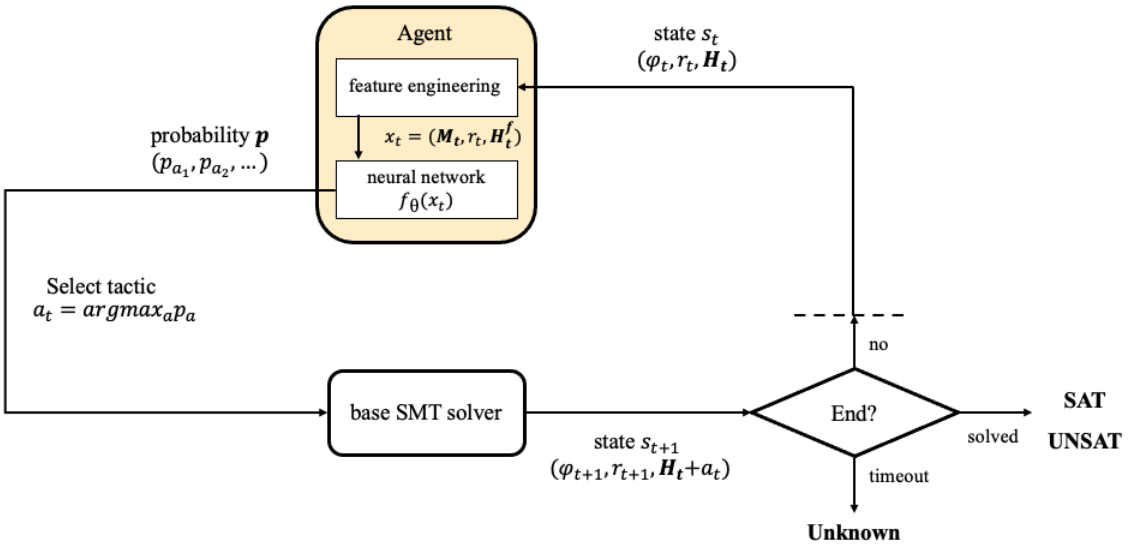



Figure 3.4: `AlphaSMT` at runtime

# Chapter 4

# Experiment Design and Results

## 4.1 AlphaSMT Specifications

AlphaSMT[1] is an adaptive RL-guided SMT solver, built based on the framework proposed in Section 3.2. For each instance, AlphaSMT sequentially and adaptively selects the expected best tactic to apply, for the purpose of efficiently solving the satisfiability of the instance.

AlphaSMT is implemented in Python 3.7. It interacts with the base solver Z3 using Z3 Python APIs. Actions are selected from Z3 built-in tactics, and formula features are calculated by Z3 built-in probes [7]. The implementation of the RL framework has greatly benefited from the work of [39]. Within the framework, the neural network engine part is built with the assistance of PyTorch.

AlphaSMT has a pre-training stage. The general procedure is described in Section 3.2. The decision agent of AlphaSMT, a neural network, is trained over iterations. In each iteration, training samples are collected from running sampling episodes, one episode for one formula in a training benchmark set. Episodes in one iteration are executed in parallel, and the neural network is trained after all episodes are complete. At the end of each iteration, the trained neural network is validated on a validation benchmark set to ensure progress. Due to practical reasons, we set a step limit $K_{max}$ for sampling episodes: if an instance is not solved within $K_{max}$ tactic attempts, the episode will end and receive a losing reward. The reason is that an episode may take an extremely large number of steps if the only losing condition is timeout. Some tactics return very quickly. Since MCTS is

[1]The solver source code, experiment settings, and result datasets are available at https://github.com/JohnLyu2/AlphaSMT.

executed at each step, without a step limit, some episodes would take an extremely long time to complete.

The general architecture of the deep neural network is described in Section 3.2.3. An embedding size of 128 is used to encode the tactic actions. The sizes of both $L_{1,1}$ and $L_{1,2}$ are 64, and the size of $L_2$ is 32. A dropout rate of 0.3 is used.

At runtime, the user can choose to run the pre-solver, `Z3`, for a specified time period. Such a pre-solving period helps reduce the overheads caused by the feature extraction and neural network for relatively easy instances.

## 4.2 Experiment Setup

`AlphaSMT` is evaluated over three benchmark sets, i.e., `CInteger`[2], a quantifier-free non-linear integer arithmetic (QF_NIA) set, `LassoRanker`[3], a quantifier-free non-linear real arithmetic (QF_NRA) set, and `Sage2`[4], a quantifier-free bit-vector (QF_BV) set. These three datasets all come from the official SMT-LIB benchmark library [2]. Table 4.1 summarizes their statistics.

