Solving word equations with Monte Carlo Tree Search and black-box solvers

Albert Garreta-Fontelles*

Abstract

We propose new approaches to solving word equations which leverage recent general techniques from the field of artificial intelligence and, optionally, already existing heuristic-based word-equation (or string) solvers. We describe and test two algorithms: one which uses any string solver as a black-box, and another which tackles word equations with little domain-specific heuristics, replacing the black-box solver of the previous framework by an artificial neural network. Our methods are obtained by modeling a certain non-deterministic algorithm L for word equations as a Markov Decision Process, and then applying techniques from planning or reinforcement learning to transform L into a probabilistic algorithm. Our experiments show promising results, and moreover the presented frameworks have the potential of being applied to other problems, such as SAT and other Constraint Satisfaction Problems.

1 Introduction

1.1 Word equations: motivation, history, and challenges

Word equations constitute a long-studied problem which lies at the intersection of algebra and computer science, being of interest both from a theoretical and an applied viewpoint. For instance, they are a key component in different computer programs designed for tasks such as software analysis, model checking, database applications, or cybersecurity [2, 5, 7, 17, 18, 23, 36, 37, 40, 42]. As an application example, suppose one is interested in finding out whether it is possible for a program P to reach a certain state S for some input. If the program involves the manipulation of words —also called *strings* in the literature—, it is often the case that S is reachable if and only if a certain word equation (often with some extra constraints) has a solution. For example, suppose the program P is the following:

Data: Two letters s_1, s_2 (strings of length 1).
while *some condition* **do**
  $\ell \leftarrow$ random constant letter from $\{a, b\}$;
  $s_1 \leftarrow s_1\ell, \quad s_2 \leftarrow \ell s_2$; `// concatenation`
if s_1 *and* s_2 *are the same word* **then**
  Execute risky third-party software ; `// State` S
else
  continue program

The question of whether it is possible to reach state S is equivalent to the following word equation with linear constraints on the length of the variables:

$$\exists s_1, s_2, w, \quad s_1 w = w s_2 \ \wedge \ |s_1| = |s_2| = 1 \tag{1}$$

Examining (1) one sees that S is reachable if and only if the input letters s_1 and s_2 are the same.

The growing number of applications of word equations has resulted in an increased effort during the past recent years to build effective and fast word equation solvers [1, 3, 11, 12, 16, 26, 38, 48, 63, 66], which we will refer to as *string solvers*. However, solving word equations is a challenging task: From a practical viewpoint, all existing string solvers are incomplete, and moreover, some commonly used string solvers have trouble solving relatively simple small

*University of the Basque Country UPV/EHU, *albert.garreta@ehu.eus*
Keywords: word equation, Monte Carlo Tree Search, solvers, planning for deterministic actions, reinforcement learning

equations (see Table 2). From a theoretical viewpoint, the question of whether word equations are decidable used to be a major open problem in the 1960 and 1970 decades. Makanin provided an algorithm for decidability in 1977 [51]. However, its termination proof is long and involved, and the algorithm was estimated to have a running time complexity of a tower of several exponents. Two decades later a new approach to the problem via compression was found by Plandowsky [53], who provided a decidability algorithm that runs in non-deterministic polynomial time. Recently Jeż [43] improved this result to non-deterministic linear space, also using compression techniques. Word equations have long been known to be NP-hard (and even very restricted versions of this problem still belong to this complexity class [10, 28]), and they are conjectured to be NP-complete [52]. An important variation is obtained when one considers, in addition to word equations, linear constraints on the length of variables ((1) is an example of this). Whether this problem is decidable remains an important open question [39, 49], despite arising frequently in practice.

In this paper we develop a family of string solvers by using certain general planning or reinforcement learning techniques and a non-deterministic algorithm L for word equations. Optionally, our framework can naturally be used on top of other string solvers, which are then treated as 'black-boxes' (see below for more details). To the best of our knowledge this is the first time that techniques from machine learning are applied to the word equation problem. Regarding approaches from other areas of artificial intelligence, see [19] for a treatment via genetic programming. See Appendix 1.2 for a general overview of related work.

The algorithm L for word equations is such that for each input satisfiable equation there exists a sequence of choices that leads the algorithm to finding a solution (*completeness*), and conversely, if the input is unsatisfiable then there is no such sequence (*soundness*). The planning algorithm that we use is a variation of the so-called Monte Carlo Tree Search (MCTS) method, which is responsible for some of the recent successes in the field of artificial intelligence and others [25, 41, 59]. Most notably, AlphaZero [57, 59] is a go, chess, and shogi playing program that achieved state-of-the-art superhuman performance. To use the MCTS algorithm on word equations, we formulate the problem of solving word equations as if it was a single-player game, which we model with a deterministic Markov Decision Process (MDP). To do this, we use the aforementioned non-deterministic algorithm L, looking at each non-deterministic choice the algorithm has to make as a point where an agent has to decide what action to take next, thereby obtaining an MDP which models the "game" of making the appropriate choices when running L. More generally, with this formulation, one can use MCTS (or any algorithm from reinforcement learning, planning, etc.), in order to transform any non-deterministic algorithm into a probabilistic algorithm, as indicated by the following diagram:

$$\begin{array}{c}\text{Non-deterministic}\\ \text{algorithm}\end{array} \longrightarrow \begin{array}{c}\text{Graph of}\\ \text{choices}\end{array} \longrightarrow \begin{array}{c}\text{Deterministic}\\ \text{MDP}\end{array} \xrightarrow{\begin{array}{c}\text{RL}\\ \text{alg.}\end{array}} \text{Policy} \longrightarrow \begin{array}{c}\text{Probabilistic}\\ \text{algorithm}\end{array}$$

It is important to note that the probabilistic algorithm that results from this process can only be used for assessing the positivity of inputs, but not the negativity. This is due to the completeness and soundness nature of the initial algorithm.

The contributions of this paper are the following:

• As discussed, we develop a word equation solver based on a well-known non-deterministic algorithm for quadratic word equations and a Monte Carlo Tree Search (MCTS) method. The latter is essentially the classical UCT algorithm [45] where the state value function is given either by a black-box string solver or by an artificial neural network. In the second case the network is trained following AlphaZero's [57, 59] method (omitting the prior policy).

• As an application of the above, we develop a *wrapper* for current existing string solvers. By this we mean a program designed for solving word equations that uses any given string solver as backend. This scheme has the potential of being applicable to many other problems, such as the SAT problem.

• Conceptually, one of the main contributions of the paper is to formulate the problem of solvability of word equations as a reinforcement learning (RL) problem by means of a Markov Decision Process (MDP). This is not straightforward as there does not seem to be an obvious natural candidate MDP for it, contrasting with other NP-hard problems recently studied in, e.g. [4, 24].

• We present and make available a database of simple and short quadratic word equations on which some state-of-the-art string solvers achieve relatively poor results. Furthermore, we construct an algorithm which allows to randomly generate these type of equations.

The code and datasets for this project are available at `https://github.com/agarreta/we-uct`. We refer to the final discussion in Section **??** for comments on our experimental results and to Appendix 1.2 for a review of work related to ours.

The paper is organized as follows: in Section 2 we provide the necessary definitions. Section 3 contains the description of the non-deterministic algorithm for word equations used in the paper. In Section 4 we briefly describe the variation of the MCTS algorithm we use. Section 5 contains all our experiments, and finally in Section **??** we discuss the experimental results and possible future work. After the references, in Appendix 1.2 we review related work. In A.1 we provide some proofs and additional details for Section 3. In Appendix A.2 we give a detailed description of the MCTS approach from Section 4, and Appendix A.3 describes the architecture of the artificial neural network used in some of our algorithms.

1.2 Related work

In what follows we review the existing literature regarding the different topics covered in this paper.

Word equations Most literature was reviewed in the introduction. For a detailed account of the history, motivation and key results regarding word equations we refer the reader to the surveys [6, 32]. Recent additional results can be found in, for example, [22, 27, 30, 33, 34, 35, 44, 55**?**].

String solvers See [13, 29] for further information on general SMT solvers. The reader interested in the details of how different string solvers work is referred to the survey [6].

Machine learning, Constraint Satisfaction Problems, and combinatorial optimization In the past few years there has been an increased effort in applying techniques from machine learning to NP-hard discrete problems (such as SAT, the Traveling Salesman Problem, etc.) which have traditionally been approached through heuristic-based domain-specific algorithms. Roughly, there are two important lines of research in this direction: one aims at building artificial intelligence systems capable of learning to solve these problems without the use of expert human knowledge, and the other attempts to combine already existing algorithms and solvers with techniques from machine learning. Currently the amount of methods developed in both directions and the corresponding literature is rather vast, and we refer to the recent survey [15] for further details. Some papers using supervised or unsupervised machine learning approaches which are related to the present work, but treating problems different than word equations, are for example [8, 9, 14, 46, 54, 56, 65].

