Reinforcement Learning State Space Properties for Model Checking of Interlockings*

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

Swansea University, Swansea, UK
{ben.lloyd-roberts, p.d.james, michael.edwards}@swansea.ac.uk

Abstract. Railway interlockings serve as safety layer within railway signalling systems by ensuring proposed signalling requests are safe given the current state of the railway. As a vital part of any railway signalling system, interlockings are critical systems developed with the highest safety integrity level (SIL4) according to the CENELEC 50128 standard. The application of formal methods, in particular model-checking, in order to verify interlockings operate correctly is well established within academia and is beginning to see real applications in industry. However, the uptake of formal methods research within the UK rail industry has yet to make a substantial impact due to current approaches often producing false positives that require manual analysis during verification. Here, it is accepted that so-called invariants can be added to reduce the number of such false positives, however automatically computing these invariants remains a challenge. In this work we present a first approach illustrating how reinforcement learning can be used to learn properties of a state space for a given railway interlocking. Understanding how a verification problem can be formulated as one where machine learning can be successfully applied is the first step towards automated learning of invariants.

Keywords: Reinforcement Learning · Interlocking · Model Checking.

1 Introduction

Interlockings serve as a filter or 'safety layer' between inputs from operators, such as route setting requests, ensuring proposed changes to the current railway state avoid safety conflicts. 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

* Supported by Siemens Mobility UK & EPSRC

larger railway yards. In 1998, Fokkink and Hollingshead [6] suggested a systematic translation of Ladder Logic into Boolean formulae. 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 of the search space. Additionally, the k-induction rule, which checks if the property ϕ holds inductively for k sequences of states, is constrained to acyclic graphs for guarantees of 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 being 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'.

2.2 Labelled Transition Systems

Each rung computes a new coil value to be used in the next execution cycle hence the significance of rung ordering in ladder logic programs. Labelled transition systems are commonly used to model this change in valuation over execution cycles.

Definition 1 (Labelled Transition System). *A labelled transition system $\mathcal{L}$, given a ladder logic formula ψL is denoted $\mathcal{L}(\psi L)$ and represented by the 4-tuple $\langle S, R, S_0 \rangle$, where S is a finite set of states, R is a transition relation and S_0 is the set of initial states.*

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 is a principle machine learning paradigm which models sequential decision making problems with respect to some goal-orientated task as the optimal control of some incompletely-known MDP, known as the environment [1]. Given a permitted set of actions to perform over a series of discrete time steps, a software agent is trained to interact with and observe changes in its environment based on intermittent reward signals. Over a process of accelerated trial-and-error agents learn a function, or policy, mapping states to optimal actions likely to return the greatest cumulative future reward. A formal definition of the RL framework and its components *w.r.t* a ladder logic model-checking problem is provided in section 3.

3 Mapping Formal Methods to RL

$$crossing' \leftrightarrow (req \wedge \neg crossing), \tag{1}$$
$$req' \leftrightarrow (pressed \wedge \neg req), \tag{2}$$
$$tlag' \leftrightarrow ((\neg corssing') \wedge (\neg pressed \vee req')), \tag{3}$$
$$tlbg' \leftrightarrow ((\neg corssing') \wedge (\neg pressed \vee req')), \tag{4}$$
$$tlar' \leftrightarrow crossing' \tag{5}$$
$$tlbr' \leftrightarrow crossing' \tag{6}$$
$$plag' \leftrightarrow crossing' \tag{7}$$
$$plbg' \leftrightarrow crossing' \tag{8}$$
$$plar' \leftrightarrow \neg crossing' \tag{9}$$
$$plbr' \leftrightarrow \neg crossing' \tag{10}$$
$$audio' \leftrightarrow crossing' \tag{11}$$

For the above ladder logic propositional formulae, we define a set of input and output variables, $I = pressed$ and $C = \{crossing, req, pressed, tlag, tlbg, tlar, tlbr, plag, plbg, plar, plbr, audio\}$. $C' = \{c' \mid c \in C\}$ respectively. A ladder logic formula Ψ with n rungs, has the form:

$$\Psi \equiv ((c'_1 \leftrightarrow \psi_1) \wedge (c'_2 \leftrightarrow \psi_2) \wedge ... \wedge (c'_n \leftrightarrow \psi_n)), \tag{12}$$

such that $\forall i, j,\ 1 \leq i, j \leq n$:

