k-induction bounded model checking[⋆]

Ben Lloyd-Roberts[1], Phil James[1], Michael Edwards[1], and Tom Werner[2]

[1] Swansea University, Swansea, UK
[2] Siemens Rail Automation UK, Chippenham, UK
{ben.lloyd-roberts, p.d.james, michael.edwards}@swansea.ac.uk
WT.Werner@siemens.com

Abstract. Interlockings within the railway domain are computerised safety critical systems responsible for the signalling components directing railway traffic. The formal verification of such systems using bounded model-checking often face limited guarantees of correctness given they operate within a bounded region of k depth. In this work we devise a reinforcement learning (RL) strategy to support SAT-based bounded model checking increasing measures of completeness. We outline a mapping from transition systems traditionally used as BMC abstractions, to Markov decision processes (MDP) typically used as interactive RL environments. Thereafter software agents are fed a reward signal to incentivise state space exploration along acyclic sequences of execution such with the aim of increasing the upper bound k. We test the feasibility of our approach against a series of artificially generated programs with increasingly larger state spaces. A number of metrics suggested in RL literature are then gathered to evaluate the robustness and scalability of the approach, particularly in industrial settings. Finally we produce representations of the state space structure to be fed back to engineers responsible for designing such programs.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

Solid state interlockings are computerised systems responsible for ensuring safe routing of trains through a given railway network. As a vital part of any railway signalling system, interlockings are critical systems regarded with the highest safety integrity level (SIL4) according to the CENELEC 50128 standard. The application of model-checking to Ladder Logic programs in order to verify interlockings is well established within academia and is beginning to see real applications in industry. As early as 1995, Groote et al. [5] applied formal methods to verify an interlocking for controlling the Hoorn-Kersenbooger railway station. They conjecture the feasibility of verification techniques as a mean of ensuring correctness criteria on larger railway yards. In 1998, Fokkink and Hollingshead [6] suggested a systematic translation of Ladder Logic into Boolean formulae.

⋆ Supported by Siemens Mobility UK & EPSRC

Newer approaches to interlocking verification have also been proposed in recent years [7, 8, 9, 10, 11]. This includes work by Linh et al. which explores the verification of interlockings written in a similar language to Ladder Logic using SAT-based model checking. After two decades of research, academic work [12, 13] has shown that verification approaches for Ladder Logic can indeed scale; in an industrial pilot, Duggan et al. [14] conclude: "Formal proof as a means to verify safety has matured to the point where it can be applied for any railway interlocking system." In spite of this, such approaches still lack widespread use within the UK Rail industry. Industrial standards for the railway and related domains increasingly rely on the application of formal methods for system analysis in order to establish a design's correctness and robustness. Recent examples include the 2011 version of the CENELEC standard on railway applications, the 2011 ISO 26262 automotive standard, and the 2012 Formal Methods Supplement to the DO-178C standard for airborne systems. However, the application of formal methods research within the UK rail industry has yet to make a substantial impact. Principally our work aims to address one of the issues hindering its uptake, by removing the need for manual analysis of false negative error traces produced during verification.

Bounded model-checking is an efficient means of verifying a system through refutation. Given an abstracted model of a target system M and some safety properties ϕ, BMC searches for counterexamples up to some precomputed bound k. Search terminates when either an error trace is produced or the bound k is reached. Determining this completeness threshold to sufficiently cover all states is often computationally intractable given it's true value will depend on model dynamics and size. Additionally, the k-induction rule, which checks if the property ϕ holds inductively for sequences of states, is constrained to acyclic graphs for guaranteeing completeness.

An invariance rule allows us to establish an invariant property ψ which holds for all initial states and transitions up to a given bound. Invariants may hold for sub-regions of the state space, meaning complete coverage isn't necessary to learn them. Supporting k-induction with strengthening invariants helps reduce the overall search space for bounded model checking, proving that invariant aspects of the system need not be considered. This can help filter cases where false negative counter examples are triggered by unreachable states. Generating sufficiently strong invariants is a non-trivial process given their construction is heavily program dependent. Usually such invariants require domain knowledge, typically devised by engineers responsible for the program implementation.