As mentioned previously, in this paper we formulate a well-known non-deterministic algorithm for solving quadratic word equations by means of a Markov Decision Process, and then we apply a Monte Carlo Tree Search algorithm which leverages the information provided by either a) a string solver (thus producing a *string solver wrapper*), or by b) an artificial neural network, thus, and most importantly, resulting in a system that uses essentially no expert field knowledge. Without trying to be exhaustive, an approach related to case a) can be found in [60], where the author combines a reinforcement learning (RL) model that proposes solutions to a combinatorial optimization problem (COP) and then a solver verifies or repairs the solutions. In [61] an actor-critic RL algorithm is used in a COP with the reward signal being given by a solver. In [46, 65] the authors use RL algorithms to improve the search heuristics of SAT solvers. On the other hand, frameworks related to case b) can be found in [4, 24, 64], where some COP's are formulated as formal games in [64] or as a natural MDP in [4, 24], and then different techniques from deep reinforcement learning are used. Finally, a genetic algorithm for word equations in free groups was proposed in [19].

2 Preliminaries

Word equations Given a set of symbols S, we write S^* to denote the *free monoid* generated by S, i.e. the set of all finite sequences of elements from S, including the empty sequence 1, together with the concatenation operation. We call its elements *words*. The *length* of a word $s \in S^*$ is its sequence-length. We let $\mathcal{X} = \{X, Y, Z, \dots\}$ and $\Sigma = \{a, b, c, \dots\}$ denote finite sets of variables and letters, respectively. A *word equation* is a formal expression of the form $U = V$ where U, V are two words from $(\mathcal{X} \cup \Sigma)^*$. A *solution* to such equation is a map $\alpha : \mathcal{X} \to \Sigma^*$ such that $\alpha(U) = \alpha(V)$ is a true equality in Σ^* (i.e. $\alpha(U)$ and $\alpha(V)$ are exactly the same word from Σ^*), where by $\alpha(U)$ and $\alpha(V)$ we denote the words in Σ^* obtained from U and V after replacing each variable X in U and V by

the word $s(X)$, for all $X \in \mathcal{X}$. For example, if $\Sigma = \{a, b\}$ and $\mathcal{X} = \{X, Y\}$, then the equation $aX = Xa$ admits the (non-unique) solution $X \mapsto a.^{t}.a = a^t$ for any integer $t \geqslant 0$. The equation $XbaY = aYbX$ admits the solution $X \mapsto aba,\ Y \mapsto ba$. The equation $aX = Xb$ admits no solution. A word equation admitting a solution is said to be *satisfiable* or *solvable*.

Markov Decision Processes (MDP) We follow [62]. A MDP is a tuple $(\mathcal{S}, \mathcal{S}_0, \mathcal{S}_f, \mathcal{A}, \mathcal{R}, \rho, r)$ where $\mathcal{S}$ is a set, called the set of *states*; $\mathcal{S}_0$ and $\mathcal{S}_f$ are subsets of $\mathcal{S}$ called *initial* and *final* states respectively; $\mathcal{A}$ and $\mathcal{R} \subseteq \mathbb{R}$ are also sets, called the set of *actions* and of *rewards*, respectively; ρ is a map $\rho : \mathcal{S} \times \mathcal{A} \to Dist(\mathcal{S})$, called the *dynamics map* which sends each state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ to a probability distribution over $\mathcal{S}$; and $r : \mathcal{S} \times \mathcal{A} \to Dist(\mathcal{R})$ is a map which sends each pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ to a probability distribution over $\mathcal{R}$. Intuitively, $\rho(s, a)$ and $r(s, a)$ indicate the different possible states and rewards, respectively, that can result when taking action a on state s. A *policy* is a map $\pi : \mathcal{S} \to Dist(\mathcal{A})$ assigning to each state a probability distribution over the set of valid actions (see below for an explanation of what "valid" means). Given an initial state s_0, one can sample an action a_1 from the distribution $\pi(s_0)$, and then one can sample a reward r_1 from $r(s_0, a_0)$ and a new state s_1 from $\rho(s_0, a_0)$. This can then be successively repeated until a final state s_f is found (or some stop condition is met), producing a sequence $s_0, a_1, r_1, s_1, \ldots, s_{f-1}, a_f, r_f, s_f$. This whole process is called an *episode*, and the sum $r_1 + \gamma r_2 + \cdots + \gamma^{f-1} r_f$ is the *return* of the episode, where $0 \leqslant \gamma \leqslant 1$ is a globally fixed *discount* factor. The goal is to find a policy that maximizes the expected return of an episode. The MDP is said to be *deterministic* if $\rho(s, a)$ and $r(s, a)$ are degenerate distributions, i.e. their support consists of a single point. In this case ρ and r are regarded simply as maps from $\mathcal{S} \times \mathcal{A}$ to $\mathcal{S}$ or $\mathbb{R}$, respectively. In general, it is often convenient to define ρ only on a subset of $\mathcal{S} \times \mathcal{A}$. In this case, if ρ is not defined in a state-action pair (s, a) then we say that the action a is *invalid* at state s. Policies are required to assign zero probability to any invalid action. One important taxonomic aspect when given an MDP is whether the dynamics and reward functions ρ, r are known or not. In this paper the former is always assumed to be the case.

3 Nielsen transformations for word equations

We next consider the decision problem of whether an input arbitrary word equation has a solution or not. We present a non-deterministic algorithm for it. This is a close variation of a well-known deterministic algorithm for assessing both the solvability or unsolvability of *quadratic* word equations[1], and whose complexity is PSPACE [31, 47]). We stress that in our case this is used on any word equation, not necessarily quadratic, and that in this general setting no deterministic algorithm can be derived from it.

We choose to present the algorithm for a single word equation for simplicity, but it is straightforward to extended it to systems of equations.

The algorithm takes a word equation as input and successively applies some rewrite rule to it, which is chosen non-deterministically among a fixed set of rules. These rules are consecutively applied until a point is reached where it is trivial to verify if the current equation is satisfiable or not, at which moment the algorithm halts and returns 1 or -1 (the input equation has or does not have a solution), respectively. More precisely, the algorithm halts as soon as it finds an equation that meets certain stop conditions which ensure the triviality of the equation. If the output is 1 (i.e. the input equation has a solution) it is a trivial task to retrace all applied rules in order to find an actual solution (we omit the details of this). The resulting process is sound and complete (see Appendix A.1.3). Following the discussion in the introduction, we can then construct an associated deterministic MDP $(\mathcal{S}, \mathcal{S}_0, \mathcal{S}_f, \mathcal{A}, \rho, r)$. More precisely, $\mathcal{S}$ is the set of all word equations; $\mathcal{S}_0$ is formed by all satisfiable equations; $\mathcal{S}_f$ consists of all states which satisfy a stop condition (to be defined below); $\mathcal{A}$ consists on the rewrite rules of the algorithm (also to be defined below); ρ is a map capturing the effect of the rewrite rules; and $r(s, a) = 0$ for all s, a unless $\rho(s, a) \in \mathcal{S}_f$, in which case $r(s, a)$ is 1 or -1 depending on whether $\rho(s, a)$ is satisfiable or not (which can be checked trivially). If, additionally, a string solver $\mathcal{O}$ is available, then we extend $\mathcal{S}_f$ with those equations where $\mathcal{O}$ returns an answer, and we modify r accordingly —in practice, this requires to call $\mathcal{O}$ on each new visited state.

In what follows we identify the algorithm with its associated MDP.

[1] The algorithm never increases the length of a quadratic equation. Hence there is a finite number of different states any execution path can meet. This allows to obtain computation bounds and a deterministic algorithm for fully solving this subclass of equations.

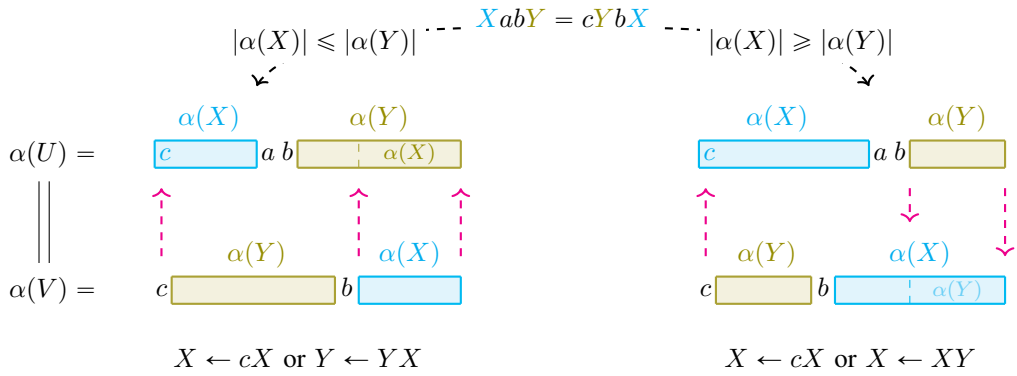

Figure 1: This figure aims at providing some intuition behind the rewrite rule (R2). Further intuition can be gathered from the proof of completeness in Appendix A.1.3. Here we have taken the example word equation $XabY = cYbX$. Let α be a hypothetical solution to it. By definition, $\alpha(U)$ and $\alpha(V)$ are graphically equal. We draw $\alpha(U)$ and $\alpha(V)$ one on top of the other in order to extract information about α by comparing the two words. Since we do not know the actual values $\alpha(X)$ and $\alpha(Y)$, we leave them blank, but we do make a guess regarding their relative length. For simplicity we suppose $\alpha(X) \neq 1 \neq \alpha(Y)$ (otherwise we can apply (R1)). In the left part of the figure we assume that $\alpha(X)$ is shorter than $\alpha(Y)$, while on the right part we assume the contrary. Clearly, looking at the first letters of $\alpha(U)$ and $\alpha(V)$, we see that $\alpha(X)$ must start with c in all cases. This is reflected by the (R2) rewrite rule where we replace X by cX (note that there is a slight abuse of notation: it would be notationally more correct to replace X by cX', where X' is a new variable). Looking at the last letters of $\alpha(U)$ and $\alpha(V)$ we see that if $|\alpha(X)| \leqslant |\alpha(Y)|$, then $\alpha(Y)$ must end with $\alpha(X)$ (left hand-side of the picture), and so we can replace Y by YX. The case when $\alpha(X)$ is larger than $\alpha(Y)$ corresponds to the (R2) rule $X \leftarrow XY$, and is depicted in the right hand-side of the figure. These are all possible valid (R2) rules that can be applied in this example.