- $c'_i \in C'$,

- $i \neq j \rightarrow c'_i \neq c'_j$
- $vars(\psi_i) \subseteq I \cup \{c_i, ..., c_n\} \cup \{c'_1, ..., c'_n\}\}$

$$V_I = \{\mu_I | \mu_I : I \rightarrow \{0,1\}\} = \{0,1\}^I \tag{13}$$
$$V_C = \{\mu_C | \mu_C : C \rightarrow \{0,1\}\} = \{0,1\}^C \tag{14}$$

The semantics of a ladder logic program is defined as a function of input and output variable valuations: $[\psi] : V_I \times V_C \rightarrow V_C$. Therefore $\psi(\mu_I, \mu_C) = \mu_{C'}$, where

$$\mu_{C'}(c_i) = [\psi](\mu_I, \{c_i, ..., c_n\}, \{c'_1, ..., c'_{i-1}\}) \tag{15}$$
$$\mu_{C'}(c) = \mu_C \textbf{ if } c \notin \{c_1, ..., c_n\} \tag{16}$$

A Ladder logic labelled transition system $LTS(\psi)$ is defined as the three tuple $(V_C, \rightarrow, V_0)$:

- $\mu_C \rightarrow \mu_{C'}$ *iff* $[\psi](\mu_C, \mu_I) = \mu_{C'}$
- $V_0 = \{\mu_C \mid init(\mu_C)\}$,

where the function $init()$ produces the initial valuation of variables μ_C, setting all as false.

We now define the finite MDP, or environment $\mathcal{E}$ used to represent the ladder logic program. A Ladder logic MDP $M(\psi)$ is a five tuple $\langle S, \mathcal{A}, P_a(s, s'), R_a(s, s'), \gamma)\rangle$, where

- $S = V_I \cup V_C$, an observation space to represent the environment state at discrete time steps.
- $\mathcal{A} = V_I$, set of formally defined actions which change the observations space
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t, a_t)$
- $R_a(s, s')$ is a reward function fed back to the agent at each time step
- γ is a discount scalar applied to the reward estimate for future time steps.

Given a user defined environment $\mathcal{E}$ comprising a set of states S, a set of actions $\mathcal{A}$ and a state transition function $f : S \times \mathcal{A} \rightarrow S^+$, a software agent is trained to learn a policy π, a function which maps optimal actions a to take from a given state s at time step t given a positive or negative reward signal $r \in \mathcal{R}$ [45, 46, 47]. Agents then attempt to converge toward an optimal deterministic $\pi(s)$ or stochastic policy $\pi(a|s)$ with the aim of learning which state-action pairs maximise cumulative future reward. Successive agent observations regarding $\mathcal{E}$ used to update π are stored as a *trajectory* τ. This sequence comprises states, actions, and rewards experienced by an agent at contiguous time-steps: $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_h, a_h, r_h)$ where h denotes the horizon, a time-step beyond which rewards are no longer considered.

Reward signals are often sparse, meaning agents are expected to naively interact with $\mathcal{E}$ over multiple time-steps before receiving feedback. Hence identifying potential rewards over future time steps, an *expected return* given the present

state s_t, becomes desirable: $G_t = \sum_{i=0}^{T} \gamma^i r_{t+i}$. T refers to the time step a task-specific terminal condition is met. Subsequently the environment returns to the initial state $I_s \subset S^+$. In continuous or sparsely rewarded episodic tasks, a scalar discount factor $\gamma \in [0, 1]$ is applied to successive rewards at each time step. This helps enumerate expectation over a potentially infinite horizon, weighting rewards according to their distance from the current time step.