We infer two principle challenges to address. First, formulating theoretical and practical frameworks in an academic-industry partnership with Siemens Rail Automation UK to represent interlocking verification as a goal-orientated reinforcement learning task. Second, devising an appropriate strategy for learning invariants within those frameworks.

The aim of this paper is to address this first challenge. Reinforcement learning is a popular machine learning paradigm with a demonstrably impressive capacity for learning near optimal strategies for goal-orientated tasks. Such approaches are well suited to problems where one need systematically learn the behaviour of a deterministic system, record observations regarding different 'states' of behaviour and identify patterns across those states. Generalisation of learned policies across different 'environments' being one of the key challenges in RL, we first aim to learn conceptually simpler goals, such as finding the progressively larger upper bounds for BMC. Understanding how a verification problem can be formulated as one of where machine learning can be successfully applied suggests the promise of learning invariants this way.

2 Preliminaries

2.1 Ladder Logic

Ladder logic programs used to implement interlocking systems allow engineers to define system behaviour in terms of conditions (rungs) and outputs (coils) expressed in a derivation of boolean algebra. Existing SAT-based approaches have already demonstrated translations from ladder logic programs to propositional formulae. The valuation of variables comprising those expressions at any given time is an instance of a program state. The set of reachable states then constitutes the abstractions notion of complete system behaviour.

We can further abstract the program's behaviour by considering any one unique valuation of all variables constitutes a unique program 'state'. Program diagrams are comprised of horizontal lines, referred to as rungs, and logical connectors between symbols, known as coils and contacts. Two types of contacts are used to represent the value of a contained program variable, also referred to as a latch. Open contacts as shown in Fig. 2.2(a) represent the unchanged value of a latch while closed contacts denote their negated value, Fig. 2.2(b). Latches often represent physical inputs to the system but are also used for any other variables required. Coils can represent one of two output types. These are either physical outputs for some system component or a computed value to be used later in the program. Output values are also referred to as latches and can be negated with a closed coil. Each rung is read from left to right and, using the logical connectives shown in Fig. 2.3, represent a condition or rule within the program. Rungs are closed off with a coil representing conditional output on the right hand side. Using coils and contacts enables the expression of logical propositions in our programs.

2.2 Finite State Automata

Observing the equivalence relations above note that certain variables appear repeatedly throughout the formula, primarily the latch crossing. Each rung computes a new coil value to be used in the next execution cycle hence the significance of rung ordering. To illustrate this change over time we introduce a new form of abstract representation, the finite state automaton.

Definition 1 (Finite State Automaton). *A finite state automaton $\mathcal{M}$ given a ladder logic fromula $\psi\mathcal{L}$ is denoted $\mathcal{M}(\psi L)$ and represented by the 4-tuple $\langle Q, \sum, \sigma, I_s \rangle$, where :*

- Q
- $\sum$
- σ
- I_s

2.3 Bounded Model-Checking

Model checking is a formal verification technique stemming from the need to systematically check whether certain properties hold for different configurations (states) of a given system. Given a finite transition system T and a formula F, model checking attempts to verify that $s \vdash F$ for every system state $s \in T$, such that $T \vdash F$. Model checking is typically performed over three distinct phases as described by Baier [20]. Initially the model, that is the mathematical representation of the system, must be constructed. This is accomplished using a model description language, of which there are several. Next the model must be tested through simulation to ensure the abstracted system functions as expected. Provided the constructed model behaves adequately the final stage of modelling requires formulating the properties to be verified according to some specification language. The constructed model and property definitions then enable the model checker to perform runs of the system for state verification. The model checking process culminates in the analysis phase wherein verification results are generated. Properties which hold for all tested states produce a 'safe' output. In the event a state is found to violate any specified properties, that is s 6—= F, a counter example trace is provided by the model checker indicating which state(s) caused the violation. Results may also indicate that the model, property formulation or simulation process are insufficient for verification and therefore require further refinement. If so, the adjustments are made and the pipeline is revised and repeated. Verification means abstraction - "...existing works have successfully mapped ladder logic semantics to propositional logic. Here expressions can be reordered in CNF ready for a SAT-solver."