Actions We now describe the rewrite rules $\mathcal{A}$ of the algorithm. At the end of the present section we provide an example. Let $U = V$ be a word equation. Denote by U_0 and U_{-1} the first and last letters of U, and similarly let V_0 and V_{-1} be the first and last letters in V. We define the following transformations (if a rule cannot be applied on $U = V$ then the equation remains the same):

- (R1) If possible, choose a letter $Z \in \{V_0, V_{-1}, U_0, U_{-1}\}$ such that Z is a variable. Then replace all occurrences of Z in $U = V$ by the empty word.
- (R2) This rewrite rule is perhaps best understood with the visual example from Figure 1. Formally: if possible, choose a letter $Z \in \{V_0, V_{-1}, U_0, U_{-1}\}$ such that Z is a variable, and choose a "side" $\ell \in \{\textit{left}, \textit{right}\}$. Replace all occurrences of Z in $U = V$ by one of the following words: 1) V_0U_0 if $Z = U_0$ and $\ell = \textit{left}$; 2) $U_{-1}V_{-1}$ if $Z = U_{-1}$ and $\ell = \textit{right}$; 3) U_0V_0 if $Z = V_0$ and $\ell = \textit{left}$; or 4) $V_{-1}U_{-1}$ if $Z = V_{-1}$ and $\ell = \textit{right}$.
- (R3) Delete the largest common prefix in U and V, and then delete the largest common suffix. For example, $cXabY = cYbcX$ becomes $XabY = YbcX$.

A rule a is *valid* at state s if and only if it can be applied and it transforms s into a different equation, i.e. $\rho(s, a) \neq s$.

Stop conditions and final states Next we define some conditions for a word equation $U = V$ such that if one of the conditions is satisfied, then the equation is trivially satisfiable or trivially unsatisfiable.

- (C1) No variables appear in the equation $U = V$. In this case the equation is trivially satisfiable or unsatisfiable.
- (C2) U and V start or end with distinct constant letters from Σ. In this case the problem is unsatisfiable.
- (C3) U and V are the exact same word from $(\Sigma \cup \mathcal{X})^*$. In this case $U = V$ is satisfiable.
- (C4) U is a variable and V contains no variables (that is, $U \in \mathcal{X}$ and $V \in \Sigma^*$), or vice-versa. In this case $U = V$ is satisfiable.

- (C5) To each word equation $U = V$ one can associate a a real-valued linear program $L_{U=V}$ such that the satisfiability of $L^{\mathbb{R}}_{U=V}$ is a necessary condition for the satisfiability of $U = V$. Hence, we define condition (C5) to hold if $L^{\mathbb{R}}_{U=V}$ is unsatisfiable, in which case the equation $U = V$ is unsatisfiable. This linear program is described in Appendix A.1.2

- (Partial oracle) When a string solver $\mathcal{O}$ is available we stop at those states where $\mathcal{O}$ is able to return an answer (1 or -1).

Algorithm properties We have completed the description of our algorithm. This is essentially the same as one can find in the literature for quadratic word equations (again, we stress that we use it on any type of equation, not necessarily quadratic), with minor modifications and some additional stop conditions. It is well-known that each of the rewrite rules (R1), (R2) and (R3) is sound, i.e. it cannot turn an unsatisfiable equation into a satisfiable equation) [31, 47] (see also Appendix A.1.3 for a proof), hence the algorithm is sound. Furthermore, one can use standard arguments to show that the whole algorithm is complete, for any solvable input there exists a sequence of actions bringing the input to a satisfiable final state (again, see Appendix A.1.3).

An appealing aspect of the associated MDP is that the maximum number of valid actions per state is 8: a small number which favors Monte Carlo Tree Search approaches[2].

Example 3.1. We now provide an example of how a satisfiable word equation can be brought to a trivial satisfiable equation using the actions above. We consider the equation $XaXa = bYbY$. The moves described below correspond to the solution $X = b$ and $Y = a$, which is not unique, e.g. $X = ba^t, Y = a^{t+1}$ is also a solution for all $t \geqslant 0$. For each action we indicate what rule is taking place.

$$XaXa = bYbY \overset{X \leftarrow bX \text{ (R2)}}{\longrightarrow} bXabXa = bYbY \overset{\text{del pref/suf (R3)}}{\longrightarrow} XabXa = YbY$$
$$\overset{X \leftarrow 1 \text{ (R1)}}{\longrightarrow} aba = YbY \overset{Y \leftarrow aY \text{ (R2)}}{\longrightarrow} aba = aYbaY \overset{\text{del pre/su (R3)}}{\longrightarrow} ba = YbaY$$
$$\overset{Y \leftarrow 1 \text{ (R1)}}{\longrightarrow} ab = ab.$$

4 Monte Carlo Tree Search

As explained in the introduction, we can potentially apply any technique from reinforcement learning in order to find a policy for the Markov Decision Process from the previous section. Since in our case the model (i.e. the dynamics map ρ) is known to the agent, we will use a Monte Carlo Tree Search (MCTS) algorithm for this. Moreover, our MDP has only 8 actions per state, which is one extra argument in favor of this approach. We refer to [20] for an in-depth discussion of different types of MCTS algorithms.

Due to space limitations, here we assume familiarity with MCTS. The full details of the algorithm can be found in Appendix A.2.

We use a variation of the classical UCT algorithm [45] with discounts (i.e. $\gamma < 1$) and transposition tables (see Remark A.1). These are two well-known techniques in general MDP's and in MCTS algorithms, respectively, which favor short episodes. In particular, it promotes not repeating the same state during an episode, something that occurs often in our applications.

The only variation in our method is the following: Our UCT algorithm requires having a function f that estimates the *value* of an equation $U = V$, i.e. a numeric estimate indicating how 'easy' it is to find a solution to $U = V$. This function is used in replacement of rollout plays for value estimation. We explore two main variations of our framework, each one featuring a different way of obtaining these estimates:

• **MCTS$_{\text{solver}}$**: In our first variation we obtain f from already existing string solvers, which we call and treat as a black-box. More precisely, given an equation $U = V$, we let its value f be $1, -1$ or 0 depending on whether a fixed string solver outputs that $U = V$ has a solution, that it has no solution, or that it is unable to provide an answer, respectively, when given $U = V$. The resulting MCTS algorithm, denoted MCTS$_{\text{solver}}$, can be seen as a *wrapper* for already existing string solvers, and it has the potential, as shown in our experiments, to improve the performance and the speed of the string solver itself. See Table 1.

[2]This is why in (R1) we only allow deletion of variables that appear on the leftmost or rightmost side of U or V, rather than allowing the deletion of any variable.

• **MCTS$_{nn}$**: On the second variation we follow recent trends in reinforcement learning and we use an artificial neural network as value estimator f. This network is trained via self-play following the same scheme used by AlphaZero [59], using the mean squared error as loss function. We denote the resulting algorithm MCTS$_{nn}$. An appealing aspect of it is that it uses little domain-specific information, yet it is capable to perform better than many popular string solvers in some datasets with small word equations (see Table 2). However, due to the long preliminary training phase that it requires, we have tested MCTS$_{nn}$ only on small word equations at the moment.

5 Experiments and results

We tested the previous Monte Carlo Tree Search (MCTS) algorithm from Section 4 applied to the Markov Decision Process (MDP) for word equations described in Section 3. Recall from Section 4 that our algorithm uses either a string solver or an artificial neural network in order to estimate the value of states. We write MCTS$_{solver}$ to denote the former and MCTS$_{nn}$ to denote the later.

