Contribution Title*

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. Formal verification is an increasingly popular approach to guaranteeing safety in critical systems. The railway in particular shares a history with formal methods spanning decades[1,2,3,4]. In this thesis we explore the limitations of current verification techniques applied to railway interlockings and how machine learning can be used to support SAT-based model checking in particular. We provide a brief overview of both disciplines and the necessary background knowledge required to replicate our approach. Finallywe present the technical implementation of a reinforcement learning (RL) environment modelling the state space of a ladder logic program responsible for encoding basic safety properties for a pedestrian controlled light crossing. An RL agent is then successfully trained to output two sets if minimal transitions for traversing the state space to aid in identifying loop free paths for further verification.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

1.1 Motivations

Advancements in machine learning (ML) research in recent years have marked an inflection point in private and public sector investments in predictive modelling. Consequently the development of autonomous intelligent systems has facilitated an unprecedented level of its adoption in complex systems. The range of applications are staggering, from autonomous vehicle navigation through computer vision, to assisted medical diagnostics. 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 large 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

* Supported by organization x.

whilst also reducing the number of false negatives that are reported by the current approach of model checking.

The current SAT-based technique used for verification faces certain limitations which this project intends to address. Due to the scale and complexity of modern critical systems such as interlockings, contemporary approaches require mathematical abstractions of the system's technical implementation. Finite state machines1 are a well-established and robust modelling technique that can be traced to 'computing machines' presented by Turing in 1936 [15]. A system's state space can be viewed as a graph. Each node representing a unique. configuration of that system where edges between nodes2 denote the transition from one state to another. It is by verifying each state of the system according to some formal safety properties that we can guarantee correctness. Even the most basic systems require loops to perform a predefined number of, or infinitely many, repeated actions. The previously mentioned approach of model checking faces limitations in this regard. Ultimately our objective is to determine the depth3 of a given state space from some initial state to identify a k-number of steps for loop free paths. That is to say we wish to reachability from an initial state to some kth state such that no transition or state is observed more than once. We therefore propose the introduction of an RL software agent to sufficiently determine the maximal depth through exploration and exploitation of the state space and transition rules. We first formulate the task as an optimisation problem. Initially we define an environment similar to a state space representation of a simple ladder logic program used for pelican crossing verification. This environment is then used to train and RL agent to first, discover all unique states and transitions within that state space and output a trace for said path4. This can then be used to reduce the search space of loop free paths.

2 Railway Signalling

2.1 Components of Signalling

Railways have always been hazardous environments. The mass and velocity of locomotives, coupled with their restricted movement and fixed trajectory presents a risk to passengers and staff alike. As a result, accidents occurred frequently in the early days of railway operation. Risks of head on collisions were exacerbated by single lane tracks, requiring the notion of movement authority and a set of rules to safely direct railway traffic. Thus the principles and practice of signalling were introduced. Movement authority can be granted or withdrawn based on the current state of the track to ensure that trains proceed safely, without risk of collision. Signalling on the other hand determines how and when movement authority is changed. We now introduce some basic terminology to describe the composition of railway networks1 in order to understand the principles of signalling. In this section we briefly discuss the function of four key railway components; signals, track segments, routes and points. To grant or remove movement authority necessitates a system capable of informing train operators of such changes. Thus physical signals were introduced. Initially human operators, referred to as policemen, monitored railway activity using a stopwatch to time departures and hand signals to notify train drivers2. The meaning behind each signal is indicated by the signal's aspect3. Traditionally fixed signals placed near track edges had physical aspects in the form of semaphore signals. Most modern signals are electronic, providing visual feedback in the form of coloured lights, denoted by the prefix s in Fig. 2.1. Common aspects are denoted by three colours; Red indicating the driver should halt, yellow suggesting drivers should proceed with caution and green to indicate a clear track ahead. In order to track train positions railway networks are built up of smaller track seg- ments, denoted by the prefix t in Fig. 2.1. Each segment is fitted with an electronic circuit to indicate whether it is occupied by a train or not. Signals are usually placed at block intervals along these track segments to prevent any two trains from occupying the same section. Consequently engineers can be notified of any future obstructions before they proceed to the affected region. Each track segment, by definition, requires a clear start and end point. They do not necessarily construct a linear track and will often lead to a split or junction in the network. Sections where track segments lead to such deviations are referred to as points, denoted by the prefix p in Fig. 2.1. Points are typic- ally lever operated sections of track regulated by a nearby computerised or human controller. Finally we introduce the idea of routes, the physical path that a train may follow along the railway. Route setting can be performed manually by a human operator or automatically by a computerised system. The potential routes available to a train are then subject to the number of points and their positions. Routes are then set by altering signal aspects and points along the necessary track segments while also considering the routes of other trains occupying the network. It is clear to see that route setting is a critical stage of railway operations and necessitates some form of verification. Here we introduce the notion of an interlocking.