[2]https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/-/tree/master/20170427-VeryMax/CInteger

[3]https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA/-/tree/master/LassoRanker

[4]https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/tree/master/Sage2

Table 4.1: Experimental Benchmark Dataset Statistics

| | | `CInteger` | `LassoRanker` | `Sage2` |
|---|---|---|---|---|
| Logic | | QF_NIA | QF_NRA | QF_BV |
| Size(#Instances) | | 1818 | 821 | 7602 |
| #Assertions | Ave. | 26.8 | 18430.1 | 330.5 |
| | Max. | 167 | 1539363 | 3944 |
| | Min. | 5 | 197 | 1 |
| #Expressions | Ave. | 1439.1 | 67300.4 | 1845.3 |
| | Max. | 69582 | 4966570 | 168350 |
| | Min. | 98 | 772 | 52 |
| #Variables | Ave. | 154.9 | 10298.3 | 278.3 |
| | Max. | 7399 | 868840 | 14155 |
| | Min. | 12 | 110 | 2 |

We divided each benchmark set into training, validation, and testing sets. We select 100 instances for training and 100 for validation from each benchmark set. These instances are randomly selected from a relatively hard subset of the whole benchmark set. All the remaining instances constitute the testing set (due to the large size of `Sage2`, we only randomly picked 1000 instances from the remaining set for testing). In each iteration, the neural network is trained on the data generated from the training instances, and validated on the validation instances. After all the training iterations, the trained neural network will act as the core of `AlphaSMT`, to be evaluated on the testing set.

All the experiments use a 300-second episode timeout. For each experiment, the training iterates 10 times. In each iteration, we ran one sampling episode for each instance in the training set. The action space $Tactics$ and the selected formula features $M$ (used for the neural network input) for experiments are shown in Table 4.2 and Table 4.3, respectively. We design experiments with different fixed tactic timeouts (45s, 60s, 90s), and with the iterative time deepening scheme. The window size of $\boldsymbol{H^f}$ for all experiments is set to

6. After training, the AlphaSMT solver is tested both with and without a pre-solver. For each benchmark set, the performance of AlphaSMT solvers with different configurations are compared with the performance of the original Z3. All parameters in Z3 (both for the base solver in AlphaSMT and the original one for comparison) are set to default. We do not run tests with different Z3 random seeds since empirical analyses show that random seeds (both in the module *sat* and *smt*) have negligible effects on solving performance in terms of these tested benchmark sets.

Table 4.2: Candidate Tactics (Action Space) for Experiments in Different SMT Logics

| **Logic** | **Candidate Tactics** |
|---|---|
| QF_NIA | simplify, smt, bit-blast, propagate-values, ctx-simplify, elim-uncnstr, solve-eqs, qfnia, lia2card, max-bv-sharing, nla2bv, qfnra-nlsat, cofactor-term-ite |
| QF_NRA | simplify, smt, bit-blast, propagate-values, ctx-simplify, elim-uncnstr, solve-eqs, qfnra, lia2card, max-bv-sharing, nla2bv, qfnra-nlsat |
| QF_BV | simplify, smt, bit-blast, bv1-blast, solve-eqs, aig, qfnra-nlsat, sat, max-bv-sharing, reduce-bv-size, purify-arith, propagate-values, elim-uncnstr, ackermannize_bv, qfbv |

Table 4.3: Formula Features $M$ included in the Neural Network Inputs for Experiments in Different SMT Logics

| Logic | Formula Features (Z3 Probes) |
|---|---|
| QF_NIA | `is-unbounded`, `arith-max-deg`, `arith-avg-deg`, `arith-max-bw`, `arith-avg-bw`, `is-qfnra`, `is-qfbv-eq`, `memory`, `size`, `num-exprs`, `num-consts`, `num-bool-consts`, `num-arith-consts`, `num-bv-consts`, `is-propositional`, `is-qfbv` |
| QF_NRA | `is-unbounded`, `arith-max-deg`, `arith-avg-deg`, `arith-max-bw`, `arith-avg-bw`, `is-qfnia`, `is-qfbv-eq`, `memory`, `size`, `num-exprs`, `num-consts`, `num-bool-consts`, `num-arith-consts`, `num-bv-consts`, `is-propositional`, `is-qfbv` |
| QF_BV | `is-pb`, `is-qflia`, `memory`, `size`, `num-exprs`, `num-bool-consts`, `num-bv-consts`, `is-propositional` |

The training and testing tasks were run on the Graham cluster provided by the Digital Research Alliance of Canada (alliancecan.ca). 15 GB of memory per node is requested for all tasks.

## 4.3 Experimental Results in QF_NIA

Table 4.4 summarizes the experimental results on the QF_NIA benchmark set `CInteger`. The solving time statistics are based on instances that can be solved by all solvers in the table (same for the results in Table 4.5). Figure 4.1a and Figure 4.1b show the results of the solver without and with a 10-second `Z3` pre-solver, respectively. All the listed versions of `AlphaSMT` solvers are trained on a 100-instance set, validated on a 100-instance set, and evaluated on a 1618-instance set.