In all experiments and for all algorithms we set a time bound of 30 seconds per equation. The MCTS algorithm is given 10 simulations per action in MCTS$_{solver}$, and either. $30, 50$, or 100 in MCTS$_{nn}$. The discount factor γ is set to 0.9 (chosen via empirical experimentation). Except when training MCTS$_{nn}$, the agent always selects actions with the highest probability assigned by the MCTS policy, instead of sampling from the policy. Moreover, if not in training mode and if during a MCTS simulations a satisfiable equation is found, then the episode is automatically ended and a return of 1 is given (this is done because at that point we can be certain that the initial equation was satisfiable, hence it is pointless to continue the episode). All experiments are conducted on datasets all whose equations are solvable. We repeated all experiments 3 times with different random seeds. The reported outcomes are the mean of these. We ran the experiments on Ubuntu Linux 20.04 with an Intel Core i9-9900K CPU at 3.60GHz, 32GB of RAM, and an Nvidia RTX2070 GPU, except for the training process of MCTS$_{nn}$, which we ran on Windows 10.0 with the same hardware.

The results are presented and discussed in Tables 1, 2, and 3.

Experiments on existing datasets In our first experiment we compare the performance of MCTS$_{solver}$ with the performance of the solver itself on three different datasets. Being far from exhaustive, we test the popular solvers z3str3 [16], z3seq, CVC4 [12, 48], TRAU [1], and Woorpje [26].

We use the first three datasets from [26], which are available at `https://www.informatik.uni-kiel.de/~mku/woorpje`, and the one descrined below. The first set consists of 200 satisfiable word equations obtained by generating random words and suitably replacing random subwords by variables. Each equation has at most 15 distinct variables, 10 distinct letters, and length of at most 300 (by length we mean the length of one side of the equation plus the length of the other side). The second dataset consists of 9 equations of the form $X_n a X_{n-1} b X_{n-2} \dots b X_1 = a X_n X_{n-1}^2 b X_{n-2}^2 b \dots b X_1^2 b a^2$, which by Proposition 2 of [28] have minimal solutions of exponential length. The third dataset is obtained from the second one by suitably replacing the b's of each equation by the left or right-hand side of some satisfiable word equation generated as in Track 1. There are two more datasets in [26] which we do not explore: one consists of word equations with length constraints, and the other consists of systems of equations. We leave these extended problems for future work.

We remark that all of the datasets contain non-quadratic equations. This is notable since our MDP is derived from an algorithm which in the literature is used exclusively on quadratic word equations.

Each call to the solver in MCTS$_{solver}$ is given a maximum time of 0.8 seconds, except 1 for Woorpje (which accepted only integer entries in seconds). For TRAU we also test a version where TRAU is given 4 seconds (see the description of Table 1 for more details). We stress that when testing the solvers alone (outside of MCTS$_{solver}$, i.e. the columns 'solv' in Table 1), we allow a maximum time of 30 seconds per equation, as is the case for all tested algorithms in this paper.

Experiments on a new dataset with quadratic word equations For this experiment we generate satisfiable equations by starting on randomly generated trivially satisfiable final states $w = w$ ($w \in \Sigma^*$) and then applying the inverse of actions (R1) and (R2) from the MDP a random number of times between 5 and 25.[3] Each time, the action (R1) is selected with a probability of 0.4 and (R2) with a probability of 0.6 (the last rule (R3) is never selected). An action that cannot be applied or that does not change the current equation is not counted. The inverse of (R1) consists in selecting a variable not appearing in the equation and randomly inserting several copies of it in the equation. More specifically, we

[3]The different number of inverse steps required to generate each equation is evenly spaced throughout a dataset: i.e. there is approximately the same number of 'easy' 'intermediate', and 'difficult' equations.

		z3str3		z3seq		CVC4		Woorpje		TRAU		
	MCTS	MCTS_s	solv	MCTS_s	solv	MCTS_s	solv	MCTS_s	solv	MCTS_s	MCTS_s^4	solv
Quadratic												
Score	71	137	31	187	170	108	61	200	200	165	184	162
C.Time		1.25	1.31	1.03	2.11	0.10	0.09	0.16	0.16	2.62	2.86	2.59\2.86
T.Time	0.16	5.70	1.31	1.47	2.17	3.99	0.10	0.16	0.16	3.14	4.29	3.30
Track 1												
Score	172	199	192	196	187	199	191	200	200	199	198	196
C.Time		0.35	0.37	1.03	0.67	0.17	0.15	0.18	0.15	0.37	0.42	0.47\0.46
T.Time	2.22	0.47	0.38	1.42	0.79	0.43	0.15	0.18	0.15	0.50	0.51	0.46
Track 2												
Score	0	2	0	2	0	2	1	4	5	3	4	3
C.Time		-	-	-	-	0.27	0.14	0.33	0.27	3.41	1.20	1.22\1.20
T.Time	-	6.90	-	5.43	-	10.85	0.14	0.33	0.97	3.41	4.62	1.20
Track 3												
Score	56	124	80	144	137	134	121	146	147	123	134	134
C.Time		0.87	1.27	0.70	0.83	0.24	0.17	0.31	0.18	2.74	2.63	1.93\2.52
T.Time	1.81	3.52	1.80	1.06	1.01	0.30	0.20	0.18	0.15	2.92	2.83	2.99

Table 1: Performance of different algorithms on the datasets described in Section 5. Here we test $\text{MCTS}_{\text{solver}}$ for the string solvers z3str3, z3seq, CVC4, Woorpje, and TRAU. The results are organized column-wise accordingly. The subcolumns MCTS_s show the results obtained by the $\text{MCTS}_{\text{solver}}$ algorithms (with 'solver' running over the previous list of solvers), while the subcolumns *solv* indicate the results obtained by the solvers themselves. For TRAU we tested one more instance of $\text{MCTS}_{\text{solver}}$, indicated under MCTS_s^4, where TRAU was allowed a maximum of 4 seconds per call, since the standard 0.8 seconds resulted in a decrease of performance for tracks 3 and 4 with respect to using TRAU itself. In the column "MCTS" we include baseline results for the algorithm $\text{MCTS}_{\text{solver}}$ where the value function is constant 0 unless the equation is a final state, in which case it is 1 or -1 accordingly. The rows *Score* indicate the total number of equations on which the algorithm obtained the correct answer, for each corresponding dataset. The rows *T.Time* (total time) indicate the average number of seconds spent on equations were the correct answer was obtained, while the rows *C.Time* (common time) indicate the average time spent on those equations where both the $\text{MCTS}_{\text{solver}}$ and the solver itself (for each separate solver) obtained the correct answer. For TRAU, the entries in 'C.time' and 'solv' include the common times of UCT_{TRAU} and TRAU, and $\text{UCT}_{\text{TRAU}}^4$ and TRAU, respectively. We have highlighted in blue or pink the most notable improvement or decrease of performance of $\text{MCTS}_{\text{solver}}$ over the solver itself (for the C.Time rows we highlight if and only if the improvement/decrease is of more than 25%).

As we can see, in most cases $\text{MCTS}_{\text{solver}}$ improves the score of the solver, with this improvement being highly significant for some solvers and some tracks, specially for z3str3, z3seq, and CVC4. On the other hand the solver Woorpje either obtains the same score or improves $\text{UCT}_{\text{woorpje}}$, and furthermore Woorpje clearly outperforms all other solves. We believe that the high performance of Woorpje makes our comparison of it with $\text{UCT}_{\text{woorpje}}$ uninformative, and that more complicated datasets should be used for this specific case. For speed comparisons we look at the column "Common time" (C.Time) column, since we believe that the equations solved by one algorithm but not solved by the other are likely to be more difficult, which may lead to an increase of the time average. We see a disparity of results, with $\text{MCTS}_{\text{solver}}$ often being slightly faster or similar as $solver$, with occasional significant improvements or performance decrease. Overall, these results seem to indicate that a long solver evaluation can or tends to be less effective and efficient than several short solver evaluations while successively transforming the equation into a more favorable one.

insert one copy of the variable on a random position of the left-hand side of the equation, and another copy in a random position of the right-hand side. Hence the resulting final equation is quadratic, and more precisely, it is *oriented*, i.e. each variable appears once in the right-hand side of equality and once in the left-hand side. *Determining the solvability of oriented quadratic equations is an* NP*-complete problem* [**?**] (see also [50]). The dataset contains 200 satisfiable oriented quadratic word equations of length at most 40, and with at most 5 and 3 different variables and constants, respectively. This set, which we will refer to as *quadratic* is available at `https://github.com/agarreta/we-uct`.

To ease the training process of the network, during this experiment all equations are automatically brought to normal form (see Section A.1.1) with respect to fixed orderings of Σ and $\mathcal{X}$. The results are presented in Tables 2 and 3.