Abstraction means translating logic to a model - "...any unique valuation of the variables comprising the target program is viewed as an individual system state. This process is expanded in sec 2"

Modelling means we can apply verification - "...decomposing the target system to an abstracted set of states allows model-checking to systematically check each state for safety conflicts"

Verification has limitations - "... while a systematic process there are no guarantees of complete coverage. The longest loop-free path would be useful knowledge" Bounded model checking (BMC) provides a subset of the state space to avoid combin- atorial explosion given the number of unreachable states. A bound is defined as a finite number of transitions between states. However unless all states are reached within that number of transitions then the exploration process is incomplete. This trade-off ensures that unreachable states are not considered but does not guarantee completeness within the bounded model

Primary limitations of the solution presented by Kanso are address in [16, 12, 17, 18]. Due to the inductive step of verification the tool checks to see if a given state satisfies some condition but does not consider the possibility of unreachable states which violate the same safety condition. Consequently if such benign violations are detected the tool risks generating false negative counter examples which requires manual inspection by an experienced engineer. This concept is illustrated in Fig. 3.5. A potential solution to this problem was introduced in subsequent work by James in 2010 [17]. In their exploration of SAT-based bounded model checking, James addresses the limitations of Kanso's approach in terms of state reachability by introducing invariants to suppress false negatives.

Clearly, introducing appropriate invariants to avoid unattainable configurations re- quires a comprehensive understanding of both the abstracted and physical system. Additionally, as system complexity increases so do the number of unreachable states. This could potentially necessitate thousands of unique invariants to suppress triggered false negatives. Currently this process requires manual analysis by specialist engineers making automation highly desirable. Generating sufficiently strong invariants automatically is an extremely complex task, one which has received considerable attention in academic literature. From software engineering techniques [21, 22, 23, 24, 25, 26, 27] to hybrid methods incorporating machine learning [28, 29, 30, 31, 32], researchers have proposed various approaches to invariant finding with varying degrees of success.

2.4 Reinforcement Learning

Reinforcement learning has shown promising results in exploring and exploiting unknown graph structures - solely based on a reward signal.

Machine learning means translating the model to another model - "...Which requires some partial translation, process is expanded in sect 2".

One of three disciplines comprising the field of ML is Reinforcement learning (RL). RL differs from other data driven ML techniques in its ability to automate tasks without the dependency of existing datasets. In this project we leverage the ability of a software agent to rapidly learn new and complex tasks with the existing formal verification techniques applied to railway interlockings. Additionally, we utilise verification approaches for Ladder Logic developed across several projects carried out at Swansea University in co-operation with Siemens Rail Automation UK. We aim to explore if approaches within ML can be used to improve the efficiency of verification whilst also reducing the number of false negatives that are reported by the current approach of model checking.

What could be improved? How do we aim to improve it? What is Reinforcement Learning? How does it benefit verification? How is this work different?

The term interlocking describes the system used to prevent safety property violations in railway signalling. Interlockings serve as a filter or 'safety layer' between inputs from operators, ensuring any changes made to the current railway state does not lead to an unsafe state. Inputs include but are not limited to; operational requests sent by the signaller, train positions detected by track circuits or aspects currently displayed by an array of signals. The recurrent nature of interlockings also allow output values from previous cycles to be used as inputs for subsequent executions.