It is obvious that all `AlphaSMT` solvers perform significantly better than the default `Z3` solver. `AlphaSMT` solvers solve around 14% more instances than `Z3`. Among the `AlphaSMT` solvers with different configurations, the solver with a 90-second tactic timeout and a 10-second pre-solver time solves the most instances (1286/1618 instances); however, the lead ahead of other AlphaSMT solvers is minor (within 2% of the total testing instances). We

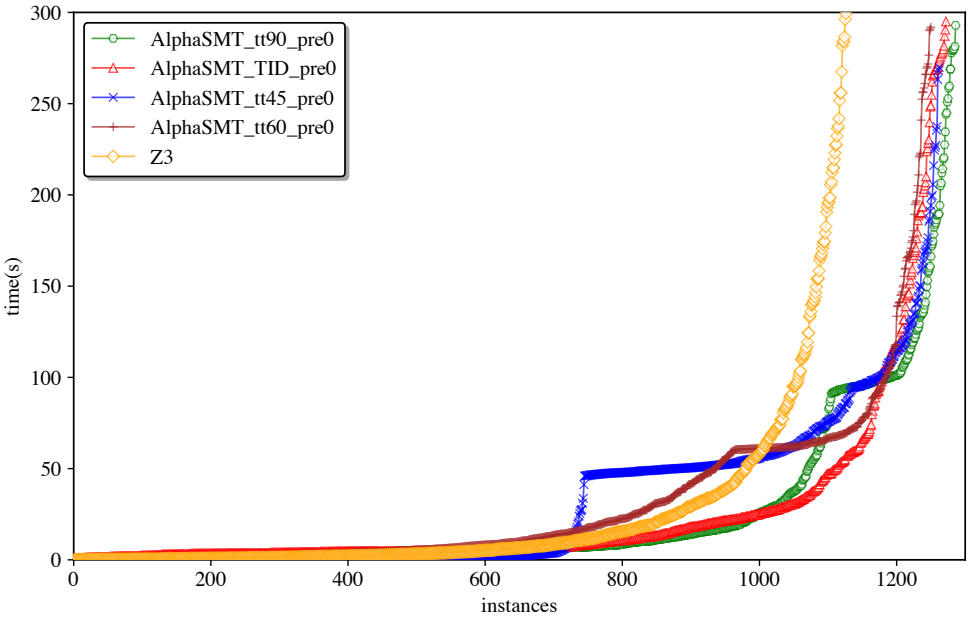
observed a trend that solvers with longer tactic timeout solve more formulas. Also, the time-iterative-deepening solvers are also among the best ones.

It also has been observed that the pre-solver tends to accelerate the solving process. The time-iterative-deepening `AlphaSMT` solver equipped with a 10-second pre-solver has the fastest average solving speed. The figures show that the lines for the fixed-tactic-timeout `AlphaSMT` solvers all have a zigzagged shape, while the lines for time-iterative-deepening solvers are smoother, which is an indication that the time-iterative-deepening scheme makes better use of time.

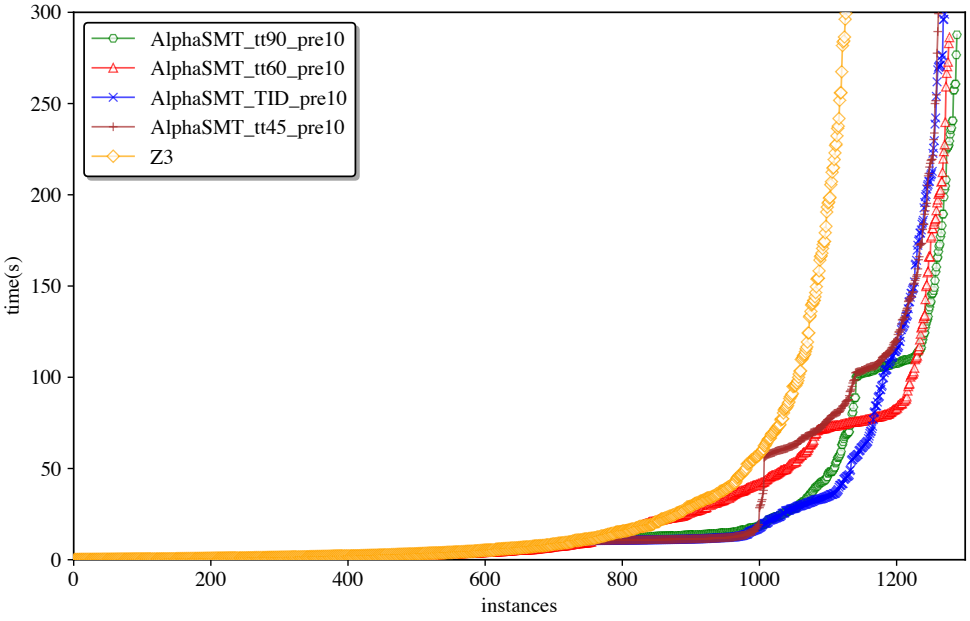
Table 4.4: Experimental Results on `CInteger` in terms of Formulas Solved and Solving Time