Quadratic	$\text{MCTS}_{\text{nn}}^{30}$	MCTS^{30}	$\text{MCTS}_{\text{nn}}^{50}$	MCTS^{50}	$\text{MCTS}_{\text{nn}}^{100}$	MCTS^{100}	z3str3	z3seq	CVC4	Woorpje	TRAU
Score	147	90	154	100	159	106	31	171	61	200	162
T.Time	1.64	3.35	2.82	2.81	4.51	3.85	1.36	2.17	0.10	0.17	3.26

Table 2: Comparison of MCTS_{nn} with the rest of solvers on the quadratic dataset. The superindex in the MCTS_{nn} algorithms indicates the number of MCTS simulations used per move (the network is fixed throughout all these, and it was trained using 50 simulations per action, as descibred below). The columns MCTS^s ($s = 30, 50, 100$) contain the results for the baseline algorithms obtained from $\text{MCTS}_{\text{nn}}^s$ replacing the network by a function that outputs 0 if the equation is not a final state, and 1 or -1 otherwise, depending on whether the equation is satisfiable or not. The rows 'Score' and 'T.Time' have the same meaning as in Table 1. We see that MCTS_{nn} attains relatively good scores, significantly outperforming z3str3 and CVC4, and attaining similar, but worse, results as z3seq and TRAU, despite using essentially no domain-specific information. A pairwise speed comparison of average time spent on commonly solved equations is given in Table 3. MCTS_{nn} was trained only on small equations and, due to the architecture choice of its neural network, it cannot be evaluated on the larger equations that appear on the other datasets (Tracks 1, 2 and 3) (see Section **??**).

C.Time	$\text{MCTS}_{\text{nn}}^{30}$	$\text{MCTS}_{\text{nn}}^{50}$	$\text{MCTS}_{\text{nn}}^{100}$	z3str3	z3seq	CVC4	Woorpje	TRAU
$\text{MCTS}_{\text{nn}}^{30}$	-	1.56\2.48	1.58\3.90	0.90\0.88	1.28\2.40	1.23\0.10	1.69\0.18	1.65\3.38
$\text{MCTS}_{\text{nn}}^{50}$	2.48\1.56	-	2.69\4.21	1.90\0.86	2.24\2.29	1.90\0.10	2.82\0.18	2.61\3.36
$\text{MCTS}_{\text{nn}}^{100}$	3.90\1.57	4.21\2.69	-	2.17\0.85	3.70\2.16	3.00\0.10	4.51\0.17	4.22\3.35
z3str3	0.88\0.89	0.86\1.40	0.85\2.51	-	1.39\1.57	0.60\0.10	1.36\0.14	1.43\3.76
z3seq	2.40\1.30	2.29\2.09	2.16\3.48	1.57\1.39	-	2.19\0.10	2.21\0.17	2.48\3.41
CVC4	0.10\1.24	0.10\1.73	0.10\2.76	0.10\0.60	0.10\2.19	-	0.10\0.22	0.10\3.99
Woorpje	0.18\1.69	0.18\2.70	0.17\4.24	0.11\1.36	0.17\2.21	0.22\0.10	-	0.18\3.27
TRAU	3.38\1.68	3.36\2.51	3.35\3.96	3.76\1.43	3.42\2.48	3.99\0.10	3.27\0.18	-

Table 3: Pairwise comparison of the average time spent by different pairs of algorithms on commonly solved equations from the quadratic dataset (see Table 2). For each entry, the number on the left (respectively, right) of the backslash is the average time spent by the corresponding algorithm in the row (resp., column).

Testing the generalization capacities of MCTS_{nn} We evaluated MCTS_{nn} on a dataset consisting of 300 small word equations taken from the systems of equations of Track 4 in [26], together with 50 small equations from Tracks 1, 2, and 3 (this is available at https://github.com/agarreta/we-uct). We tested MCTS_{nn} in this dataset twice: once before starting the training process, and another time when the training was over. On the first occasion MCTS_{nn} solved 273 equations, while on the second the score was of 266. Hence at the moment MCTS_{nn} does not seem to exhibit generalization capabilities, which is a point to look into in future work.

Training details The $\text{MCTS}_{\text{solver}}$ algorithms can be applied directly without any initial training. On the other hand, MCTS_{nn} requires a training phase where the parameters of the neural network are fitted as explained in Section A.2.

Here we explain the specific details of this training procedure. The total number of trainable parameters of the network is approximately 150k.

We run 7 "train workers" in parallel until a total of 200k word equations have been attempted. These equations are generated with the same method as the one used for creating the "quadratic" dataset. The parameters of the artificial neural network are optimized every time 1000 equations have been attempted. To measure that the network is being trained towards the right direction we periodically run an extra "test worker" on a test dataset, which is a subset of 100 equations taken from the quadratic dataset. The 7 "train workers" take actions by sampling from their policy, while the test worker always takes the action with the highest probability, as done in all previous experiments. In Figure 2 we show the score evolution of both the test and the train workers, as well as the prediction losses of the network.

The learning rate is 0.001 and we use a batch size of 64. We used a constant 50 MCTS simulations per action. None of the equations seen during training appear in the test or the quadratic dataset.[4]

Figure 2: Test, train, and loss evolution during training. The uppermost (brown) graph shows the percentage of equations solved by the test worker on the test dataset (consisting of 100 equations from the quadratic dataset). The middle (blue) graph indicates the percentage of equations solved by the train workers, and the lowermost (green) graph shows 100 times the loss incurred by the network. We display the running mean across 3000 equations, averaged over 3 training sessions with different random seeds. The shaded area represents the standard deviation.

Representation of equations and neural network architecture As neural network we defaulted to the same ResNet-based architecture as in [57, 59], and we removed the part concerning the prior policy. The details of such architecture as well as our choice of parameters are given in the Appendix A.3. This neural network, denote it f, takes as input three-dimensional matrices (called *tensors*). Hence before we can input a word equation $U = V$ into f we need to encode the equation in such a form. We do this in the following way: To each symbol $z \in \mathcal{X} \cup \Sigma$ we associate a $2 \times L$ matrix M_z where the first row consists all of zeros except for ones in each position i such that the i-th letter of U is z (see the subsequent example). The second row of the matrix M_z is obtained in the same way by replacing U with V. The entire three dimensional tensor is obtained by stacking the matrices $\{M_z \mid z \in \mathcal{X} \cup \Sigma \cup \{1\}\}$ following

[4]The possible number of satisfiable word equations that can result from our generating procedure is large enough that we naturally did not encounter any repeated equation.

a fixed ordering of $\mathcal{X} \cup \Sigma \cup \{1\}$, so the shape of the resulting tensor is $(|\mathcal{X}| + |\Sigma| + 1) \times 2 \times L$. For example, if $\mathcal{X} = \{X, Y, Z\}$, $\Sigma = \{a, b, c\}$, $L = 5$, then the word equation $bYac = bXcX$ is represented as the following $7 \times 2 \times 5$ tensor:

$$\left[\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}_X, \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}_Y, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}_Z, \right.$$
$$\left. \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}_a, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}_b, \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}_c, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}_1 \right]$$

where we have ordered $\mathcal{X} \cup \Sigma \cup \{1\}$ as $X, Y, Z, a, b, c, 1$, and the subscripts of the 2×5 submatrices are just a notational aid for identifying the matrices $\{M_z \mid z \in \mathcal{X} \cup \Sigma \cup \{1\}\}$.

A key limitation of this approach is that the network does not accept equations of arbitrary length, or with arbitrarily many different letters and variables. Rather, these need to be fixed beforehand. We used this ResNet-based architecture as a default baseline network. We believe a more suitable design can be used for word equations. This, and some alternatives we tried, is discussed further in the next final Section **??**.

Acknowledgements

We are grateful to Alexei Miasnikov for useful and interesting conversations, and to him and Marialaura Noce for reviewing the paper.

The author was supported by the ERC grant PCG-336983, the Basque Government grant IT974-16 Ministry of Economy, and the Industry and Competitiveness of the Spanish Government Grant MTM2017-86802-P.

A Appendix

A.1 Additional details of the word equation algorithm

This section furnishes the algorithm presented in Section 3.

A.1.1 Fixed-order normal forms

The satisfiability of a word equation is invariant under permuting the constant symbols and permuting the variable symbols. Hence it can be convenient to use a "canonical representation" of a word equation. We will only use this representation before inputting an equation into an artificial neural network.

More precisely, let $\sigma : \Sigma \to \Sigma$ and $\delta : \mathcal{X} \to \mathcal{X}$ be bijections, and given $w \in (\Sigma \cup \mathcal{X})^*$ denote by $(\sigma, \delta)(w)$ the word obtained from w by replacing each constant $a \in \Sigma$ by $\sigma(a)$, and each variable $X \in \mathcal{X}$ by $\delta(X)$. Clearly, $\alpha : \mathcal{X} \to \Sigma^*$ is a solution to a word equation $U = V$ if and only if $(\sigma, \delta) \circ \alpha$ is a solution to the word equation $(\alpha, \delta)(U) = (\alpha, \delta)(V)$, where $(\sigma, \delta) \circ \alpha$ is defined by $(\sigma, \delta) \circ \alpha(w) = (\sigma, \delta)(\alpha(w))$ for any word w. For this reason, we fix an ordering of $\mathcal{X}$ and an ordering of Σ, and we say that the *normal form* of $U = V$ is the word equation $(\alpha, \delta)(U) = (\alpha, \delta)(V)$ where α, δ are such that the order in which letters from Σ appear in the word $((\alpha, \delta)(U))((\alpha, \delta)(V))$ respects the order of Σ, and similarly for variables and the order of $\mathcal{X}$. For example, if we have $\mathcal{X} = (X, Y)$ and $\Sigma = (a, b)$ (with X and a being first to Y and b, respectively), then the normal form of $YbbYaa = XbbX$ is $XaaXbb = YaaY$.