2.2 Interlockings

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. That is to say any route setting or track changes must be verified by the interlocking system before any physical change takes place. It is also the interlocking's responsibility to prevent any attempts to force unsafe actions. In the event a fault is detected during routine operation the interlocking should initiate a fail safe state. Typically this means setting all signal aspects to red and bringing traffic to a halt until the issue is resolved. The original 19th century interlocking systems comprised of mechanical levers for controlling railway components and a locking bed to prevent operations conflicting with safety conditions. As railway traffic increased so did operational complexity and risk. Fortunately by the 1930s relay interlockings had superseded mechanical levers, introducing buttons and electrical circuits to both monitor train positions and partially automate the signalling process. In the 1980s significant legislation was passed allowing the embedding of microprocessors in safety-critical systems. Solid state interlockings (SSIs) were developed shortly after. Contemporary interlocking systems are largely software controlled, offer greater flexibility for maintenance and support automatic route setting through computerised controllers. Such levels of complex automation require interlockings to process large sets of data from varying sources before altering the state of the railway. Embedded systems within trains, track circuits and other sensor arrays transmit data back to the interlocking pertaining to the current state of the railway network. Every mutable component must be considered precisely when computing subsequent states of a changing railway network. This is particularly challenging given computerised railways operate as immensely complex distributed systems and therefore are subject to concurrency issues. As such, interlockings require a robust method for validating and verifying routes. Consequently interlockings have also become increasingly complex, presenting new challenges in terms of how they perform verification. Interlocking systems require accurate information in real-time to monitor the current state of the railway network4. First the interlocking must read input values from a range of various sources. These values often pertain to the mutable railway components discussed earlier. 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. Once the necessary inputs have been received by the interlocking they are used to compute output values to pass through the ladder logic program. For now readers should consider this stage as the layer responsible for ensuring proposed changes to the railway composition comply with a strict set of safety properties. The specific functions and semantics of ladder logic programs are discussed in section 2.3. Finally outputs verified by the ladder logic program are committed back to the initial range of input sources. Here any safe track changes can be made. Additionally the signaller is sent an update from the interlocking providing information regarding the latest control cycle. This process repeats indefinitely until the fault detection system identifies a problem.

2.3 Ladder Logic Programs

In the early days of process control, engineers required a way of implementing electrical circuits that could behave logically. Initially this was achieved using racks of electrical relays implemented with combinational logic. Schematic diagrams were then used to represent various relay devices and their physical connections. Such schematics were designed and documented using ladder logic to represent the circuit structure. Ladder logic has since evolved into a graphical programming language widely used in programmable logic controllers (PLCs). 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 atomic propositions in our programs. However we still require a way to represent connectives. Fig 2.3 shows that two types of connections can be established between contacts. Horizontal lines denote conjunction, equivalent to a logical AND connective in propositional logic5. Vertical lines between contacts denote disjunction, equivalent to a logical OR connective in propositional logic.

3 Verification and the Railway

As discussed in our introduction, formal verification of large systems is often infeasible without abstractions since there are too many variables to consider at once. However abstraction can often result in a trade-off between maximising coverage and capturing sufficient detail of the target system. We thus introduce the notion of validation vs verification. Safety properties for example are derived through a series of different processes; first in a requirements document expressed in natural language. This articulates what the system should be capable of but not necessarily how it achieves this. The implementation of said safety properties is defined in a specification document. These documents are defined in a formal specification language and are not necessarily reflective of the system requirements. Validation therefore ensures correctness according to the system requirements where verification concerns correctness w.r.t the formal specification. Consequently formal verification can only guarantee correctness if the specification is correct. For railway interlocking, safety properties are defined in control tables expressed in ladder logic.