| | Formulas Solved | | Solving Time | | | |
|---|---|---|---|---|---|---|
| | Number | Percentage | Ave. (s) | Speedup Percentile against Z3 | | |
| | | | | $90^{th}$ | $50^{th}$ | $10^{th}$ |
| AlphaSMT_tt45_pre0 | 1262 | 78.0% | 28.4 | 0.04× | 0.63× | 22.47× |
| AlphaSMT_tt45_pre10 | 1261 | 77.9% | 19.4 | 0.35× | 1.02× | 2.64× |
| AlphaSMT_tt60_pre0 | 1250 | 77.3% | 21.4 | 0.28× | 0.63× | 1.69× |
| AlphaSMT_tt60_pre10 | 1277 | 78.9% | 17.1 | 0.73× | 1.17× | 1.55× |
| AlphaSMT_tt90_pre0 | 1286 | 79.5% | 18.8 | 0.08× | 0.56× | 6.85× |
| AlphaSMT_tt90_pre10 | 1288 | 79.6% | 15.9 | 0.77× | 1.15× | 2.43× |
| AlphaSMT_TID_pre0 | 1272 | 78.6% | 19.3 | 0.06× | 0.43× | 8.22× |
| AlphaSMT_TID_pre10 | 1270 | 78.5% | 15.5 | 0.67× | 1.02× | 3.03× |
| Z3 | 1126 | 69.6% | 23.0 | / | / | / |

**Note**: (1) the first column describes the solver configurations: `tt` refers to the tactic timeout and `TID` refers to the time-iterative-deepening tactic timeout scheme. `pre` denotes the pre-solver time. For example, `AlphaSMT_tt45_pre0` is the AlphaSMT solver that uses a tactic timeout of 45 seconds and applies no pre-solver; (2) the statistics of solving time are based on instances that can be solved by all listed solvers.




(a) no pre-solver




(b) 10-second pre-solver time

Figure 4.1: The cactus plot for the experimental results on the benchmark set `CInteger`

## 4.4 Experimental Results in QF_NRA

Table 4.5 summarizes the experimental results on the QF_NRA benchmark set `LassoRanker`. Same with the experiments on `CInteger`, all `AlphaSMT` solvers are trained on a 100-instance set and validated on a 100-instance set. There are 621 instances in the testing set.

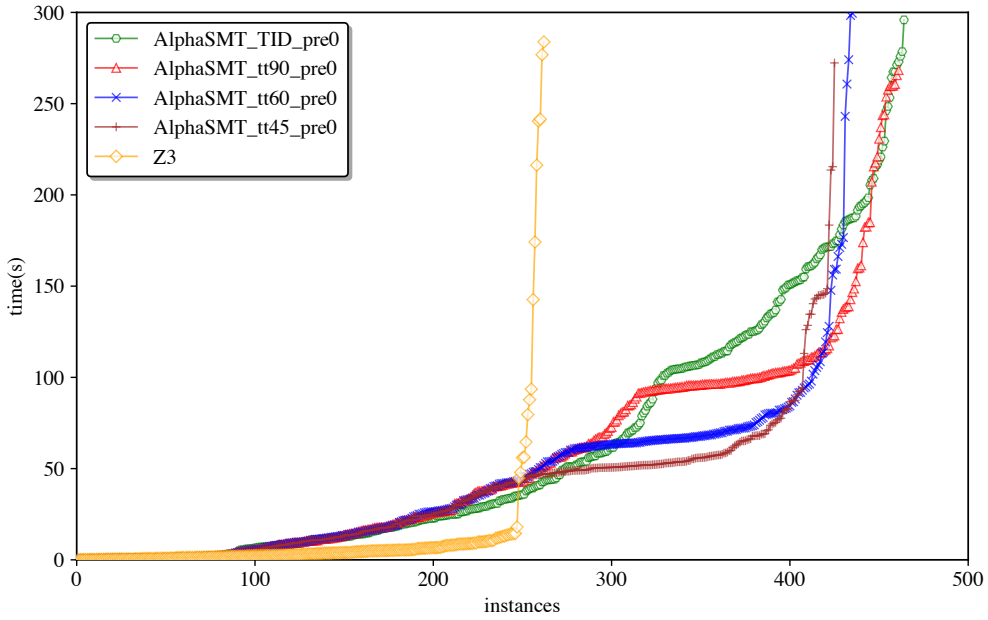
Table 4.5: Experimental Results on `LassoRanker` in terms of Formulas Solved and Solving Time

| | Formulas Solved | | Solving Time | | | |
|---|---|---|---|---|---|---|
| | Number | Percentage | Ave. (s) | Speedup Percentile against Z3 | | |
| | | | | $90^{th}$ | $50^{th}$ | $10^{th}$ |
| AlphaSMT_tt45_pre0 | 425 | 68.4% | 31.3 | 0.03× | 0.10× | 2.62× |
| AlphaSMT_tt45_pre10 | 428 | 68.9% | 11.3 | 0.31× | 0.84× | 0.99× |
| AlphaSMT_tt60_pre0 | 435 | 70.0% | 37.6 | 0.02× | 0.09× | 2.75× |
| AlphaSMT_tt60_pre10 | 445 | 71.7% | 12.8 | 0.27× | 0.81× | 0.97× |
| AlphaSMT_tt90_pre0 | 461 | 74.2% | 47.2 | 0.02× | 0.08× | 2.61× |
| AlphaSMT_tt90_pre10 | 471 | 75.8% | 5.0 | 0.94× | 1.09× | 1.30× |
| AlphaSMT_TID_pre0 | 464 | 74.7% | 51.5 | 0.02× | 0.11× | 2.30× |
| AlphaSMT_TID_pre10 | 473 | 76.2% | 7.6 | 0.81× | 1.03× | 1.27× |
| Z3 | 262 | 42.2% | 11.4 | / | / | / |