A.1.2 Associated linear program

Here we give the details of stop condition (C5) from Section 3, which depends on whether a certain linear program is satisfiable or not.

To each word equation $U = V$ one can associate an integer linear program $L^{\mathbb{Z}}_{U=V}$ such that the satisfiability of $L^{\mathbb{Z}}_{U=V}$ is a necessary condition for the satisfiability of $U = V$ (see below). In particular, the satisfiability of the real-valued relaxation of the problem (i.e. allowing variables to take real values) is also a necessary condition. Hence, we define condition (C5) to hold if the associated *real-valued* linear program $L^{\mathbb{R}}_{U=V}$ is unsatisfiable, in which case the equation $U = V$ is unsatisfiable. We note that, unlike the NP-complete integer version, satisfiability of real linear

programs is a problem which can be solved in polynomial time, with existing solvers being very fast in practice. This is the reason why we use the real-valued version instead of the harder integer one.

The system $L^{\mathbb{R}}_{U=V}$ (or $L^{\mathbb{Z}}_{U=V}$) is obtained as follows: Intuitively, suppose $\alpha : \mathcal{X} \to \Sigma^*$ is a hypothetical solution to $U = V$. Then, after replacing each occurrence of each variable $X \in \mathcal{X}$ in $U = V$ by $\alpha(X)$, we have that the number of occurrences of any constant letter $a \in \Sigma$ on the left side of the resulting equality is the same as on the right side (indeed, both sides are the same word from Σ^*). Thus the following integer identities are satisfied: $|U|_a + \sum_{X\in\mathcal{X}} |U|_X |\alpha(X)|_a = |V|_a + \sum_{X\in\mathcal{X}} |V|_X |\alpha(X)|_a$for all $a \in \Sigma$, where by $|W|_z$ we denote the number of times a letter z appears in a word W. Formally, introduce real-valued (or integer-values) variables $n_{X,a}$ for each $X \in \mathcal{X}$ and $a \in \Sigma$. Then the system $L^{\mathbb{R}}_{U=V}$ (or $L^{\mathbb{Z}}_{U=V}$) consists on the linear equations $\sum_{X\in\mathcal{X}} |U|_X n_{X,a} + |U|_a = \sum_{X\in\mathcal{X}} |V|_X n_{X,a} + |V|_a$ for each $a \in \Sigma$.

A.1.3 Proof of completeness and soundness

We proceed to prove that the algorithm described in Section 3 is sound and complete. The proof follows standard arguments and it is essentially the same proof that one can find in the literature.

Clearly, the satisfiability assessment provided in the description of the stop conditions (C1) through (C5) is correct. Hence we focus on word equations $U = V$ that do not satisfy any of these conditions.

Proof of soundness It suffices to show that none of the rules (R1), (R2), (R3) can transform an unsatisfiable word equation $U = V$ into a satisfiable one $U' = V'$. To prove this we assume that $U' = V'$ has a solution $\alpha' : \mathcal{X} \to \Sigma^*$ and from α' we construct a solution α to $U = V$. If (R1) was applied for deleting a variable say X, then we take α to be exactly as α' except we set $\alpha(X) = 1$. If (R2) was applied to, say, rewrite a variable X into zX, where $z \in \mathcal{X} \cup \Sigma$, then we take α to be precisely α' except that we set $\alpha(X) = \alpha'(z)\alpha'(X)$ if $z \in \mathcal{X}$, or $\alpha(X) = z\alpha'(X)$ if $z \in \Sigma$. Finally, if (R3) was applied, then we can take α to be exactly α'.

Proof of completeness Let $U = V$ be a satisfiable word equation. We prove that there exists a sequence of rewriting rules (R1), (R2), (R3) that transform $U = V$ into an equation satisfying some of the stop conditions and being trivially satisfiable. We let $\alpha : \mathcal{X} \to \Sigma^*$ be a solution to it.

We define the *complexity* of the tuple $(U = V, \alpha)$ to be the tuple $(n_v(UV), |\alpha(U)| + |\beta(V)|)$, where $n_v(W)$ denotes the number of different variables appearing in a word $W \in \mathcal{X} \cup \Sigma$, and $\alpha(W)$ we mean the word obtained after replacing each $X \in \mathcal{X}$ with $\alpha(X)$. We order complexities using the left lexicographic order.

The proof proceeds by induction on the complexity of $(U = V, \alpha)$, with the base cases corresponding to equations satisfying some stop condition being obvious. Hence assume $U = V$ does not satisfy any stop condition (C1) through (C5), so in particular $n_v(UV) > 0$. Suppose first that $\alpha(X) = 1$ for some $X \in \mathcal{X}$ appearing in U or in V. Then we apply (R1) in order to delete X from $U = V$. Then α is still a solution of the resulting equation $U' = V'$, and the tuple $(U' = V', \alpha)$ has less complexity than $(U = V, \alpha)$. Hence by induction there exists a sequence of rewriting rules (R1), (R2), (R3) that bring $U' = V'$ to a satisfiable equation meeting some stop condition (C1) through (C4). Thus, such a sequence also exists for $U = V$. Now assume that $\alpha(X) \neq 1$ for all $X \in \mathcal{X}$ appearing in U or in V. Suppose (R3) can be applied, i.e. U and V have a non-trivial common prefix or suffix. Then α is still a solution to the resulting problem $U' = V'$. Moreover $(U' = V', \alpha)$ has less complexity than $(U = V, \alpha)$, and so again we can proceed by induction. Assume now that (R3) cannot be applied. Then the first letter of either U or V is a variable, otherwise we can apply (R3) or else U and V both start with different letters, in which case $U = V$ has no solution. Now let X be a variable such that either U or V start with X, say U, and if V starts with a variable Y then $|\alpha(Y')| < |\alpha(X)|$. Denote by $z \in \mathcal{X} \cup \Sigma$ the first letter of V. Now apply (R2) so that X is replaced by zX. The resulting equation $U' = V'$ admits the solution α' defined as $\alpha'(Y)\alpha(Y)$ for all $Y \in \mathcal{X}$ such that $Y \neq X$, and $\alpha'(X) = z[\alpha(X)$ if $z \in \Sigma$, where by $z[\alpha(X)$ we denote $\alpha(X)$ after deleting the prefix z; and if $z \in \mathcal{X}$ then $\alpha'(X) = \alpha(z)[\alpha(X)$ (i.e. delete the prefix $\alpha(z)$). The resulting tuple $(U' = V', \alpha')$ has the same complexity as $(U = V, \alpha)$ and moreover U' and V' start both with the same letter z, hence now we can apply (R3) in order to obtain a satisfiable tuple $(U'' = V'', \alpha')$ with less complexity than $(U = V, \alpha)$. Now we can proceed by induction as before. This completes the proof.

A.2 Details of the Monte Carlo Tree Search algorithm

Here we give a detailed descrition of the MCTS algorithm used in our methods.

Broadly, any MCTS algorithm takes a state $s_0 \in \mathcal{S}$ as input and outputs a probability distribution over the set of actions $\mathcal{A}$, which we denote *mcts*(s_0) (or mcts$_\theta(s_0)$ when some parameters θ are involved), hence producing a policy. In this paper we always take the action with the maximum probability with respect to *mcts*(s_0), except during the training process from Section 5 where actions are sampled from *mcts*(s_0).

To obtain the distribution *mcts*(s_0) one iteratively updates a tree T which stores information about different execution paths starting at s_0. Once this is done, the information stored in the tree T is used to obtain *mcts*(s_0). We call each iteration a *simulation*[5]. Every vertex v of T is labeled with a state from $\mathcal{S}$, and each edge starting at a vertex v is labeled with a state-action pair (s, a) where s is the label of v and a is some action. Each vertex v is either a leaf (i.e. it has no outgoing edges) or it has one outgoing edge for each action in $\mathcal{A}$.

We assume we have been given a function $f : \mathcal{S} \to [-1, 1] \subseteq \mathbb{R}$ which maps states to real numbers between -1 and 1, which we take as an estimation of the *value* of states (i.e. the expected return). This map will be provided either by a neural network or by a partial oracle (for us this will be a string solver).

The number of simulations performed is determined by a global fixed parameter. Each simulation consists of the four consecutive steps described below. The tree and its associated information is maintained and updated throughout an episode (instead of building a tree from scratch for each move in an episode). It is initialized at the beginning of an episode with a single vertex v_0 labelled with the initial state. The tree is deleted once the episode ends.

We now describe the four steps that comprise a simulation (see also Figure 3).

Selection. Starting at the initial vertex v_0 the algorithm travels along the tree T by successively selecting edges (i.e. actions) until reaching a leaf. This produces a sequence $s_0, a_1, v_1, \ldots, a_N, s_N$ given by the labels of the vertices and the edges. For each $1 \leqslant n \leqslant N$ the edge with label a_n is selected so that

$$a_n =_{\text{def}} \operatorname{argmax}_{a \in \mathcal{A}_s} \left(Q(s_n, a) + c\sqrt{\frac{\log(N(s_n))}{1 + N(s_n, a)}} \right), \tag{2}$$

(ties are decided randomly) where $\mathcal{A}_s$ is the set of valid actions at s; c is an exploration parameter that we set to 1.25; $Q(s, a)$ is an estimation of the return attained when taking action a when at state s (to be defined below); and $N(s)$ and $N(s, a)$ are the total number of times a vertex labeled s, and an edge labeled (s, a), respectively, have been traversed during all selection phases of the episode.

