Digging for Decision Trees: A Case Study in Strategy Sampling and Learning

Carlos E. Budde[1], Pedro R. D'Argenio[2,3], and Arnd Hartmanns[4]

[1] University of Trento, Trento, Italy
[2] Universidad Nacional de Córdoba, Córdoba, Argentina
[3] CONICET, Córdoba, Argentina
[4] University of Twente, Enschede, The Netherlands

Abstract. We introduce a formal model of transportation in an open-pit mine for the purpose of optimising the mine's operations. The model is a network of Markov automata (MA); the optimisation goal corresponds to maximising a time-bounded expected reward property. Today's model checking algorithms exacerbate the state space explosion problem by applying a discretisation approach to such properties on MA. We show that model checking is infeasible even for small mine instances. Instead, we propose statistical model checking with lightweight strategy sampling or table-based Q-learning over untimed strategies as an alternative to approach the optimisation task, using the Modest Toolset's modes tool. We add support for partial observability to modes so that strategies can be based on carefully selected model features, and we implement a connection from modes to the dtControl tool to convert sampled or learned strategies into decision trees. We experimentally evaluate the adequacy of our new tooling on the open-pit mine case study. Our experiments demonstrate the limitations of Q-learning, the impact of feature selection, and the usefulness of decision trees as an explainable representation.

1 Introduction

The model of Markov decision processes (MDPs) [8, 37] precisely captures the interplay of controllable or uncontrollable (nondeterministic) choices with randomness. It is simple yet versatile, which has made it popular in finance and operations research [16], in machine learning as the conceptual foundation of reinforcement learning [38], and in verification as the core formalism of probabilistic model checking (PMC) [6, 25]. MDPs are fully discrete: they transition between discrete states in discrete time; the number of choices per state is countable or finite; and the outcome of a choice is sampled from a discrete probability distribution. Modelling some applications, however, requires an explicit representation of continuous real time. Examples include performance evaluation scenarios such as computing the expected number of requests handled by a server per second, or determining the probability for the execution delay of a real-time task to exceed a certain bound. Models based on timed automata (TA) [3] offer a notion of "hard" real time with fixed deterministic delays or nondeterministic intervals. The tool Uppaal [7] prominently supports the analysis of

TA models via traditional and and statistical model checking (SMC) [12]. When events occur at random times with a known rate, a "soft" real-time model with stochastic, exponentially-distributed delays based on continuous-time Markov chains (CTMCs) is more appropriate.

In this paper, we present a novel case study about the optimisation of transport operations in an open-pit mine. It requires consideration of both controllable choices and stochastic time. The optimisation goal is to schedule trucks carrying material from shovels to dumps so that the expected total amount of material carried at the end of a shift is maximised. In abstract terms, this scenario is naturally modelled as a network of Markov automata (MA) [20,24], which combine the features of MDPs and CTMCs in a compositional manner. The optimisation goal can then be phrased as a query for the strategy maximising the value of a time-bounded expected accumulated reward property.

The analysis of MA is implemented by mcsta [14] and Storm [28] via PMC and by modes via SMC [11]. We use mcsta and modes in this paper; they are part of the Modest Toolset [23], available online at modestchecker.net. Whereas PMC [6] uses iterative numeric algorithms on an in-memory representation of the entire MA's state space (or of a subset sufficient for an ϵ-precise approximation [4]), SMC [1] performs a statistical evaluation of a large number of samples of the MA's behaviour. Efficient PMC algorithms exist to compute the values of unbounded properties by analysing the MA's embedded MDP, and for time-bounded reachability probabilities [13,15]. For time-bounded expected rewards, however, the only available PMC algorithm today uses a discretisation approach [27], which exacerbates the state space explosion problem of PMC by introducing many new states for the many discrete time steps needed for a reasonably precise result. We show in Sect. 6 that even checking the embedded MDP of small variants of our case study is infeasible; analysing our property of interest using a discretisation-based approach is thus out of the question.

Our contributions are (1) the introduction of the new open-pit mining case study (Sect. 2) and its MA model in the Modest modelling language [9,22] (Sect. 5); (2) an experimental evaluation (Sect. 6) of two approaches that extend SMC from estimation to optimisation (as required for MA) on several instances of varying sizes of the case study: smart lightweight strategy sampling (LSS) [18,34] and (explicit table-based) Q-learning [38,39], both implemented in modes; and (3) two extensions to Modest and modes to make LSS and Q-learning feasible and their outcomes explainable (Sect. 4). Our first extension is support for partial observations of states by designating a subset of the model's variables as observable. A good choice of observables reduces the sample space of strategies for LSS and the potential table size for Q-learning without removing much relevant information, making both methods more effective. Their outcomes are an opaque strategy identifier for LSS and a large table of strategy choices for Q-learning, both of which are hardly useful for mine operators. We thus created a new connection from modes to dtControl [5] to obtain a more explainable representation in the form of decision trees, implemented with a focus on minimal memory usage to be able to tackle large problems like the open-pit mining case.

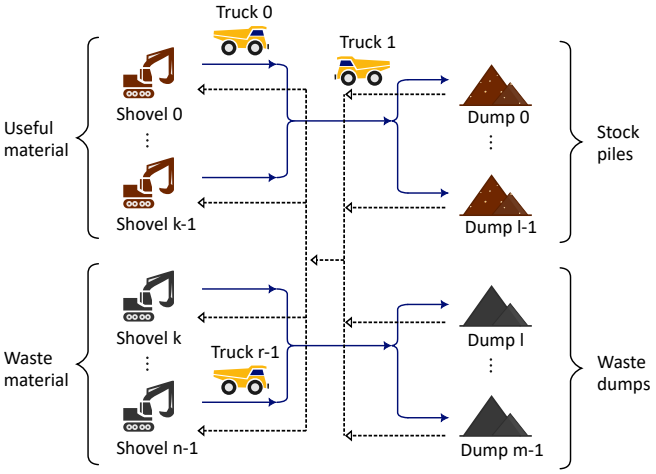


Fig. 1: Schematic view of the open pit mine

2 The Open-Pit Mining Case Study

Material transportation is one of the most important aspects that affects productivity in an open pit mine [2]. Material needs to be transported from the extraction points to different places depending on if the material contains ore or is simply waste. The material is loaded onto trucks by shovels that operate in a region of a single type of material. In general, there could be different type of ores in different regions, but we will consider only one type of ore here. The useful material containing ore is hauled by the trucks and dumped on stockpiles that later will be taken to crushers at the beginning of the ore processing line. The waste material is hauled and dumped in a separate designated place. Once a truck has dumped its load, it returns to *some* shovel and repeats the process.

Trucks may need to queue and wait at some loading or dumping place until other trucks are loaded or unloaded. Therefore, an appropriate distribution of the trucks that minimises their waiting time (and any other possible non-productive time) is crucial. The problem of assigning trucks to shovels and dumping places during production is called the *truck dispatching problem* [2, 35], which we address in this paper. Using the Modest Toolset, we propose a *flexible truck allocation* [35] approach to solve the problem. In such an approach, trucks are assigned a next dumping place or shovel by a dispatch system whenever they are done loading at a shovel or unloading at a dumping place, respectively.

Fig. 1 shows a schematic view of the mine operation in which k shovels load useful material and $N - k$ shovels load waste material. There are l dumping places associated to stockpiles and $M - l$ waste dumps. A truck loaded with useful material can only be assigned a stockpile to haul the load to, while a truck loaded with waste material can only be assigned a waste dump. Empty trucks that have just dumped their load can be assigned to any shovel. A truck

takes time to move from one point to the other. Similarly, loading and dumping are also activities that take time. Our objective is to model this scenario with MODEST and use the available tools of the Modest Toolset in order to maximise the productivity of the trucks. Concretely, our optimisation goal is to maximise the total load of material transported in one operation shift.