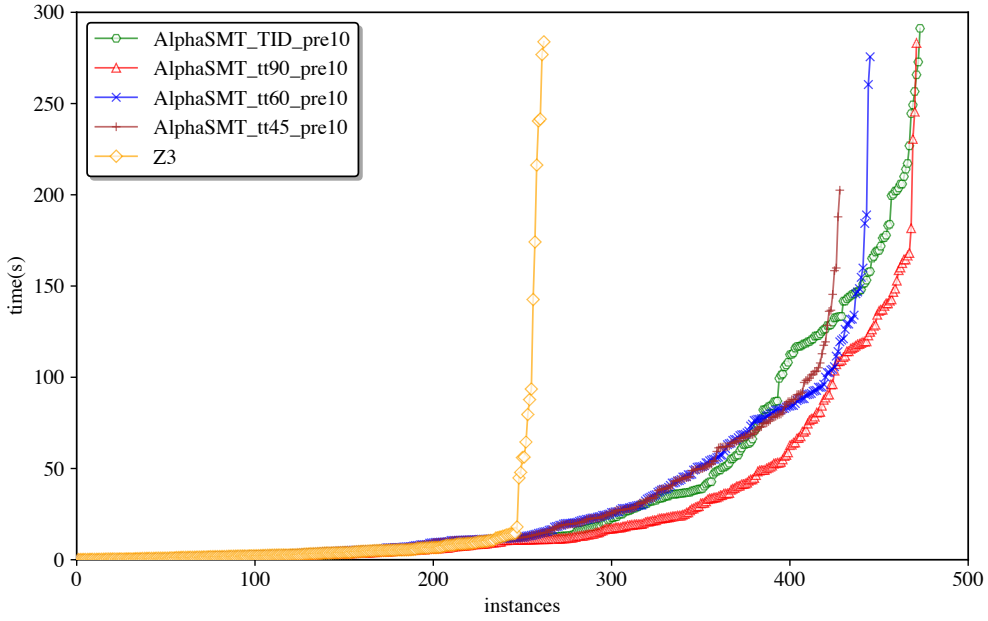
**Note**: (1) the first column describes the solver configurations: `tt` refers to the tactic timeout and `TID` refers to the time-iterative-deepening tactic timeout scheme. `pre` denotes the pre-solver time. For example, `AlphaSMT_tt60_pre10` is the AlphaSMT solver that uses a tactic timeout of 60 seconds and a pre-solver time of 10 seconds; (2) the statistics of solving time are based on instances that can be solved by all listed solvers.

The general patterns are similar to but more exaggerated than the ones observed in the QF_NIA logic. In this harder benchmark set, the best `AlphaSMT` solver solves 76.2% (473/621) of the total testing instances, while `Z3` can only solve 42.2% (262/621). Figure 4.2a and Figure 4.2b compare the performance of all `AlphaSMT` solvers and `Z3`. We first observed that very few instances are solved by `Z3` after 20 seconds, while for `AlphaSMT`,

the turning points appear much later. We have also observed that longer tactic timeout helps solve more benchmarks and the pre-solver significantly expedites the average solving speed. The `AlphaSMT` solver with the time-iterative-deepening scheme and the pre-solver solves more instances than all other solvers with a competitive speed.




(a) no pre-solver




(b) 10-second pre-solver time

Figure 4.2: The cactus plot for the experimental results on the benchmark set `LassoRanker`

## 4.5 Experimental Results in QF_BV

For the QF_BV benchmark set `Sage2`, we compared the performance of `AlphaSMT` and `Z3` on a 1000-instance testing set. The `AlphaSMT` solver under test has a tactic timeout of 90 seconds and a pre-solving time of 10 seconds. Figure 4.3 shows the evaluation results. We found that `AlphaSMT` solves more instances than `Z3` (`AlphaSMT`: 753/1000; `AlphaSMT`: 727/1000), while `Z3` solves faster on average (for instances that solved by both solvers, `AlphaSMT` average time: 38.9 seconds; `Z3` average time: 32.4 seconds). Due to time constraints, we only tested `AlphaSMT` for `Sage2` with one specific configuration. Fine-tuning `AlphaSMT` configurations could potentially boost its speed.