This formula is intended to balance exploration and exploitation of the state-space by discouraging state-action pairs that have been executed often even if their Q-value is high.

Expansion. If the label of v_N is not a final state, we add a new vertex v_a (with label the state resulting of applying s on s_N) and an edge with endpoint v_a and label (s_N, a), for each action $a \in \mathcal{A}$.

Evaluation. Here we store a value for $V(s_N)$ in case this has not been stored previously yet. If s_N is a final then $V(s_N)$ is the reward of reaching state s_N. Otherwise $V(s_N) = f(s_N)$.

We note that this step is a simplification since our MDP only produces nonzero rewards at the end of episodes. The general case requires taking the accumulated (discounted) rewards from s_0 to s_N.

Update. The Q-values $Q(s, a)$ are updated for each edge with label (s, a) traversed during the selection phase. Each $Q(s, a)$ is defined as follows[6]: Let $P(s, a)$ be the set of all paths in the tree T that start at a vertex with label s, whose first edge has label a, and such that the last vertex is a leaf of T. Given $p \in P(s, a)$, and a vertex v in p, we define $\phi(v) =_{\text{def}} \gamma^{d-1} V(v)$, where d is the distance from the first vertex p_0 of p to v. We then let $Q(s, a) =_{\text{def}} \texttt{mean}\,(\phi(v) \mid v \in p,\ p \in P(s, a),\ v \neq p_0)$. Note: we let $Q(s_n, a) = \infty$ if no edge with label (s_n, a) has been traversed.

Remark A.1 (Transposition tables)**.** In classical formulations of MCTS, the statistics Q, N are computed on vertices and edges, and not on their labels. A *transposition table* [21] may be used in order to include all edges or vertices with the same label when computing these statistics. Deviating from this approach, in this paper we have directly defined the statistics as functions of the labels of edges and vertices, rather than as functions of vertices and edges. Both formulations are equivalent.

Remark A.2. We note that, unlike MCTS$_{\text{nn}}$, the algorithms MCTS$_{\text{solver}}$ require no training whatsoever and can be readily used with any solver on any problem.

[5] Sometimes in the literature the term simulation refers to what here we call the "evaluation" phase.

[6] As shown in Figure 3, the Q-values are efficiently computed in an incremental way as simulations take place, as opposed to being computed from scratch every time it needs to be updated.

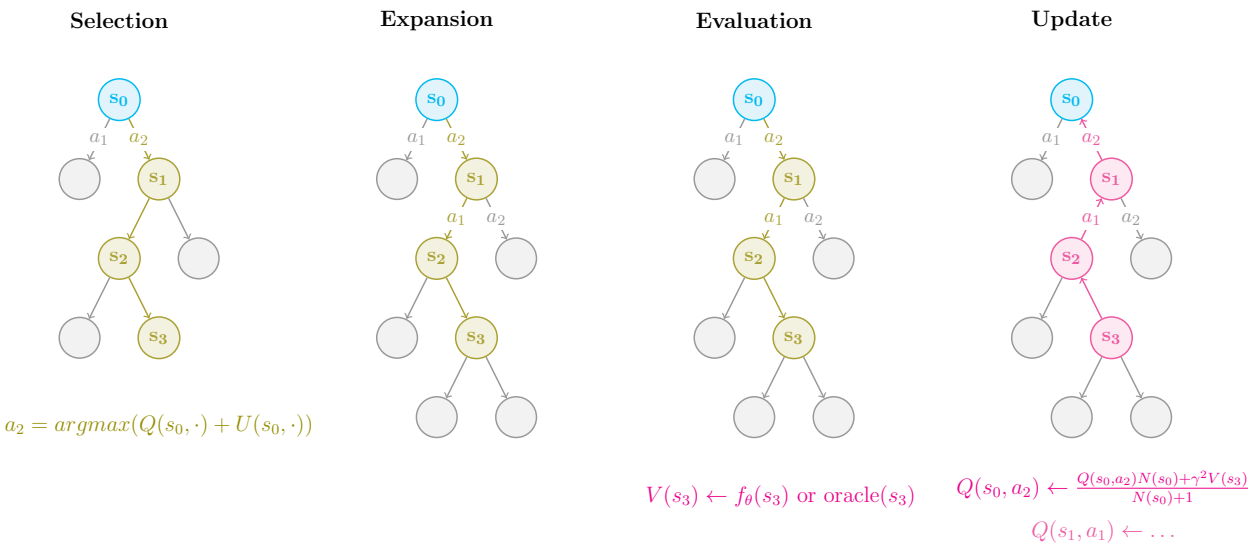

Figure 3: Each simulation of the MCTS algorithm consists on the four steps represented here, and described previously. In the last step one needs to update all Q-values of the state-action pairs lying under magenta edges.

Training of the value neural network Here we focus on the algorithm MCTS$_{\text{nn}}$ in particular, i.e. when the state value approximations are provided by an artificial neural network, which we denote $f = f_\theta$, where θ are the parameters of the network. Similarly as in [57, 59], to fit these parameters one first conducts a training procedure, where an agent successively attempts to solve different randomly generated episodes. For each episode $(s_0, a_0, r_1), \ldots, (s_f, a_f, r_{f+1})$ we extract the pairs $\{(s_1, r(s_1)), (s_2, r(s_2)), \ldots, (s_{f+1}, r(s_{f+1}))\}$ where $r(s_i)$ is the return of the episode starting at state s_i. In general, $r(s_i) =_{\text{def}} r_{i+1} + \gamma r_{i+2} + \cdots + \gamma^{f-i} r_{f+1}$,, and in our particular setting where rewards are nonzero only at the end of the episode, $r(s_i) =_{\text{def}} \gamma^{f-i} r_{f+1}$,. The tuples $(s_i, r(s_i))$ are added to a buffer (a dataset) which is later used to train the network f_θ following the usual gradient descent method. The loss function is obtained from the squared error: $\text{loss}_\theta(s) =_{\text{def}} \texttt{mean}\left[(r(s) - f_\theta(s))^2\right]$. We use L2 regularization during the training of the network, i.e. we add $\alpha||\theta||$ to the loss, where α is a fixed parameter which we set to 1e-4. The learning rate is fixed to 1e-3.

The whole training process is executed as follows: first f_θ is initialized randomly and some episodes are executed by successively sampling from the policy obtained via the MCTS algorithm above. Then f_θ is trained on the data collected during these episodes, and after this the data is deleted. These two steps are repeated consecutively until some performance requirement or computational bound is met.

As discussed in [57, 59], this procedure is a form of "policy-iteration" [62]: "good moves" are used as data to improve the estimations made by f_θ, while better estimations improve the quality of the moves performed during episodes.

A.3 Neural network architecture

Here we provide some details about the neural network architecture used in our experiments concerning MCTS$_{\text{nn}}$. The network is similar to the one used in [58]. The total number of trainable parameters is around $150k$.

The network consists of an initial *simple* block, followed by a number of *residual* blocks, which we set to 2 (see below for a description of these blocks). Following these there is one last convolutional layer, followed by a linear layer with output dimension 1 and a hyperbolic tangent activation. Note that here we have a "single-headed" network, as opposed to the two-headed architecture from [59].

The simple block consists in a two-dimensional convolutional layer, followed by a batch normalization layer and a rectifier nonlinearity (ReLU). The number of input and oputput channels of the convolutional layer are, respectively, $|\mathcal{X}| + |\Sigma| + 1$ (the $+1$ to account for the empty word) and a global parameter, which we set to 64. All other convolutional layers in the network use the same number of input and output channels. All convolutional layers use a kernel of size 3, with stride of 1 and padding of size 1. This preserves the width and height dimensions of the input.

Each residual block is formed by a two-dimensional convolutional layer, followed by a batch normalization layer and

a rectifier nonlinearity. This in turn is followed by another two-dimensional convolutional layer, a batch normalization layer, a skip connection, and finally a rectifier nonlinearity. The skip connection consists in adding the input of the residual block to the output of the layer previous to the skip connection.

B Bibliography

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. Trau smt solver for string constraints. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–5. IEEE, 2018.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 150–166, Cham, 2014. Springer International Publishing.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. Norn: An smt solver for string constraints. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 462–469, Cham, 2015. Springer International Publishing.

[4] Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving NP-Hard Problems on Graphs with Extended AlphaGo Zero. *arXiv e-prints*, page arXiv:1905.11623, May 2019. arXiv:1905.11623.

[5] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, and C. Zhang. Combining string abstract domains for javascript analysis: An evaluation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–57, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[6] Roberto Amadini. A survey on string constraint solving. *ArXiv*, abs/2002.02376, 2020.

[7] Roberto Amadini, Mak Andrlon, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Constraint programming for dynamic symbolic execution of javascript. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 1–19, Cham, 2019. Springer International Publishing.