3 Background: Modelling and Analysis

In this section, we introduce all the existing concepts and technology necessary for our formal modelling and analysis of the open-pit mining case study.

Preliminaries. Given a set S, its powerset is 2^S. A probability distribution over S is a function $\mu\colon S \to [0,1]$ s.t. $spt(\mu) \stackrel{\text{def}}{=} \{\, s \in S \mid \mu(s) > 0 \,\}$ is countable and $\sum_{s \in spt(\mu)} \mu(s) = 1$. $Dist(S)$ is the set of all probability distributions over S.

3.1 Markov Automata

Markov automata combine MDPs and CTMCs in an orthogonal manner by providing two types of transitions: $s \xrightarrow{a} \mu$ as in MDP, and $s \overset{\lambda}{\dashrightarrow} s'$ as in CTMC. We now define Markov automata formally and describe their semantics.

Definition 1. *A* Markov automaton *(MA) is a tuple* $M = \langle S, s_0, A, P, Q, rr, br \rangle$ *where* S *is a finite set of* states *with* initial state $s_0 \in S$, A *is a finite set of* actions*,* $P\colon S \to 2^{A \times Dist(S)}$ *is the* probabilistic transition *function,* $Q\colon S \to 2^{\mathbb{Q} \times S}$ *is the* Markovian transition *function,* $rr\colon S \to [0,\infty)$ *is the* rate reward *function, and* $br\colon S \times Tr(M) \times S \to [0,\infty)$ *is the* branch reward *function.* $Tr(M) \stackrel{\text{def}}{=} \bigcup_{s \in S} P(s) \cup Q(s)$ *is the set of all* transitions*; it must be finite. We require that* $br(\langle s, tr, s' \rangle) \neq 0$ *implies* $tr \in P(s) \cup Q(s)$.

We also write $s \xrightarrow{a} \mu$ for $\langle a, \mu \rangle \in P(s)$ and $s \overset{\lambda}{\dashrightarrow} s'$ for $\langle \lambda, s' \rangle \in Q(s)$. In $s \overset{\lambda}{\dashrightarrow} s'$, we call λ the *rate* of the Markovian transition. We refer to every element of $spt(\mu)$ as a *branch* of $s \xrightarrow{a}_P \mu$; a Markovian transition has a single branch only. We define the *exit rate* of $s \in S$ as $E(s) = \sum_{\langle \lambda, s' \rangle \in Q(s)} \lambda$.

Intuitively, the semantics of an MA is that, in state s, (1) the probability to take Markovian transition $s \overset{\lambda}{\dashrightarrow} s'$ and move to state s' within t model time units is $\lambda / E(s) \cdot (1 - \mathrm{e}^{-E(s) \cdot t})$, i.e. the residence time in s follows the exponential distribution with rate $E(s)$ and the choice of transition is probabilistic, weighted by the rates; and (2) at any point in time, a probabilistic transition $s \xrightarrow{a} \mu$ can be taken with the successor state being chosen according to μ. We refer the interested reader to e.g. [27] for a complete formal definition of this semantics.

Example 1. We show example MA M_e without rewards in Fig. 2. Its initial state I has a choice between two probabilistic transitions with action labels `a` and `b`. The former leads to each of states S_1 and S_2 with probability 0.5. The dashed transitions are Markovian, labelled with their rates. In state B, there is a race between two Markovian transitions; the expected time spent in B is $\frac{1}{7}$ time units.

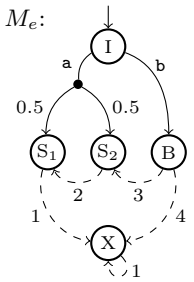


Fig. 2: Example MA M_e

```
action a, b;
int s;
alt {
:: a palt { :0.5: {= s = 2 =} :0.5: {= s = 1 =} }
:: b; alt {
   :: rate(3) {= s = 2 =}
   :: rate(4) {= s = 0 =}
   }
};
do {
:: when(s >= 1) rate(s) tau {= s-- =}
:: when(s == 0) rate(1) tau
}
```

Fig. 3: MA M_e in MODEST

An MA without Markovian transitions is an MDP; an MA without probabilistic transitions is a CTMC. The separation of transitions into probabilistic and Markovian enables parallel composition with action synchronisation without the need to prescribe an ad-hoc operation for combining rates as would be necessary for CTMC or continuous-time MDP (CTMDP) [37]. For verification, after applying the semantics of parallel composition, we make the usual *closed system* and *maximal progress* assumptions: probabilistic transitions face no further interference and take place without delay. The choice between multiple probabilistic transitions in a state remains nondeterministic, but all Markovian transitions can be removed from states that *also* have a probabilistic transition.

The behaviour of closed, deadlock-free MA M is characterised by the set Π of infinite timed paths in its semantics. Let Π_{fin} be the finite path prefixes.

Definition 2. *A* strategy *is a function* $\sigma\colon \Pi_{fin} \to Tr(M)$ *s.t.* $\forall s \in S\colon \sigma(s) = tr$ *implies* $tr \in P(s) \cup Q(s)$*. A* time-dependent *strategy is in* $S \times [0,\infty) \to Tr(M)$*; a* memoryless *strategy is in* $S \to Tr(M)$*.*

If we "apply" a strategy to an MA, it removes all nondeterminism, and we are left with a stochastic process whose paths can be measured and assigned probabilities according to the rates and distributions in the (remaining) MA. We again refer the interested reader to e.g. [27] for a fully formal definition.

Given an MA model, we are interested in determining the maximum (supremum) or minimum (infimum) value over all strategies of the following properties:

Expected accumulated reachability rewards: Compute the expected value of the random variable that assigns to π the sum of its branch rewards and of each state's rate reward multiplied with the residence time in that state, up to the first state in goal set $G \subseteq S$. This is well-defined if the maximum (minimum) probability to reach G is 1; otherwise, we define the minimum (maximum) expected value to be ∞. Memoryless strategies suffice to achieve optimal results (i.e. the maximum and minimum expected values).

Time-bounded expected accumulated rewards: Compute the expectation as above, but instead of stopping at goal states, stop once a given amount of time $\mathcal{T} \in [0,\infty)$ has elapsed. Time-dependent strategies suffice.

Expected reachability rewards can be computed via standard MDP model checking by considering the MA's embedded MDP, i.e. replacing all remaining Markovian transitions out of a state by a single probabilistic transition, using the rates as weights to determine the branch probabilities. For time-bounded expected rewards, the best choice among probabilistic transitions may depend on the time remaining to $\mathcal{T}$—thus the need for time-dependent strategies. The only available model checking algorithm for this type of property discretises the remaining time [27], exacerbating the state space explosion problem. We therefore propose to use SMC instead, which however cannot directly deal with the optimisation problem induced by the nondeterministic choices. We describe the two methods to deal with finding (near-)optimal strategies in SMC in Sect. 3.3 below.

3.2 Modest for Markov Automata

MODEST [9, 22] is the modelling and description language for stochastic timed systems. It provides process-algebraic operations such as parallel and sequential composition, parameterised process definitions, process calls, and guards to construct complex models from small reusable parts. Its syntax leans on commonly used programming languages, and it provides conveniences such as loops and an exception handling mechanism. To specify complex behaviour in a succinct manner, MODEST provides variables of standard basic types (e.g. `bool`, `int`, or bounded `int`), arrays, and user-defined recursive datatypes akin to functional programming languages. We introduce MODEST for modelling MA by example:

Example 2. Fig. 3 shows a MODEST model whose concrete semantics is our example MA M_e. Choices between multiple transitions are implemented with the `alt` construct. Probabilistic transitions can be labelled with user-defined action names like `a` and `b`, while Markovian transitions—for which a `rate` is given—must use the predefined non-synchronising action `tau`. The `palt` construct implements the probability distributions of probabilistic transitions; `;` is the sequential composition operator. Assignments `s = 2` are given in assignment blocks `{= ... =}`; if a block has multiple assignments, they are executed atomically.