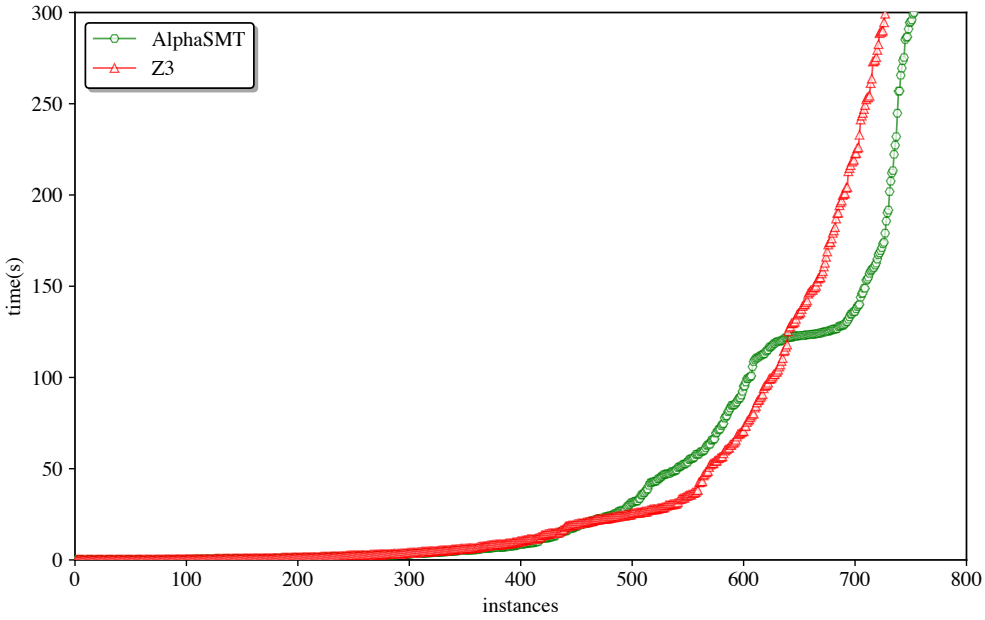



Figure 4.3: The cactus plot for the experimental result on the benchmark set `Sage2`

## 4.6 Tactic Selection Insights from `AlphaSMT`

In this section, we look into what exact strategies that `AlphaSMT` uses to solve testing benchmarks; these strategies help explain the performance of `AlphaSMT` and may provide tactic selection insight for practitioners. `AlphaSMT` uses a trained neural network to sequentially select a tactic to apply during the solving process. This tactic selection depends

on a combination of factors, i.e., the current formula features, the tactic application history, and the solving time already spent. Although it is hard to interpret all the neural network's decision criteria, we have extracted the *effective strategies* that `AlphaSMT` uses for each benchmark set, shown in Table 4.6. We define an *effective strategy* as a sequence of tactics that has successfully solved certain benchmarks in the test set, and each tactic in the sequence effectively rewrites the formula (tactics that fail or timeout are excluded).

Table 4.6: Effective Strategies used by `AlphaSMT` during Testing

| **Benchmark Set** | **Effective Strategies** |
|---|---|
| `CInteger` | `[qfnra-nlsat]`<br>`[qfnia]`<br>`[smt]`<br>`[solve-eqs, qfnia]` |
| `LassoRanker` | `[qfnra-nlsat]`<br>`[smt]`<br>`[propagate-values, smt]`<br>`[solve-eqs, smt]`<br>`[solve-eqs, qfnra-nlsat]`<br>`[propagate-values, solve-eqs, smt]`<br>`[propagate-values, solve-eqs, qfnra-nlsat]` |
| `Sage2` | `[qfnra-nlsat]`<br>`[smt]`<br>`[qfbv]`<br>`[purify-arith, qfbv]`<br>`[purify-arith, max-bv-sharing, solve-eqs, smt]` |

We found that there are not many unique effective strategies for each benchmark set: 4 for `CInteger`, 7 for `LassoRanker`, and 5 for `Sage2`. Also, every effective strategy is not long (at most 4 steps) and only a small number of tactics appear in all the effective strategies. The improved performance of `AlphaSMT` may stem from its ability to (1) sequentially choose the appropriate next tactic, and (2) adaptively switch to the next-best strategy after one fails or timeouts. We also observed that `AlphaSMT` uses much fewer tactics and

shorter strategies in the testing phase than in the training. The reason is that in testing all decisions are purely made by the trained neural network, while in training the decisions are made jointly by the neural network and the Monte Carlo Tree Search (MCTS) step. The MCTS helps find better next-step tactics tailored to the status quo formula by lookahead planning. However, such lookahead steps are too expensive to use in the testing phase, as the solving process is time critical. Our future emphasis would be to improve the runtime framework so that it can explore a larger and more complex space without the expensive lookahead costs.

## 4.7 Result Summaries

In the training phase, `AlphaSMT`'s performance on the validation set is usually dramatically improved in the first several iterations and becomes more stabilized in the later iterations. It shows that `AlphaSMT` is learning towards an optimal policy within and across training iterations. As for the testing results, `AlphaSMT` solves substantially more instances than its base solver `Z3`, for all the tested logics. This achievement is impressive since the RL only selects and orders the existing built-in tactics. The improvement is largest in the `QF_NRA` benchmark set: `AlphaSMT` solves 80.5% more instances in `LassoRanker` than `Z3`.