Given a finite MDP, there exists at least one optimal policy $\pi^*(s)$. Finding the optimal policy requires some measure of decision quality. Action-value functions from Q-learning $Q_\pi(s, a) = E\left[G_t \mid S_t = s, A_t = a\right]$ help provide an estimate of which state-action pairs under a given policy are likely to maximise future returns. An optimal policy, π^* has an optimal Q-function which returns the maximum possible expected return of any policy for all state-action pairs, $Q^\star(s, a) = \max_\pi E\left[R_t | s_t = s, a_t = a, \pi\right]$. The Bellman principle of optimality [48], 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 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.

$$Q^\star(a, s) = E\left[r_{t+1} + \gamma \max_{a'} q^\star(s', a')\right] \tag{17}$$

Using the Bellman optimality equation as a direct iterative update to the Q-function, $Q_{i+1}(s, a) = E\left[r_t + \gamma \max_{a'} Q_i(s', a')\right]$ is often intractable for complex problems. Since Q-value estimates are approximated for each discrete observation of s, such methods fail to generalise to new observations. Consequently, a neural network J parameterised by weights θ, the Q-network, is introduced to approximate the Q-function indirectly [46]. Training is performed by minimising some loss function $L_i(\theta_i)$ which changes at each iteration i given the target Q-value approximation, $\hat{y}_i = E_{s' \sim \mathcal{E}}\left[r_t + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | (s, a)\right]$ depends on θ.

$$L_i(\theta_i) = E_{s, a \sim p(\cdot)}\left[(\hat{y}_i - Q(s, a; \theta_i))^2\right] \tag{18}$$

where $p(s, a)$ refers to a probability distribution over states s and actions a. Q-network weights are then updated according to the Q-learning algorithm with stochastic gradient descent.

$$\nabla_{Q_i} L_i(Q_i) = E_{s, a \sim p(\cdot); s' \sim \mathcal{E}}\left[(r_t + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta)) \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{19}$$

Under a single policy network $J(\theta)$ both $q(s, a)$ and $q(s', a')$ are approximated using the same weights θ. This introduces over-approximations of future return since both estimates are coupled. As the network weights update, $q(s, a)$ approaches the target value while the approximation of $q(s', a')$ continues to change. A second DQN, the target network, adjusts a separate set of weights, $\bar{\theta}$, at time-step intervals to match those of the policy network, stabilising learning.

$$L_i(\theta_i) = E_{s,a\sim p(\cdot)}\left[(r_{t+1} + \gamma_{t+1} Q_{\bar{\theta}}(s_{t+1}, \underset{a'}{\operatorname{argmax}}\, Q_\theta(s_t + 1, a')) - Q_\theta(s_t, a_t))^2\right] \tag{20}$$

Following Q-value initialisation, agents have no a priori estimates regarding optimal state-action pairs. Acting greedily, $\max_{a'} Q_\pi(s, a)$, according to π likely results in sub-optimal solutions where insufficient experience is accumulated regarding $\mathcal{E}$. Conversely, sustained exploration, randomly sampling $a_t \sim A_t$, reduces the likelihood of an agent leveraging previous observations to maximise returns. An initial exploration rate $\epsilon \in [0, 1]$ is used to dictate how $p(s, a)$ is sampled for each episode. Agents act greedily with probability 1-ϵ or randomly with probability ϵ, reducing ϵ by a small decay factor at each episode.

To support the training of our policy network we generate a dataset D of capacity N experiences representing an agent's replay memory, a 5-tuple $e_t = (s_t, a_t, r_{t+1}, s_{t+1}, p_t)$, sampled at each time-step. Network updates are applied using minibatch samples from replay memory, $e_t \sim D$, with the aim of decorrelating consecutive experiences. This replicates i.i.d data traditionally used in supervised learning. Modifying experiences with an importance weight allows the network to prioritise minibatch samples with probability p_t according to the magnitude of their absolute TD-error; the difference between policy network and target network estimates of the Q-value for that time-step.

$$p_t \propto \left| r_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a') - Q_{\theta(s_t, a_t)} \right|^\omega \tag{21}$$

Machine learning means translating the model to another model - "...Which requires some partial translation.

labelled transition systems to MDPs

We implement our environment as the subset of all reachable states produced by executions of some arbitrary ladder logic program. Thus our action set comprises all propositional variables in V where a single action entails negating a single boolean.

2^{6453} in the case of one interlocking program

definition of reward function $R(s, a) = E\left[r_{t+1} | s_t = s, a_t = a\right]$

4 Implementation

[action space is one hot encoded binary vector of possible boolean valuations]

4.1 LL program generator

4.2 Learning Environment

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. To 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

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

Displayed equations are centered and set on a separate line.

$$x + y = z \tag{22}$$

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

Theorem 1. *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 [**?**,**?**,**?**], [**?**,**?**,**?**,**?**].

References

1. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)