3.3 Analysis of Markov Automata via Statistical Model Checking

Monte Carlo methods such as *statistical model checking* (SMC) [1], which is in essence discrete-event simulation [33] for formal models and properties, can estimate expected rewards in CTMCs. To do so, the SMC algorithm (pseudo-) randomly samples n finite paths—*simulation runs*—through the CTMC, collects each path's accumulated reward value, and returns the average of the collected values. The result is correct up to a statistical error and confidence depending on n. We assume that we can effectively perform simulation runs on a high-level description of the MA (e.g. in MODEST). Then, in contrast to PMC, SMC does not need to store the CTMC's entire state space and thus runs in constant memory. SMC is easy to parallelise and distribute on multi-core systems and compute clusters.

Input: MA M, iteration budget K, and strategy budget N with $N \leq K$.
Output: The maximising strategy $\sigma_{\max}$.
1 $\Sigma := \{\, \sigma_i \mid \sigma_i = \text{sample uniformly from } \mathbb{Z}_{32}, 0 \leq i \leq N \,\}$
2 **while** $|\Sigma| > 1$ **do**
3 **foreach** $\sigma \in \Sigma$ **do** $\hat{R}_\sigma :=$ average of $\lceil K/|\Sigma| \rceil$ simulation run values for σ
4 $\Sigma := \{\, \sigma \in \Sigma \mid \hat{R}_\sigma$ is among the $\lceil |\Sigma|/2 \rceil$ highest values in $\{\, \hat{R}'_\sigma \mid \sigma' \in \Sigma \,\} \,\}$
5 $\sigma_{\max} :=$ the only remaining strategy identifier in Σ

Algorithm 1. Lightweight strategy sampling with the smart sampling heuristics

The simulation of nondeterministic models like MDP or MA, however, requires a strategy to resolve the nondeterminism during simulation. Ideally, such a strategy should be the optimal one, i.e. one that results in the maximum or minimum expected reward value. Obtaining optimal strategies experimentally is infeasible in any sensible model. Instead, best-effort methods to find near-optimal strategies have been devised, notably based on *strategy sampling* and *learning*.

Lightweight strategy sampling (LSS) was devised for MDP [34]. On MDP M,

(i) it randomly selects a set Σ of N strategies, each identified by a fixed-size integer (e.g. of 32 bits as in our implementation),
(ii) employs a heuristic (that involves simulating the DTMCs $M|_\sigma$ resulting from applying a strategy $\sigma \in \Sigma$ to M) to select the $\sigma_{opt} \in \Sigma$, $opt \in \{\, \max, \min \,\}$, that appears to induce the highest/lowest probability, and finally
(iii) performs standard SMC on $M|_{\sigma_{opt}}$ to provide an estimate $\hat{R}_{\sigma_{opt}}$ for the optimal expected accumulated reward.

Unless Σ happens to include an optimal strategy and the heuristic identifies it as such, $\hat{R}_{\sigma_{max}}$ is only an underapproximation (overapproximation) of the maximum (minimum) expected reward, and subject to the statistical error of the final SMC step. The effectiveness of LSS depends on the probability mass of the set of near-optimal strategies among the set of all strategies that we sample Σ from: It works well if a randomly selected strategy is somewhat likely to be near-optimal, but usually fails in cases where many decisions need to be made in exactly one right way in order to get any non-negligible reward at all.

Smart sampling. We use an MA adaptation of LSS with the *smart sampling* heuristics [18] for step (ii). It is schematically presented in Alg. 1 for $opt = \max$. It receives as inputs the MA, the strategy budget N, and the iteration budget K. N determines how many strategies will be randomly selected while K is the number of simulation runs to be performed in each iteration. We require $N \leq K$ so that in the first round, we have at least one simulation per strategy. After sampling the strategies (line 1), the algorithm runs $\lceil K/N \rceil$ simulations for each strategy estimating accumulated rewards (line 3). Afterwards, it discards the worst half of the strategies (line 4) and simulates the remaining ones with twice

Input: MA M, time bound $\mathcal{T}$, strategy identifier $\sigma \in \mathbb{Z}_{32}$, hash function $\mathcal{H}$.
Output: The reward r accumulated along the sampled run.

1 $s := s_0$, $t := 0$, $r := 0$ *// initialise current state, time, and reward*
2 **while** $t \leq \mathcal{T}$ **do** *// run until time bound is reached*
3 **if** $P(s) = \emptyset$ **then** *// s has only Markovian transitions*
4 $\langle t', \langle \lambda, s' \rangle \rangle :=$ sample sojourn time and transition from $Q(s)$
5 $r := r + \min(t', \mathcal{T} - t) \cdot rr(s) + br(\langle s, \langle \lambda, s' \rangle, s' \rangle)$ *// collect the reward*
6 $t := t + t'$ *// increase current time*
7 **else** *// s is a probabilistic state*
8 $\langle a, \mu \rangle := (\mathcal{H}(\sigma.s) \bmod |P(s)| + 1)$-th element of $P(s)$ *// select transition*
9 $s' :=$ sample the next state according to μ
10 $r := r + br(\langle s, \langle a, \mu \rangle, s' \rangle)$ *// collect the reward*
11 $s := s'$ *// set new current state*

Algorithm 2. A single simulation run using a sampled strategy σ in LSS

the number of simulation runs per strategy (line 3 again). The loop repeats until only one strategy remains, which is σ_{max}. In this way, the number of simulation runs, and thus the runtime, grows only logarithmically in N.

Lightweight strategies. The key to LSS is the constant-memory representation of strategies as (32-bit) integers. It enables the algorithm to run in constant memory, which sets it apart from simulation-based machine learning techniques such as reinforcement learning, which need to store learned information (e.g. Q-tables, see below) for each visited state.

Alg. 2, the simulation called in line 3 of Alg. 1, shows how this is done. Apart from the MA M and the time-bounded reward property's stopping time $\mathcal{T}$, its inputs include the strategy identifier $\sigma \in \mathbb{Z}_{32}$ and a (usually simple non-cryptographic) uniform deterministic hash function $\mathcal{H}$ that maps to values in $\mathbb{Z}_{32}$. The algorithm is mostly self-explanatory. Lines 4-6 take care of the selection of the Markovian step, the time advance, and the update of the reward for the Markovian case. The notable part lies in the case of a choice between $k > 1$ enabled probabilistic transitions (line 7). Assuming some total order on the transitions, the $(\mathcal{H}(\sigma.s) \bmod k)$-th transition is selected, where $\sigma.s$ is the concatenation of the binary representations of σ and s (line 8). This selection procedure is deterministic, so we can reproduce the decision for state s at any time knowing σ. For nontrivial $\mathcal{H}$, it is also highly unpredictable: changing a single bit in σ may result in a different decision for many states. The selected transition is then used to sample the next state and the update of the reward (lines 9 and 10). Finally, after updating the accumulated reward and the current state, the loop continues until the time limit $\mathcal{T}$ is reached.

Observe that Alg. 2 implements a memoryless strategy, whereas time-dependent strategies would be needed to obtain optimal results for time-bounded reward properties. This is a practical simplification to make LSS work effectively: If we used time-dependent strategies directly by e.g. feeding a floating-point

Input: MA M, time bound $\mathcal{T}$, number of episodes N, family of learning rates $\{\alpha_i\}_{i=1}^N$, family of exploration likelihoods $\{\epsilon_i\}_{i=1}^N$, discount factor γ.
Requires: s_0 is a state with probabilistic transitions.
Output: Table $\mathcal{Q}$ assigning expected discounted reward to state-action pairs.