[8] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning to solve circuit-SAT: An unsupervised differentiable approach. In *International Conference on Learning Representations*, 2019. URL: https://openreview.net/forum?id=BJxgz2R9t7.

[9] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. PDP: A General Neural Framework for Learning Constraint Satisfaction Solvers. *arXiv e-prints*, page arXiv:1903.01969, Mar 2019. arXiv:1903.01969.

[10] Dana Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1):46 – 62, 1980. URL: http://www.sciencedirect.com/science/article/pii/0022000080900410, doi:https://doi.org/10.1016/0022-0000(80)90041-0.

[11] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 255–272, Cham, 2015. Springer International Publishing.

[12] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

[14] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. *arXiv e-prints*, page arXiv:1611.09940, Nov 2016. arXiv:1611.09940.

[15] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *ArXiv*, abs/1811.06128, 2018.

[16] M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 55–59, Oct 2017. doi:10.23919/FMCAD.2017.8102241.

[17] Prithvi Bisht, Timothy L. Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *CCS '11*, 2011.

[18] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. Technical Report MSR-TR-2008-153, Microsoft and, October 2008. URL: https://www.microsoft.com/en-us/research/publication/path-feasibility-analysis-for-string-manipulating-programs/.

[19] Richard F Booth, Dmitry Y Bormotov, and Alexandre V Borovik. Genetic algorithms and equations in free groups and semigroups. *Contemporary Mathematics*, 349:63–82, 2004.

[20] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton, and et al. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI*, 2012.

[21] B. E. Childs, J. H. Brodeur, and L. Kocsis. Transpositions and move groups in monte carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 389–395, Dec 2008. doi:10.1109/CIG.2008.5035667.

[22] L. Ciobanu Radomirovic, V. Volker Diekert, and M. Elder. Solution sets for equations over free groups are edt0l languages. *International Journal of Algebra and Computation*, 26(5):843–886, 8 2016. doi:10.1142/S0218196716500363.

[23] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification (CAV)*, volume 10981 of *LNCS*, pages 183–190. Springer, 2018.

[24] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. *arXiv e-prints*, page arXiv:1704.01665, Apr 2017. arXiv:1704.01665.

[25] Mogens Dalgaard, Felix Motzoi, Jens Jakob W. H. Sørensen, and Jacob Sherson. Global optimization of quantum dynamics with alphazero deep exploration. *npj Quantum Information*, 6:1–9, 2019.

[26] Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. On solving word equations using sat. In Emmanuel Filiot, Raphaël Jungers, and Igor Potapov, editors, *Reachability Problems*, pages 93–106, Cham, 2019. Springer International Publishing.

[27] Joel D. Day, Vijay Ganesh, Paul He, Florin Manea, and Dirk Nowotka. The satisfiability of word equations: Decidable and undecidable theories. In Igor Potapov and Pierre-Alain Reynier, editors, *Reachability Problems*, pages 15–29, Cham, 2018. Springer International Publishing.

[28] Joel D. Day, Florin Manea, and Dirk Nowotka. The Hardness of Solving Simple Word Equations. *arXiv e-prints*, page arXiv:1702.07922, Feb 2017. arXiv:1702.07922.

[29] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. URL: http://doi.acm.org/10.1145/1995376.1995394, doi:10.1145/1995376.1995394.

[30] Timothy Deis, John Meakin, and G. Sénizergues. Equations in free inverse monoids. *Internat. J. Algebra Comput.*, 17(4):761–795, 2007. doi:10.1142/S0218196707003755.

[31] Volker Diekert. *Makanin's Algorithm,* in Lothaire.*, Algebraic Combinatorics on Words*, page 387–442. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2002. doi:10.1017/CBO9781107326019.013.

[32] Volker Diekert. More than 1700 years of word equations. In Andreas Maletti, editor, *Algebraic Informatics*, pages 22–28, Cham, 2015. Springer International Publishing.

[33] Volker Diekert and Murray Elder. Solutions to twisted word equations and equations in virtually free groups. *arXiv e-prints*, page arXiv:1701.03297, Jan 2017. arXiv:1701.03297.

[34] Volker Diekert, Artur Jeż, and Wojciech Plandowski. Finding all solutions of equations in free groups and monoids with involution. *Inform. and Comput.*, 251:263–286, 2016. doi:10.1016/j.ic.2016.09.009.

[35] Volker Diekert, Florent Martin, Géraud Sénizergues, and Pedro V. Silva. Equations over free inverse monoids with idempotent variables. *Theory Comput. Syst.*, 61(2):494–520, 2017. doi:10.1007/s00224-016-9693-1.

[36] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 151–162, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1273463.1273484.

[37] Dominik D. Freydenberger and Mario Holldack. Document spanners: From expressive power to decision problems. *Theory of Computing Systems*, 62(4):854–898, May 2018. doi:10.1007/s00224-017-9770-0.

[38] Vijay Ganesh, Adam Kieżun, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael Ernst. HAMPI: a string solver for testing, analysis and vulnerability detection. In *Computer aided verification*, volume 6806 of *Lecture Notes in Comput. Sci.*, pages 1–19. Springer, Heidelberg, 2011. doi:10.1007/978-3-642-22110-1_1.

[39] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: What's decidable? In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, pages 209–226, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[40] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 277–291, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[41] Audrey Gaymann and Francesco Montomoli. Deep neural network and monte carlo tree search applied to fluid-structure topology optimization. *Scientific reports*, 9(1):1–16, 2019.

[42] Hossein Hojjat, Philipp Rümmer, and Ali Shamakhi. On strings in software model checking. In Anthony Widjaja Lin, editor, *Programming Languages and Systems*, pages 19–30, Cham, 2019. Springer International Publishing.

[43] Artur Jeż. Word equations in linear space. *CoRR*, abs/1702.00736, 2017. URL: http://arxiv.org/abs/1702.00736, arXiv:1702.00736.

[44] Olga Kharlampovich, Igor G. Lysënok, Alexei G. Myasnikov, and Nicholas W. M. Touikan. The solvability problem for quadratic equations over free groups is np-complete. *Theory of Computing Systems*, 47(1):250–258, Jul 2010. doi:10.1007/s00224-008-9153-7.

[45] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine learning: ECML 2006*, volume 4212 of *Lecture Notes in Comput. Sci.*, pages 282–293. Springer, Berlin, 2006. doi:10.1007/11871842_29.

[46] 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.

[47] A. Lentin. *Equations dans les monoides libres*. Mathématiques et Sciences de L'Homme Series. Walter de Gruyter GmbH, 1972. URL: https://books.google.es/books?id=VEXoygAACAAJ.

[48] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient smt solver for string constraints. *Formal Methods in System Design*, 48(3):206–234, Jun 2016. doi:10.1007/s10703-016-0247-6.

[49] Anthony W. Lin and Rupak Majumdar. Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility. *arXiv e-prints*, page arXiv:1805.06701, May 2018. arXiv:1805.06701.

[50] Igor Lysenok. Quadratic equations in free monoids with involution and surface train tracks. *preprint*, 08 2014.

[51] G. S. Makanin. Equations in a free group. *Mathematics of the USSR-Izvestiya*, 21(3):483, 1983. URL: http://stacks.iop.org/0025-5726/21/i=3/a=A05.

[52] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004. doi:10.1145/990308.990312.

[53] Wojciech Plandowski and Wojciech Rytter. Application of lempel-ziv encodings to the solution of word equations. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 731–742, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[54] Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2020. URL: https://openreview.net/forum?id=BkevoJSYPB.

[55] Z. Sela. Word Equations I: Pairs and their Makanin-Razborov Diagrams. *arXiv e-prints*, page arXiv:1607.05431, Jul 2016. arXiv:1607.05431.

[56] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018. URL: http://arxiv.org/abs/1802.03685, arXiv:1802.03685.

[57] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[58] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv e-prints*, page arXiv:1712.01815, Dec 2017. arXiv:1712.01815.

[59] David Silver, Thomas Hubert, Julian Schrittwieser, and et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. doi:10.1126/science.aar6404.

[60] Helge Spieker. Towards sequence-to-sequence reinforcement learning for constraint solving with constraint-based local search. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 10037–10038. AAAI Press, 2019. doi:10.1609/aaai.v33i01.330110037.

[61] Helge Spieker and Arnaud Gotlieb. Towards hybrid constraint solving with reinforcement learning and constraint-based local search.

[62] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, second edition, 2018.

[63] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Progressive reasoning over recursively-defined strings. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 218–240, Cham, 2016. Springer International Publishing.

[64] Ruiyang Xu and Karl Lieberherr. Learning self-game-play agents for combinatorial optimization problems. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '19, pages 2276–2278, Richland, SC, 2019. International Foundation for Autonomous Agents and Multiagent Systems. URL: http://dl.acm.org/citation.cfm?id=3306127.3332083.

[65] Emre Yolcu and Barnabas Poczos. Learning local search heuristics for boolean satisfiability. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 7992–8003. Curran Associates, Inc., 2019. URL: http://papers.nips.cc/paper/9012-learning-local-search-heuristics-for-boolean-satisfiability.pdf.

[66] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.