ML is the theory and practice of utilising large collections of related information, or datasets, to identify and predict patterns. Using intelligently designed algorithms, ML allows us to generalise the behaviour required to perform a given task autonomously. Reinforcement learning (RL) is one such area of ML which leverages human ability to formulate problems and utilise computational means of solving them through a process of accelerated trial and error. RL centres around the concept of a software agent capable of learning optimal policies for interacting with a human specified environment. In this section we discuss the principles of reinforcement learning in general, common algorithms and their relevance to this work.

Deep reinforcement learning (DRL) addresses the limitations of Q-functions in determining an optimal policy. Instead a neural network, called the policy network, can be implemented to approximate the optimal Q-function. This process is known as deep Q-learning where our policy network is a deep Q-network (DQN) [46]. The DQN receives a number of states from the environment as input to approximate a function generating the optimal Q-value such that it satisfies the Bellman equation, see Fig. 5.3 Q-values output by the network are then compared to the target optimum to compute the loss. Ideal for approximation, the deep neural network architecture will then learn do minimise this loss by adjusting weights using SGD and back propagation. This process is repeated for all state-action pairs until the optimal Q-function has been approximated.

To support the training of our policy network we generate a dataset of experiences sampled at each time step. A value N is set to limit the size of our dataset, referred to as replay memory. Using training experiences from replay memory helps avoid learning local trends in the data, see Fig. 5.4. Highlighted data points represent experiences randomly sampled for training, known as a minibatch. Note how the larger replay memory provides more general coverage of past experiences meaning linear correlations in short sequences are less influential on learning. Our first training cycle involves setting a replay memory size N and random weight initialisation for the policy network. Any state preprocessing is then performed before being passed to the DQN. We compute the loss for the given Q-value output by the network for the action listed in our most recent sample from replay memory. In order to compute the loss for the current training iteration, we require some method for calculating the maximum expected discounted reward for the next state. Without a Q-table storing a comparable value for s' we must approximate this value by passing s' to the policy network using different samples from replay memory. Thus we can determine the greatest Q-value output by the network over all actions $a'1,...,a'n$, in s'. Next we perform gradient descent, typically stochastic gradient descent (SGD), to minimise the loss and update the weights of our DQN. We repeat this process for every new time step until an optimal policy is found. Alternatively we incorporate a second DQN, called the target network. This aims to solve the problem of approximating future maximum expected reward with a single policy network. Given weights are only adjusted once the loss has been computed, both q(s,a) and q(s′,a′) are approximated using the same weights. This introduces undesirable properties for learning. As the network weights adjust, q(s,a) approaches the target value while our approximation of q$\star$(s',a′) continues to change. Using a second DQN allows us to adjust a separate set of weights by for outputting target Q-values. These are then adjusted at arbitrary time steps to match the current weights of the policy network.

Theorem 1 (Experience Replay). *The agent's experience at a given time step, et is defined:*

$$e_t = (st, at, rt + 1, st + 1) \tag{1}$$

2.5 MDPs

The essential process behind RL is that of gamification [45, 46, 47]. Imagine a player has been placed in some game environment with no knowledge regarding their objective or the scope of the environment. Depending on the task we wish the agent to learn, our environment is encoded with a rule set which describes performable actions and a reward scheme to either penalise or reinforce agent decisions, see Fig. 5.1. We treat this reward as an in-game score the agent can see. The action set available to the agent influences new states of the environment. Agents are then incentivised to explore their environments through repeated sequences of actions while reacting to each reward. Formally, we define the environment our agent must traverse as a sequential mathematical framework known as a Markov decision process (MDP). RL differs from other supervised learning methods in some fundamental yet subtle ways. Agents, unlike models aim to optimise decision making in order to reach some goal. Supervised and unsupervised learning typically involve training a statistical model on real world samples, necessitating a data set for training. RL, through repeated simulation can produce an agent having learned an optimal policy of which humans would be incapable. That is to say the agent learns independently without the need for external observations. RL interpreters issue positive or negative reward depending on agent behaviour. Action sequences we wish to reward are done so with a positive score. Otherwise the agent is assigned a negative score. Consequently, agents will attempt to find the optimal policy to maximise cumulative positive reward and minimise negative rewards. This can be expressed probabilistically $P(s', r|s, a)$. Given the current state s and an action a is performed, what is the probability of a subsequent state s' leading to a reward r. Depending on the learning strategy chosen, agents will attempt to maximise the cumulative reward, meaning future decisions will be considered at each step. What constitutes the terminal step depends on the environment and how it is imple- mented. In our case we require some metric of agent performance other than reward to determine whether the environment should reset. Such problems are referred to as episodic tasks.1. To this end we introduce a terminal condition which computes the total number of unique transitions and states to check if the agent has discovered them all. Throughout the training process it is expected the agent will minimise the number of necessary transitions to the point an optimal path is learned. Continuous tasks, which have no terminal step, face the challenge of enumerating future cumulative reward for infinitely many steps. One solution to dealing with an infinite horizon is discounting.