1 **for** $i := 1$ **to** N **do**
2 $\quad s := s_0$, $t := 0$ *// initialise current state and time*
3 $\quad$ **while** $t \leq \mathcal{T}$ **do** *// run until time bound is reached*
4 $\quad\quad \langle a, \mu \rangle :=$ sample uniformly from $P(s)$ *// s has probabilistic transitions*
$\quad\quad\quad \oplus_{\epsilon_i} \operatorname{arg\,max}_{\langle a', \mu' \rangle \in P(s)} \mathcal{Q}(s, a')$ *// $\oplus_{\epsilon_i}$: random choice with probability ϵ_i*
5 $\quad\quad s' :=$ sample the next state according to μ
6 $\quad\quad r := br(\langle s, \langle a, \mu \rangle, s' \rangle)$ *// collect the reward*
7 $\quad\quad s := s'$, $s'' := s'$ *// set new current state*
8 $\quad\quad$ **while** $t \leq \mathcal{T} \wedge P(s') = \emptyset$ **do** *// while s' has only Markovian transitions:*
9 $\quad\quad\quad \langle t', \langle \lambda, s'' \rangle \rangle :=$ sample sojourn time and transition from $Q(s')$
10 $\quad\quad\quad r := r + \min(t', \mathcal{T} - t) \cdot rr(s') + br(\langle s', \langle \lambda, s'' \rangle, s'' \rangle)$ *// collect reward*
11 $\quad\quad\quad s' := s''$, $t := t + t'$ *// set new state and increase time*
12 $\quad\quad \mathcal{Q}(s, a) := (1 - \alpha_i) \cdot \mathcal{Q}(s, a) + \alpha_i \cdot \left(r + \gamma \cdot \max_{\langle a'', \mu'' \rangle \in P(s'')} \mathcal{Q}(s'', a'')\right)$
13 $\quad\quad s := s''$ *// set new current state*

Algorithm 3. Q-learning algorithm for MA

representation of $\mathcal{T}$ into $\mathcal{H}$ as well, then all strategies would break down to behaving like the uniformly random strategy [17]; if we used discretisation like in the model checking algorithm, the space of strategies would blow up so much that the probability of sampling a useful strategy would be negligible.

Q-learning (QL) [39] is a popular method for reinforcement learning (RL) [38], a machine learning approach to train agents to take actions maximising a reward in uncertain environments. Mathematically, the agent in its environment can be described as e.g. a MA[5]: the agent chooses actions while the environment determines the probabilistic outcomes of the actions in terms of successor states. QL maintains a Q-function $\mathcal{Q}\colon S \times A \to [0, 1]$ stored in explicit form (as a so-called *Q-table*) initialised to 0 everywhere. For MA, the Q-table only needs to store values for states with probabilistic transitions.

Algorithm. Using a family of *learning rates* $\{\alpha_i\}_{i=1}^N$, a family of *exploration likelihoods* $\{\epsilon_i\}_{i=1}^N$, and a discount factor $\gamma \in (0, 1]$, N learning *episodes* are performed following Alg. 3. For each episode i starting from s being the MA's initial state s_0 assumed w.l.o.g. to be probabilistic, the algorithm selects with probability ϵ_i whether to *explore* possibly new decisions by sampling uniformly from the set $P(s)$ or to *exploit* what has been learned so far in the Q-table (line 4). This is called an *ϵ-greedy strategy*. Afterwards, the next state is selected randomly

[5] To ease the presentation, we assume actions to uniquely identify transitions per state.

according to the transition's probability distribution and the reward is collected (lines 5 and 6). Since the Q-table is only defined on states with probabilistic transitions, all rewards of states with only Markovian transitions that follow up to the next probabilistic one are accumulated into the Q-table entry of the preceding probabilistic state (lines 8-11). After collecting the rewards, the Q-table is updated (line 12). Here, the learning factor α_i determines the impact of new information on the existing knowledge. Typically, α_i and ϵ_i decrease as i—the number of episodes run so far—increases. A higher ϵ_i allows the algorithm to explore various actions, avoiding premature convergence to sub-optimal solutions. As the algorithm learns more about the environment, it becomes beneficial to gradually reduce ϵ_i, leading to a focus on exploiting the best-known strategies. Similarly, a high initial α_i allows for rapid learning but can destabilise due to noisy data or outliers. Reducing α_i over time helps stabilise the learning process as the algorithm converges, integrating new information more conservatively [39]. As an optimisation, we skip Q-table updates for states with only one transition.

Discounting and convergence. An episode is very similar to a simulation run. The main differences to simulation as used for LSS are that we update the Q-table to estimate the "quality" $\mathcal{Q}(s, a)$ of taking the action a from state s and follow an "ϵ_i-greedy" strategy. RL traditionally optimises for expected discounted rewards, thus the discount factor γ. Since our objective is undiscounted, we set γ to 1. Standard results [38,39] guarantee that the algorithm converges to the maximal expected accumulated reward as $N \to \infty$, as long as (i) every state is guaranteed to be visited infinitely often (i.e. $\epsilon_i > 0$ for all $i \geq 0$), (ii) parameters α_i and ϵ_i decrease as $i \to \infty$, and (iii) the families $\{\alpha_i\}_{i \geq 1}$ and $\{\epsilon_i\}_{i \geq 1}$ fulfill some variant of the stochastic approximation conditions.

Performance and scalability. While QL is similar to LSS in that it uses simulation runs—so both are so-called model-free techniques—its memory usage is in $\mathcal{O}(|S| \cdot |A|)$ (for the Q-table) and thus more similar to probabilistic model checking. For many models, however, QL only explores—and thus stores a Q-value for—a subset of S. This happens because some parts of the state space have a very low probability of being reached from s_0 within the specified number of episodes. Additionally, no Q-values need to be stored for states that have Markovian transitions only or just a single probabilistic transition. The time spent in LSS and QL depends on the number of simulation runs performed. For LSS using our smart sampling approach, we need $\mathcal{O}(N \cdot \log K)$ runs (where K is the strategy budget and N is the simulation budget per iteration), while QL needs $\mathcal{O}(N)$ runs (where N is the number of episodes to learn from). Each run (episode) in QL is however slightly more computationally expensive than in LSS due to the computations involving the Q-table.

4 Tool Extensions

To tackle our new open-pit mining case study, we extended the modes statistical model checker [11] of the Modest Toolset [23] with three crucial new features

that we describe in this section: Syntax and LSS- as well as QL-support for partial obervability (Sect. 4.1), a constant-memory implementation of strategy extraction from SMC (Sect. 4.2), and a connection to the dtControl tool to obtain a decision tree representation of strategies (Sect. 4.3). We also implemented LSS and QL for MA and memoryless strategies as described in the previous section, as QL was previously only supported for MDP.

4.1 Partial Observability

In practical machine learning, it is common practice to expose to the learner not the full state of a model, with values for all of the model's variables, but only a set of carefully selected *features*. The learner then effectively works on a smaller state space. We can cast feature selection as a variant of *partial observability*, though not with the intention of finding the optimal strategy for the actual partially-observable MDPs [30] or MAs: such strategies require tracking probability distributions about which actual states the model could be in based on the history of observations, resulting in complex strategies and complicating any possible LSS or QL approach.