`AlphaSMT` configurations affect the performance; the difference is also more obvious in the `QF_NRA` results. In general, longer tactic timeout help solve more instances, and the time-iterative-deepening works well. This finding meets our expectations, because the tactic timeout constraints the power of each tactic. However, the differences in the number of solved formulas are not big among all `AlphaSMT` solvers, which suggests that all listed `AlphaSMT` solvers effectively explored the searching space during the training. We expect that `AlphaSMT` will work well on relatively hard benchmark sets as long as there exists improvement space in terms of tactic selection.

On the other hand, having a pre-solving stage definitely improves the solving speed, especially for easy instances. This outcome is expected, as the rationale behind the pre-solver is to save the overheads for simple instances. Furthermore, the `AlphaSMT` solvers were trained on relatively harder benchmarks, and the reward mechanism mainly encouraged solving more instances. They are not tailored for fast speed for easy instances. We also observed the zigzagged shape for lines of the fixed-tactic-timeout solvers in the cactus plots. Each turning point represents a shift of tactics after a tactic timeout. The time-iterative-deepening scheme helps smooth the curve and is expected to perform well once we have a longer or shorter total timeout for the solving process.

When looking into the exact strategies that `AlphaSMT` applies, we found that there are not many unique effective strategies used by `AlphaSMT`. The current performance supremacy of `AlphaSMT` over `Z3` may come from `AlphaSMT`'s ability of (1) sequentially choosing the appropriate next tactic, and (2) adaptively switching to the next-best strategy after one fails or timeouts. Future work will explore how to explore more strategy space at runtime.

# Chapter 5

# Conclusion and Future Work

## 5.1 Conclusion

This thesis research starts with raising the question of tactic selection: for each SMT instance to solve, how to craft a good strategy by sequencing tactics? To answer this question, we first formalize this problem as a reinforcement problem (RL) and propose to solve the problem using a RL framework that combines deep Monte-Carlo Tree Search (MCTS) and logical reasoning. Then, an SMT solver, `AlphaSMT`, is built based on this proposed framework. The evaluation of `AlphaSMT` on three SMT benchmark sets shows positive results that the RL guided strategy solves significantly more benchmarks than the default strategy.

The main findings and contributions of this research are as follows:

- This thesis formally brings up the tactic selection problem for SMT solving. The problem shares similarity with the algorithm selection problem, but with more flexibility: the solving strategy can be built by sequentially choose a tactic to apply, instead of only choosing one start-to-end algorithm.
- The tactic selection problem is rigorously formalized as a Markov Decision Process in the RL context. The key RL elements, e.g., agent, environment, action, state, reward, are clearly defined in the tactic selection problem. In essence, the formalization describes the iterative process of a decision agent interacting with a base SMT solver to learn how to solve the formula the most effectively and efficiently.

- We propose an RL framework that combines deep MCTS and logical reasoning for the tactic selection problem. In the framework, a deep neural network is served as both the value function and the policy, evaluating state-action pairs and making the tactic decision. The neural network is trained on training samples collected from sampling solving episodes. The sampling episodes use MCTS as a lookahead planning step, and receive feedback from the base SMT solver. After iteratively trained on a training benchmark set, the neural network is expected to make wise tactic choices for a specialized type of SMT tasks.
- An RL-guided adaptive SMT solver, `AlphaSMT` was built based on the proposed framework. We evaluated its performance on three benchmark sets: `CInteger`, a QF_NIA set, `LassoRanker`, a QF_NRA set, and `Sage2`, a QF_BV set. Solvers with various configurations (tactic timeout time, pre-solver time...) were tested. We found that `AlphaSMT` solved significantly more instances than its base solver `Z3`, with an improvement of 10% on `CInteger` and of 34% on `LassoRanker`. It was also observed that a reasonably longer tactic timeout helped solve more instances and a pre-solver setting accelerated the solving process for many easy instances.

## 5.2 Limitations

Due to time constraints, we have only tested `AlphaSMT` in the SMT logics of QF_NIA, QF_NRA, and QF_BV. We plan to test it on a larger scale covering more benchmark sets and logics in the future, to demonstrate its robustness and generality. Also, we only verbally argue our method's superiorness over `FastSMT`, but have not provided relevant experimental results to prove this argument. Such evaluation experiments are also on our to-do list.

## 5.3 Future Work

Reinforcement learning for tactic selection is a rich topic, and the current version of `AlphaSMT` is just the groundwork on this idea. The tactic selection problem can be viewed as a specific program synthesis problem, which aims to automatically build an algorithm for a given problem. The program synthesis problem has been considered a holy grail of the field of Computer Science and one of the most central problems in the theory of programming [26]. Advances in the tactic selection problem could potentially lend insights and significantly help the more general program synthesis problems.