2.6 Policy Learning

A policy is a rule set learned by an agent which determines the next best action. This is determined based on samples from the current probability distribution over any given action yielding a positive reward given the state.

Definition 2 (Policy). *A probability distribution $\pi(a|s)$ describes the likelihood of action (a) returning a positive reward given environment state (s).*

Definition 3 (Trajectory). *A sequence τ, comprised of the states s0,...,sn, actions a0,...,an, and rewards r0,...,rn experienced by an agent following a policy π*

The start state s0 is randomly sampled from a start state distribution, Is in our case. Continuous random sampling of the probability distribution is done to avoid repetitive behaviour. Initially agents have no understanding of their environment or which actions lead to the greatest cumulative positive reward. Most initial actions will result in negative scoring as parameters adjust over time. Agents are expected to perform poorly at first, particularly with larger action sets given the potential number of sequences to sample. It is this evolutionary process that allows RL agents to generalise so well when learning long term goals. In the context of this project we have two options for representing our environment. 1) Using maximum coverage where the environment consists of all potential system configurations or 2) As the subset of all reachable states produced by executions of the pelican crossing program. The first environment consists of 4096 states, each with twelve directed edges representing the change in value for a single variable. Thus our action set comprises all propositional variables in V where a single action entails negating one boolean. The second environment consists of six states, those reachable by the system. Each state in this environment changes based on the value of input variable pressed, resulting in an action set 0,1.

Theorem 2. *(Optimal Policy): iven a finite MDP, there exists at least one optimal policy J, over a set of parameters θ, such that: $J(\theta) = E\pi[r(\tau)]$ Finding the optimal policy requires some measure of decision quality. These are referred to as value functions*

2.7 Value Functions

Value functions return an expected reward, which determines the overall benefit of a given policy. They can be differentiated by the observations they use to determine the value. So called state-value functions, denoted vπ, provides the value assigned to a state under policy π. Informally, state-value functions produce a valuation assigned to the current state which determines the expected reward from subsequently adhering to the same policy π. We define the value function vπ(s):

Theorem 3. *(Optimal state-value function): An optimal policy J(θ) has an optimal state-value function for all s $\in$ S, defined:*

$$v \star (s) = max \pi v \pi (s) \tag{2}$$

Informally the optimal state-value function returns the maximum possible expected return of any policy for each state. Additionally action-value functions, denoted qπ, provides the value assigned to an action under policy π. Action-value functions, also knowns as Q-functions, produce a quality valuation assigned to the current action which determines the expected reward from subsequently adhering to the same policy π. We define the value function qπ(s):

The output of our Q-function is known as the Q-value.