In the feature-oriented approach, we simply replace states by observables in the LSS and QL algorithms. Let $\omega\colon S \to \mathit{Obs}$ map states to some finite set of observables Obs. Then, in Alg. 2 for LSS, we replace line 8 by

$$\langle a, \mu\rangle := (\mathcal{H}(\sigma.\omega(s)) \bmod |P(s)|)\text{-th element of } P(s);$$

in Alg. 3 for QL, we replace line 5 by $\oplus_{\epsilon_i} \arg\max_{\langle a',\mu'\rangle \in P(s)} \mathcal{Q}(\omega(s), a')$ and line 12 by

$$\mathcal{Q}(\omega(s), a) := (1-\alpha_i)\cdot \mathcal{Q}(\omega(s), a) + \alpha_i \cdot \left(r + \gamma \cdot \max_{\langle a',\mu'\rangle \in P(s')} \mathcal{Q}(\omega(s'), a')\right).$$

We require that ω projects only states with the same actions to the same observable, i.e.

$$\forall\, o \in \mathit{Obs}\ \forall\, s_1, s_2 \in \omega^{-1}(o)\colon (\exists\, \mu_1\colon s_1 \xrightarrow{a} \mu_1) \Leftrightarrow (\exists\, \mu_2\colon s_2 \xrightarrow{a} \mu_2). \tag{1}$$

Otherwise, the Q-table would become inconsistent; for LSS, the inconsistencies would become apparent during strategy extraction (see Sect. 4.2 below).

To allow the modeller to describe Obs and ω, we extended the Modest language by the ability to mark a subset of the model's variables as `observable`. Then Obs is the set of states projected to observable variables only, and ω discards the values of all non-observable values in a state. This is inspired by the way the Prism language supports partially-observable models [36].

4.2 Strategies from SMC

When the LSS process with smart sampling of Alg. 1 completes, it returns a strategy in the form of a strategy identifier $\sigma_{\max} \in \mathbb{Z}_{32}$. When QL as in Alg. 3 is finished, it returns a strategy in the form of the Q-table $\mathcal{Q}$. Both representations are not very useful: $\sigma_{\max}$ contains no explicit information about its expected reward or the actual decisions the strategy makes to attain it, while $\mathcal{Q}$ is large,

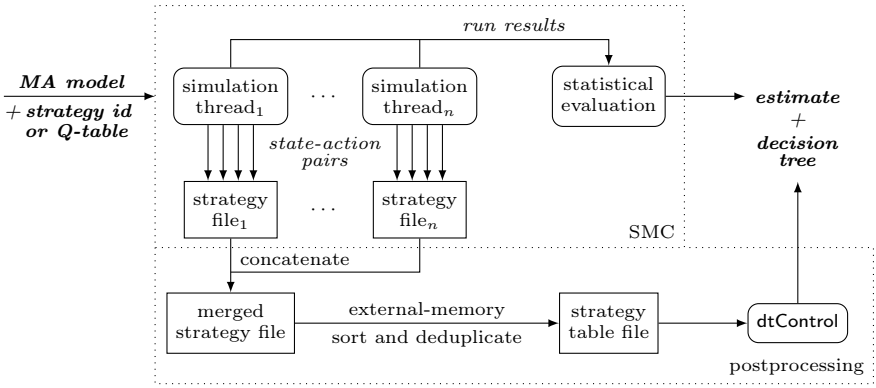


Fig. 4: Schematic overview of strategy extraction from SMC

contains the decisions only implicitly, and $\max_{\langle a,\mu\rangle \in P(s_0)} \mathcal{Q}(s_0, a)$ is an approximation of the expected reward with no guarantee as to how far from the strategy's real expected reward it is. To obtain an estimate of the strategy's expected reward with a statistical error guarantee, modes therefore performs a separate SMC analysis of the MA under the strategy. That is, it performs a number of new simulation runs that is sufficient to attain the error and confidence requested by the user, and returns their average value as the estimate of the expected reward.

For the open-pit mining case study, however, we are also interested in understanding the corresponding strategy: How does the mine operator need to schedule its trucks in order to achieve the computed (hopefully near-maximal) total load of material transported? To answer this question, we have extended modes with a novel method to extract (an approximation of) the strategy in a more explicit form that runs in constant memory, just like SMC and LSS. Fig. 4 shows a schematic overview of the implementation: Each simulation thread (in a multi-core or distributed setting) writes all state-action pairs chosen by the strategy that it encounters during the simulation runs it performs into a separate binary file on disk. Thus there is no overhead for coordinating the threads while they run. Only after the SMC process is done and the estimate has been returned do we process these files: We first concatenate them into a single binary "merged" strategy file. Then we apply an external-memory merge sort algorithm to sort the state-action pairs in this file according to some arbitrary order for the purpose of eliminating any duplicates. At this point, we detect if an LSS strategy under partial observability violates Eq. 1. If this is the case, the error is reported and the process is aborted. Otherwise, the resulting table of action choices for unique states is written to a text file in mcsta's strategy file format.

4.3 Decision Trees

The tabular strategy file that our new SMC strategy extraction method in modes delivers is human-readable and arguably more useful that the integer strategy

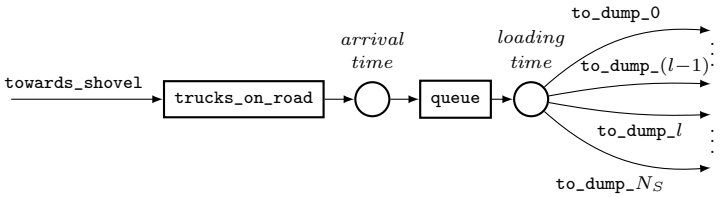


Fig. 5: Schematic view of the behaviour of trucks in a `ShovelSystem`

identifier or the huge Q-table. It may still be very large, however, hiding interesting patterns such as dependencies between ranges of state variable values and strategy choices. To obtain a more explainable representation, we implemented a new connection from modes and mcsta to the dtControl tool. dtControl [5] reads tabular strategy representations produced by model checkers such as Prism [32] and Storm [28] or by Uppaal Stratego [19] and learns a decision tree that succinctly represents the strategy. We implemented support for the textual strategy file format now used by modes and mcsta via a new dataset loader for dtControl. By supporting both the SMC and the PMC tool of the Modest Toolset, we can also compare—on small models—the trees dtControl learns for the complete strategy obtained by mcsta vs. the one for the usually incomplete table (that only contains the states actually visited during SMC) generated by modes. We show decision trees obtained via this new connection for the open-pit mining case study in Sect. 6.

5 Case Study Modelling

To improve the tractability of the truck dispatching problem, we make some simplifications on the open-pit mine model. First, we assume all timing aspects stochastically distributed under exponential distributions, enabling modelling as MA. Though different truck models can haul significantly different loads in practice, we assume they are all equal. Also, distances, which determine the travelling times, are point to point. However, to reduce the number of combinations, we consider that the distance towards one point—a shovel or a dump—is the same from any other point (but may differ from the distance towards a different point).