Some specific future directions are as follows:

- `AlphaSMT` is currently built upon `Z3`, and only selects tactics provided by `Z3`. Combining tactics from different solvers seems to be a more powerful idea, and the problem could be considered a mixture of algorithm selection and tactic selection.
- There are some prefixed parameters in `AlphaSMT`, e.g., pre-solver time, the timeout structure in time iterative deepening. These parameters can also be learned during training and be picked dynamically upon inference.
- Now `AlphaSMT` is not an online solver, since the policy does not change at runtime. The current training process involving the MCTS lookahead step is too expensive to apply at solver runtime. We are exploring a new runtime design to make `AlphaSMT` an online solver, which also learns upon inference.
- The tactic selection problem is now modeled as a sequential decision problem. In this modeling, once a formula is transformed by a tactic, there is no means to get back to the original formula. For example, if an NLA formula is converted to a BV formula using the `Z3` tactic `nla2bv`, `AlphaSMT` cannot work on the NLA formula any longer. A different strategy structure could potentially provide more flexibility on this issue. Parallelism is another direction of future work.
- Currently, each tactic is treated as a "blackbox" and the reinforcement learning is not tied intimately to the solver's inner workings. Little internal information is shared between steps. For example, if a tactic is timed out, the formula remains the same and no solving information has been passed over; however, valuable insight could have been learned during this failed process. How to better leverage internal solver information is worth exploring.
- `AlphaSMT` now only works with big reasoning engines provided by `Z3`, such as Gaussian Elimination, Tseitin transformation, SAT solvers. Most of these reasoning steps are modularizable. Future work can explore working with lower-level aspects of the tactics or even creating new tactics.

# References


[1] Mislav Balunovic, Pavol Bielik, and Martin Vechev. Learning to solve smt formulas. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10337–10348. Curran Associates, Inc., 2018.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[3] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[4] Yash Chandak, Georgios Theocharous, James Kostas, Scott Jordan, and Philip Thomas. Learning action representations for reinforcement learning. In *International conference on machine learning*, pages 941–950. PMLR, 2019.

[5] Wenxiang Chen, Adele Howe, and Darrell Whitley. Minisat with classification-based preprocessing. *SAT COMPETITION 2014*, page 41, 2014.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[7] Leonardo De Moura and Grant Olney Passmore. The strategy challenge in smt solving. *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, pages 15–44, 2013.

[8] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Sat competition 2020. *Artificial Intelligence*, 301:103572, 2021.

[9] Vijay Ganesh, Sanjit A. Seshia, and Somesh Jha. Machine learning and logic: A new frontier in artificial intelligence. Personal communication, 2022.

[10] Sean B Holden et al. Machine learning for automated theorem proving: Learning to solve sat and qsat. *Foundations and Trends® in Machine Learning*, 14(6):807–989, 2021.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[13] Raihan H Kibria. Evolving a neural net-based decision and search heuristic for dpll sat solvers. In *2007 International Joint Conference on Neural Networks*, pages 765–770. IEEE, 2007.

[14] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Alogorithm Point of View*. Springer, 2016.

[15] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Improving sat solver heuristics with graph networks and reinforcement learning. *arXiv preprint arXiv: 1909.11830*, 2019.

[16] Chunxiao Li, Charlie Liu, Jonathan Chung, Piyush Jha, and Vijay Ganesh. A reinforcement learning based reset policy for cdcl sat solvers. Personal communication, 2023.

[17] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[18] Jia Hui Liang. Machine learning for sat solvers. *PhD Thesis*, 2018.

[19] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*, pages 123–140. Springer, 2016.

[20] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for cdcl sat solvers. In *Theory and Applications of Satisfiability Testing–SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*, pages 94–110. Springer, 2018.

[21] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. IOS press, 2021.

[22] Saeed Nejati. Cdcl (crypto) and machine learning based sat solvers for cryptanalysis. *PhD Thesis*, 2020.

[23] Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive restart and cegar-based solver for inverting cryptographic hash functions. In *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers 9*, pages 120–131. Springer, 2017.

[24] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[25] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A Seshia. Medleysolver: online smt algorithm selection. In *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*, pages 453–470. Springer, 2021.

[26] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, 1989.

[27] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[28] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[29] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[30] Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. Machsmt: A machine learning-based algorithm selector for SMT solvers. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 303–325. Springer, 2021.

[31] Joseph Scott, Guanting Pan, Elias B Khalil, and Vijay Ganesh. Meta-solving for deep neural network verification. Personal communication, 2022.

[32] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance sat solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*, pages 336–353. Springer, 2019.

[33] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[34] 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.

[35] 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.

[36] Mate Soos, Raghav Kulkarni, and Kuldeep S Meel. Crystalball: gazing in the black box of sat solving. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings*, pages 371–387. Springer, 2019.

[37] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[38] Guy Tennenholtz and Shie Mannor. The natural language of actions. In *International Conference on Machine Learning*, pages 6196–6205. PMLR, 2019.

[39] Shantanu Thakoor, Surag Nair, and Megha Jhunjhunwala. Learning to play othello without human knowledge, 2016.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[41] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The smt competition 2015–2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):221–259, 2019.

[42] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008.

[43] Edward Zulkoski. Understanding and enhancing cdcl-based sat solvers. *PhD Thesis*, 2018.