Theorem 4. *An optimal policy, Jθ has an optimal Q-function for all s $\in$ S and all a $\in$A:*

$$Q \star (s, a) = max \pi Q \pi (s, a) \tag{3}$$

Informally, the optimal Q-function returns the maximum possible expected return of any policy for all state-action pairs. The Bellman principle of optimality [48], Eq. 5.7, states for any state-action pair at the current time step, the expected return from an initial state s, taking action a and following the optimal policy J(θ) thereafter is equal to the expected reward from taking action a in state s, plus the maximum achievable expected discounted return from any subsequent state-action pairs.

The Bellman optimality equation is an integral metric used to learn the optimal Q-function which in turn is used to learn the optimal policy. Given an optimal Q-function action a^{prime}, a Q-learning algorithm will find the best action a$'$ which maximises the Q-value for s$'$.

2.8 Reward/Goal Shaping

Q-learning refers to a policy learning method which uses Q-functions to calculate maximum expected future reward. Learning is formulated as an iterative process of parameter adjustment known as value iteration.

2.9 Exploration Strategy

In the previous section we briefly discussed initial exploration rates to compensate for zeroed Q-tables. A popular approach to setting exploration rates is the epsilon greedy strategy [49, 50, 51, 52]. In order to balance the exploration-exploitation trade-off [53, 54], we set an exploration rate ϵ between 0 and 1 to dictate whether the agent prioritises exploratory or exploitative behaviour. Values nearing 0 represent a greedy strategy where the agent is more likely to exploit previous knowledge. Epsilon values nearing 1 therefore encourage exploratory behaviour. Additionally we assign an epsilon decay value to decrease the exploration rate for each episode. Including an epsilon decay value provides some beneficial learning properties. As the agent learns more about the environment it relies less on exploration. Instead we expect to have observed a sufficiently comprehensive understanding of the environment thus settle on a purely greedy strategy. We also influence exploration and exploitation at every time step. First we randomly generate a number, i between 0 and 1 to compare with the current value of ϵ. If i ¿ ϵ the agent selects an action with the highest Q-value for the state-action pair. Where i ¡ ϵ, the agent randomly samples an action to explore the environment. In this project we aim to train an RL agent to learn the optimal path through our pelican crossing state space. Say we issue a positive reward of 1 for every new state discovered and -1 for repeated transitions. To update the new Q-value we approximate the right hand ride of the Bellman equation. First, we iteratively compare the loss of the current Q-value and the optimal Q-value for each state-action pair. Our aim is to minimise this loss until convergence.

Designing appropriate reward systems is arguably one of the greatest challenges in RL. Reward shaping is often performed specifically for the application

3 Mapping Formal Methods to RL

4 Implementation

4.1 LL program generator

4.2 Learning Environment

Constructing an environment is arguable the longest phase of training an RL agent. To ensure our agent performs well in real-world applications, the learning environment must be sufficiently representative of the problem domain. Through what is essentially the process of gamification, we design an environment that records agent actions and issues a reward based on the behaviour we wish to reinforce. This process presents a number of distinct challenges in our context. First, given a simple ladder logic program, is it possible to encode its functionality and constraints using an imperative programming language. Second, an efficient method of indexing visited states during exploration to determine a k-number of steps before the agent is forced to revisit states. Finally, we must identify the optimal reward scheme and learning parameters to enforce the desired behaviour. We design an openai-gym [102] like environment to train our agent (Appendix A). Consider the environment as a game. The agent makes successive transitions between states from some arbitrary start position sampled from Is. Recall the set I defined our physical inputs received by the ladder logic program shown in sect. 3.2. At each step the agent is presented with a binary decision to make, selected from our action set A, where: $A \equiv I$