Modest model. The schematic of all activities that involve a truck—from starting to go to the shovel until it is finally served and assigned to a dump—is depicted in Fig. 5. This is what the Modest model of the shovel system in Fig. 6 aims to capture. The process `ShovelSystem` receives two parameters: `shovel_id` identifies this shovel and determines if it loads useful or waste material while `t_time` is the average time that a truck needs to travel to this shovel. Variables are introduced in lines 2-5. Variable `trucks_on_road` counts the trucks that are approaching this shovel and variable `queue` counts the trucks that have arrived and are queuing for the shovel. The observable Boolean `full` indicates when a truck

```
1  process ShovelSystem(int(0..N_S) shovel_id, int(0..MAX_TIME) t_time) {
2     int(0..NR_TRUCKS) queue = 0;
3     int(0..NR_TRUCKS) trucks_on_road = 0; // trucks travelling to this shovel
4     observable int(0..MAX_OBS) stress = 0; // shovel stress level
5     observable bool full = false; // if true, a truck was just loaded
6     do {
7     :: towards_shovel // a truck is sent towards this shovel
8        {= trucks_on_road++, stress = min(trucks_on_road+queue, MAX_OBS) =}
9     :: when(trucks_on_road > 0) // a truck arrives to this shovel
10       rate(trucks_on_road / t_time) {= queue++, trucks_on_road-- =}
11    :: when (queue > 0) rate(1 / LOAD_TIME) // a truck is being loaded
12       {= queue--, stress = min(trucks_on_road + queue, MAX_OBS), full = true =}
13       // The truck is sent to a dump of the right kind for this shovel
14    :: when(full && shovel_id <  k) to_dump_0          {= full = false =}
15       ...
16    :: when(full && shovel_id <  k) to_dump_(l − 1)    {= full = false =}
17    :: when(full && shovel_id >= k) to_dump_l          {= full = false =}
18       ...
19    :: when(full && shovel_id >= k) to_dump_(N_D − 1) {= full = false =}
20    }
21 }
22 process DumpSystem(int(0..N_D) dump_id, int(0..MAX_TIME) h_time) {
23    int(0..NR_TRUCKS) queue = 0;
24    int(0..NR_TRUCKS) trucks_on_road = 0; // trucks travelling to this dump
25    observable int(0..MAX_OBS) stress = 0; // dump stress level
26    observable bool empty = false; // if true, a truck has just unloaded
27    do {
28    :: towards_dump // a truck is sent towards this dump
29       {= trucks_on_road++, stress = min(trucks_on_road+queue,MAX_OBS) =}
30    :: when (trucks_on_road > 0) // a truck arrives to this dump
31       rate(trucks_on_road / h_time) {= queue++, trucks_on_road-- =}
32    :: when (queue > 0) rate(1 / UNLOAD_TIME) // a truck is dumping the material
33       {= queue--, stress = min(trucks_on_road + queue, MAX_OBS),
34          load = TRK_LOAD, empty = true =}
35       // The truck is sent to a shovel
36    :: when (empty) to_shovel_0         {= empty = false =}
37       ...
38    :: when (empty) to_shovel_(N_S − 1) {= empty = false =}
39    }
40 }
```

Fig. 6: MODEST models for the shovel and dump systems

is full and should be dispatched to some dump. Observable variable `stress` do not play a role in the control flow; we explain it later.

The whole behaviour is captured in the loop in lines 6-20. When a truck is dispatched to the shovel, it synchronises with action `towards_shovel` (line 7) which in turn increases the counter of trucks travelling towards this shovel. If there are trucks travelling to the shovel (line 9), then one of them may arrive. The time of arrival of a truck is determined by an exponential distribution of average `t_time` that is weighted by the total number of trucks approaching the shovel (line 10). On arrival, the truck enqueues and the number of approaching trucks decreases by one. If some truck is waiting for loading then there is certainly one being loaded and the time of loading is determined by an exponential distribution of average `LOAD_TIME` (line 12). When the truck finishes loading, it is removed

from the queue and variable `full` is set to indicate that the truck needs to be dispatched. This happens in lines 14-19. In particular, if `shovel_id` is smaller than a given number k, this shovel loads useful material and the truck can only be dispatched to one of the l dumps with stockpiles. If instead `shovel_id` is greater or equal to k, it loads waste and hence the truck should be dispatched to one of the $N_D - l$ waste dumps. The dispatching is carried out with one of the `to_dump_`i actions which in turns synchronises with the `towards_dump` action of the dump system with identification number i (in process `DumpSystem`). Action `towards_dump` in the dump is the analogon to `towards_shovel` in the shovel system. A dump system is modelled similarly, the only difference being that trucks can be dispatched to any shovel regardless of the material they handle.

To obtain a reduced observable domain, only variables `full` and `stress` are made observable. Variable `full` needs to be observable in order to distinguish the states in which the dispatching actions are enabled to ensure Eq. 1 holds. Variable `stress` is specifically designed to have a reduced observable domain and can be understood as the shovel stress level. It contains the total number of trucks that are either approaching to or waiting in the shovel but up to the maximum `MAX_OBS` which indicates the maximum stress. This constant is different in each shovel; we made it depend on the arrival time and the loading time. A similar decision has been taken for the dump systems.

Apart from the shovel systems and the dump systems, the model is completed with an initialisation process. Since the mine is not assumed to be in any particular initial state, the initialisation module dispatches the trucks nondeterministically to any of the sites.

Optimisation objective. The objective is to maximise productivity, i.e. the expected total load of material moved from the shovels to the dumps during a shift. Hence, we obtain a branch reward of the capacity of the truck each time a truck finishes dumping its load. This is indicated with the assignment `load = TRK_LOAD` in line 34 of the `DumpSystem` (see Fig. 6) to the *transient* global variable `load`. A transient variable is not part of the state and only takes a value other than its default (0 for `load`) during the execution of the assignment block. In Modest notation, we analyse the property

```
Xmax[T == SHIFT](S(load))
```

which represents the maximum expected value of the sum of `load` values along the execution before reaching time `SHIFT`, which we set to 200.

Though not particularly interesting for the truck dispatching problem, we also estimate the *minimum* expected accumulated value of `load` as well as the value obtained by the uniform random strategy (that randomly chooses a transition every time it encounters a state with multiple probabilistic transitions). This is useful to gain insights into the ability of the LSS and QL engines to actually optimise (measured by the spread between maximum and minimum value) and find nontrivial strategies (that differ from the uniform random one). In addition, as the result of the integration of modes with dtControl, we are able to derive decision trees that explain the near-optimal strategies that we find.

Table 1: Experimental setup

# trucks:	4	5	9	10	35	40	80
# shovels:	6	1	3	6	6	8	10
# dumps:	5	2	2	5	5	8	10
# ore shovels:	3	0	1	3	3	4	5
# ore dumps:	3	0	1	3	3	4	5
combinations shovel → dump:	15	2	3	15	15	32	50
combinations shovel ← dump:	30	2	6	30	30	64	100
comb. ore shovel → ore dump:	9	0	1	9	9	16	25

(a) Mine model instances—name is # trucks

Run name	Parameters
LSS 10k 1k	10k simulation traces, 1k strategies
LSS 100k 10k	100k simulation traces, 10k strategies
Qlearn 100k	100k episo., 0.5 λ, 0.02 fin-λ, 1.0 ε, 0.02 fin-ε
Qlearn 3M	3M episo., 0.6 λ, 0.01 fin-λ, 0.6 ε, 0.02 fin-ε

(b) modes executions

6 Experimental Results

Setup. We use the LSS and QL implementations of modes to estimate the maximum and minimum expected accumulated load on seven instances of our model from Sect. 5—see Table 1a for an overview of their configurations. For each instance, besides the uniform run, we execute modes with the algorithms from Table 1b and let it build confidence intervals of 1% relative half-width in the final SMC analysis. For each model, property, and run configuration, we execute modes in two modes: the default with fully-observable states (F), which considers all model variables observable, and with partially-observable states (P) following the `observable` annotations in the model. Each experiment is run until a confidence interval of the requested width is built for the property under study. We use the center of the intervals to compare the minimum and maximum loads found by each run on each model, keeping track of the wall-clock runtime. We execute our experiments on an AMD Ryzen 9 7950X3D system (16 cores) with 128 GB of RAM running 64-bit Ubuntu Linux 22.04.

Results for loads. Figs. 7 and 8 show the results obtained in overview and in detail, respectively. In Fig. 7, for each model and execution configuration, we plot a vertical rectangle: its lower bound is the minimum estimated by the configuration, and its upper bound is the maximum. Therefore, the taller the rectangle, the bigger the difference between minimum and maximum. The width of a rectangle bears no meaning, but its colour does: P runs are blue-greens and F runs are red-oranges; the darker the hue, the longer the runtime[6]—but see Fig. 8 for an objective runtime comparison.