4 Mapping Formal Methods to RL

5 Reinforcement Learning

Humans are exceptionally unique in their ability and aptitude to learn through experience. From gestation to formal education our body and brain learn to respond to their environ- ment. Exposure to new information and a means of processing that information is how we learn to function independently. Consider the ability to communicate using natural language, which is often taken for granted. This is generally learnt through observation, repetition and correction.

ML is conceptually similar in this regard. 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. Fig. 4.1 illustrates a high level overview of the main topics within ML. The three primary approaches are supervised, unsupervised and reinforcement learning (RL). Each approach differs from the other based on the algorithms they employ and chiefly, how observations are drawn from existing data. Selecting the appropriate methodology is heavily influenced by the application domain. In this section we provide a general overview of ML theory and practice, primarily in supervised learning given its relevance to our work. In supervised learning our objective is to learn a general function which maps a given input x to the correct output y. This learning approach differs from its counterparts by utilising training labels, see Fig. 4.2. Labels are typically represented by a vector, Y , containing discrete values associating inputs from a training set X to their correct outputs. Ultimately our goal is to train a sufficiently robust model to understanding the underlying structure of the data distribution. Consequently the model is able to take observations from previously unseen data and predict the correct output without knowledge of the data label.Learning from past experiences is a capability of most intelligent life. Humans in particular are exceptional in this regard. The rate at which we process information, reason logically and derive conclusions based on what we can observe is exceptional compared to other animals. We do however, face certain biological and physical limitations that influence our rate of thinking. Contrarily, computers have no ability to operate independently without a set of explicit instructions. Yet, recent advances in ML have proven machines can not only excel at complex tasks [41, 42, 43] but learn to match, if not surpass, human performance levels [44]. 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.

The above definition provides us with the necessary elements to construct our own propositional representation of a ladder logic program. First we distinguish between input and output variables. Each rung provides us with an individual coil and corresponding latch denoting the conditional output. Unifying all coils in the program under a single set produces the following definition:

5.1 Deep Learning

In traditional ML approaches we are often expected to use a significant amount of prior knowledge regarding our application domain. Without a clear understanding of both the data.

Artificial neural networks (ANN) predate the field of deep learning by several decade. Originating in work presented by McCulloch and Pitts [40], neural networks are a computing system modelled after biological networks of neurons found in the nervous system. Each neuron in the network is represented by a connected node which can 'fire' some signal based on activity in a previous layer. A simplistic ANN architecture can be seen in Fig. 4.7. Ultimately, neural networks are used as universal function approximators. Given some input we aim to learn the function for a given output. This allows us to predict future outputs for unseen inputs since we have a general approximation of their relationship. Application domains have a significant influence on the NN architecture used. We observe a number of layers and nodes in Fig. 4.7, forming the network. Different types of layers adjust the transformation applied data received by each neuron. Fig. 4.7 shows a dense, fully connected hidden layer which simply connects the output of one neuron to another 2. Similarly a convolutional layer is a specific type used in image processing. This however, is beyond the scope of our work. Input layers are how we present initial information to the network, usually samples from a dataset. Using our earlier example for classification, an input layer would consist of 12 neurons, each representing a feature (single boolean) of the current state. Relationships between neurons are denoted by a weighted edge which is assigned some value between 0 and 1. The total weighted sum of all edges connected to a single neuron are subsequently passed to an activation function. Typically this is some non-linear function which maps the weighted sum to another value between 0 and 1. In deep learning this process is continued for every layer until the output layer is reached. The configuration of the output layer depends entirely on the task we are trying to learn. For a binary classification problem the output layer would consist of two neurons, valid and invalid according to our earlier example. The exact value output by the network will be some probability of the output neuron resembling the ground truth.

As discussed in the previous section, Q-tables scale poorly to large environments with many actions. 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:*

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

5.2 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.

5.3 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.

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

Theorem 3. *content...(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 4. *(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*

5.4 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 5. *(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 6. *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$'$.

5.5 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.

5.6 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

6 Implementation

6.1 LL program generator

6.2 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.

6.3 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

7 Experiments

7.1 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.

7.2 Interlocking examples

Displayed equations are centered and set on a separate line.

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

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

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

Theorem 7. *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 [1], an LNCS chapter [2], a book [3], proceedings without editors [4], and a homepage [5]. Multiple citations are grouped [1–3], [1, 3–5].

References

1. Author, F.: Article title. Journal **2**(5), 99–110 (2016)
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). https://doi.org/10.10007/1234567890
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017