Our single input variable pressed is valued either 0 or 1. Once an action is selected, each transition must then be computed as a single execution of the ladder logic program. We define a transition function, Alg. 1, which receives an action ai, and the current state as input. Depending on whether the agent selects 0 or 1, the function returns a new state with an updated valuation. These are the transitions σ : q $\sum\rightarrow$ q$'$ defined in sect 3.3 w.r.t finite state automata. Under the assignment rules of our transition function, the agent will eventually discover all six reachable states. Fig. 8.1 represents the finished environment with all paths the agent can explore. Note this is essentially a more complete version of our automaton in sect. 3.3. The formulation of our environment means the agent benefits from having to traverse a directed graph. This can often reduce the number of decisions to consider at each step particularly for large graphs with many edges. Additionally our agent samples from a small action set, meaning fewer potential transitions from a single node. With our transition function generating a total of six states, we continue to implement a collection of traces to record agent exploration. With each action we store the ensuing transition, a 3-tuple (q,a_i,q$'$), and the new state q$'$ in separate hash tables. Our traces let us determine whether the current state or latest transition is unique given we have a record of past steps.

4.3 Agent Training

Given we wish to find the maximal depth of the state space without repeating transitions, a positive reward is issued for new discoveries. Contrarily we issue a negative reward for adding previously recorded states with a harsher reward for repeating transitions. In regard to the exploration-exploitation trade-off we issue a larger reward for discovering new transitions over new states. This is primarily due to agent's prioritising actions that are guaranteed to issue positive reward while attempting to avoid negative scores. To avoid infinite loops we introduce a terminal condition which resets the environment when all states and transitions have been discovered. To encourage the agent to satisfy the terminal condition as soon as possible we issue a large positive reward once it has been reached.

We can use DQN(s) to approximate the optimal Q-function and, consequently the optimal policy. o this end we implement two DQNs, the first policy network to train our agent and the second target network to approximate our target values. DQNs train differently to Q-tables in their use of replay memory. We implement a basic Replay class to store sequences of 5-tuples (s,a,s$'$,r), where s is the current state, a denotes the action taken, s$'$ representing the next state and r being the reward. We then define a max capacity N, which dictates the number of experiences our agent remembers. One the max capacity is reached, the agent pops the oldest entry and adds the latest experience to memory. During the training process our agent samples experiences according to some arbitrary batch size. If the replay memory is less than this batch size, we cannot sample. Therefore we introduce some basic condition checking to determine whether sampling is feasible at any given time step. In regard to the selected exploration strategy, ϵ, we employ the same implementation used for Q-tables. Here we set an initial value for ϵ, a minimum value and our ϵ-decay rate. Concerning our DQN architecture, we construct an input layer of 12 nodes, one for each $v_i \in$ V , two fully connected hidden layers and a binary output layer to reflect the action set A. To avoid the problem of moving targets, we set an update rate of 10 epochs for our target network. That is to say we update the weights of the target network for every 10 passes through our policy network. Our chosen optimiser is Adam [103], an SGD variant based on AdaGrad [104] and RMSProp [105], due to its speed and accuracy.

Network Architecture

5 Experiments

5.1 Extended state-spaces

The agent learns over time to avoid repeated actions until an optimal trace is output. This trace can then be used to find the maximal depth of the state space without looping for all states in Is, if such a path exists. We discover through a series of experiments that there are two sets of unique transitions for all start states such that no transition is repeated twice. Concerning the discovery of loop free paths, as discussed in [17], it may be possible to either explicitly state which path should be verified by the SAT-solver or leverage discoveries made by the agent in identifying a loop free path. By increasing the max step threshold within our environment allows the agent sufficient time to explore the state space and understand what satisfies the terminal condition. Consequently, provided a long enough training time, the agent will always find an optimal path. In fact it was discovered through a series of experiments, specifically ones with a low ϵ-decay rate, that there exist two optimal paths for every start state. These traces can then be analysed to determine the maximum k-steps before looping.

5.2 Interlocking examples

Displayed equations are centered and set on a separate line.

$$x + y = z \tag{4}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. **??**).

Table 1. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic

Theorem 5. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [**?**], an LNCS chapter [**?**], a book [**?**], proceedings without editors [**?**], and a homepage [**?**]. Multiple citations are grouped [**?**,**?**,**?**], [**?**,**?**,**?**,**?**].