Fig. 7 does not include results for Q-learning with the heavier budget (3M episodes), because these runs failed for all large models as shown in more detail in Fig. 8. Furthermore, in three cases of Q-learning with 100k episodes the minimum load achieved was higher than the maximum load: this is indicated in Fig. 7 with empty rectangles whose contour is drawn with a dashed line (P model 80, and F models 9 and 35). This inversion never occurred when using LSS. Finally, the transversal horizontal gray bars are the results of the uniform runs, which are plotted to serve as reference point.

[6] We use the average runtime of both properties per model and run, whose coefficient of variation was in almost all cases below 10% and in two cases 25.3% and 25.1%.

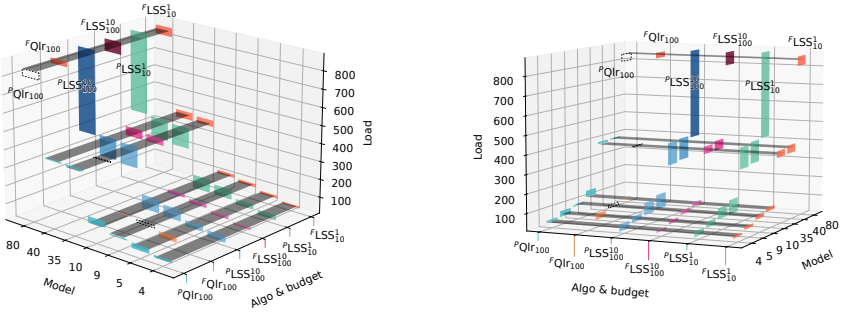


Fig. 7: Overview of experimental results: minimum and maximum loads.

Fig. 7 shows how the uniform values are closer to the maximum load found by other runs than to the minimum loads. Besides, all runs with the uniform random strategy took less than 2–3 seconds to finish. However, in all cases for LSS, there was no overlap between the final confidence interval produced for the maximum load and that produced by the uniform random strategy. Thus LSS shows a statistically significant improvement for minimum and maximum loads over the uniform strategy here, even in the lighter LSS configuration.

Fig. 8 allows for a better comparison, where we see that the runtimes do not vary significantly between F and P runs, but instead what we gain by using partially-observable states are better minimising strategies. Fig. 8 also makes it clear that increasing the simulation budget of LSS from 10k 1k to 100k 10k impacts runtime severely, but has only a minor effect in terms of finding better strategies here. Q-learning runs achieved no significantly different results than the uniform strategy, but incurred higher runtimes (except vs. LSS 100k 10k) and failed for all large models due to running out of memory for 3M episodes.

Results for strategies. We also study the strategies—and corresponding decision trees—that modes synthesised to achieve the loads shown in figs. 7 and 8. For each model, partial- or fully-observable state, and minimum- or maximum-load objective, Fig. 9 compares the number of choices in a strategy vs. the number of nodes of the decision tree built for it. We observe that decision trees achieve a slight compression (i.e. they contain fewer nodes than there are choices of the corresponding strategy) only in the case of F models. This was expected, since fully-observable models contain variables that may be of no use to (minimise or) maximise the load, so removing them from the picture increases the entropy—we provide a concrete example in Fig. 10b.

Moreover, the strategies found by Q-learning contain much fewer choices than those of LSS (but note that for models $\geqslant 35$ with F states, and $\geqslant 40$ with P states, Q-learning failed to produce any strategy). In several cases when computing minimum loads, Q-learning failed to learn non-zero values for any relevant states, resulting in empty strategies (since the choices had to be made randomly for lack of information). As already discussed, the minimum and maximum loads

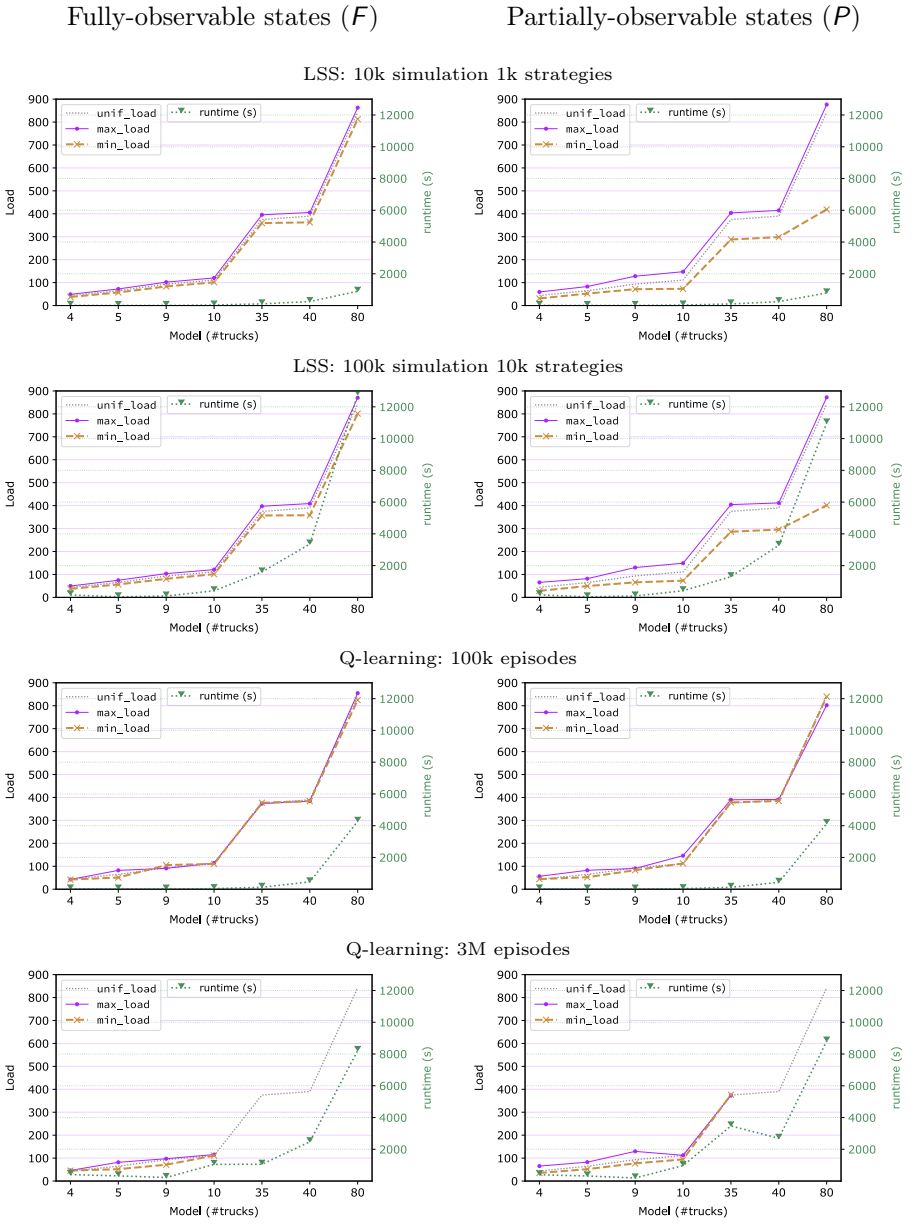


Fig. 8: Min. and max. expected loads via different algorithms and budgets

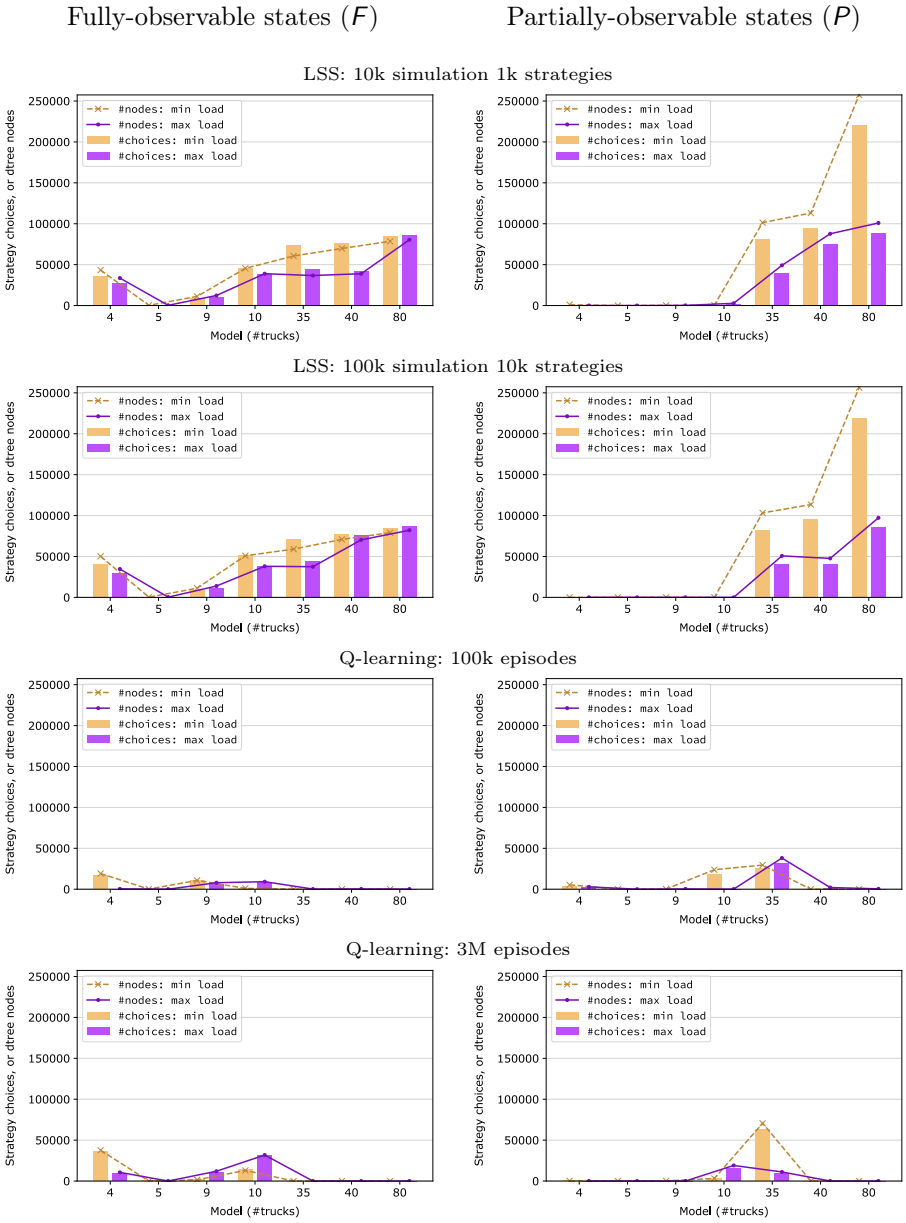
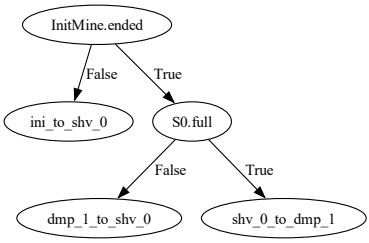


Fig. 9: Number of strategy choices vs. decision tree nodes

(a) Model 4, max load, LSS 10k 1k, P



(b) Model 5, max load, LSS 10k 1k, F

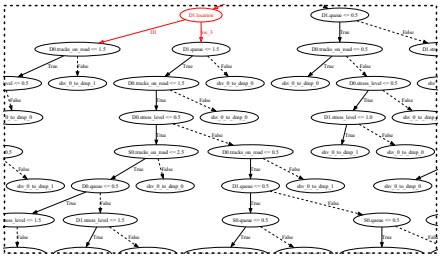


Fig. 10: Two decision trees of LSS 10k 1k

achieved by this algorithm in its two configurations are quite close to the uniform random strategy. And while Q-learning with a 3M-episodes budget did achieve a higher maximum load than the uniform, this is still comparable to the loads achieved by LSS. Since, on top of this, Q-learning consumes much more memory and runtime than both LSS and the uniform random strategy, the fact that it produces smaller decision trees is arguably not a useful advantage here.

Finally, Fig. 10 shows decision trees built by dtControl from strategies synthesised by modes running LSS 10k 1k on models 4 and 5 for maximum load (results for minimum are similar). The DT from Fig. 10a is small enough to follow its logic. Its first action is to initialise the mine at the (ore) shovel 0. Then, an action is chosen depending on the state of that shovel: if it is full, send a truck to transport useful material to the (ore) dump 1; otherwise, send a truck to shovel 0, awaiting for it to fill up. This simplistic decision tree reflects the six strategy choices that modes found during SMC following LSS 10k 1k on P model 4.

The DT of Fig. 10a is small because it was built from a P state, and despite the fact that model 4 is of middle size: by its number of ore shovels and dumps it is effectively larger than models 5 and 9. In contrast, model 5 is the smallest with 1 shovel and 2 dumps, none of which are for ore. Notwithstanding, the snippet shown in Fig. 10b—built from a F state—comes from a DT containing 269 nodes and 268 decisions. This includes the values of variables such as full shovels as discussed above, but also the number of trucks en route, queued at a shovel, or at a dump, and even the "program counter" of the `DumpSystem` process that should actually be irrelevant (marked in red in Fig. 10b). Compare this to the DT that both LSS 10k 1k and LSS 100k 10k build for maximum load using P states, and which consists of three nodes: do either `ini_to_dmp_0` (initialise the system) or else `shv_0_to_dmp_0`. This strategy sends a truck from *the* shovel to its closest dump, thus maximising the load in this toy example, which however becomes complicated if all model variables are observable as in the F case. For min load and P state, LSS 10k 1k and LSS 100k 10k never choose to send to `dmp_0` but do it to `dmp_1` instead.

Effectiveness of LSS and QL. Since no model checker implements the analysis of time-bounded expected accumulated rewards on MA, we have no "ground truth" to judge the effectiveness of LSS and QL in our new implementation. In order to have an idea of the possible performance of a model checker, we instead ran mcsta for a rather similar property, namely the expected accumulated *reachability* reward where the goal is given by the expiration of a randomly set timer. The outcome was that mcsta could check the three smallest models (not without some effort in the 4 trucks model for which it took over 15 minutes). For all of the larger models (namely, models 10, 35, 40 and 80), it ran out of memory.

7 Conclusion

Motivated by a novel case study challenge to optimise operations in an open-pit mine, we investigated the state of the art in SMC-based optimisation methods available in tools from the probabilistic verification community. We explained the LSS and QL approaches, with detailed pseudocode documenting our adaptations to the model of MA. To improve the effectiveness of the analysis, and to obtain explainable results, we extended the modes statistical model checker with support for partial observability to sample/learn based on selected model *features*, with a memory-efficient approach to collecting strategy decisions—the first approach to turn LSS's strategy identifiers into useful information—and with a connection to dtControl to obtain decision tree representations of complex strategies. Based on a Modest model of the open-pit mining operations, we compared the effectiveness and performance of LSS and QL, with and without selecting model features. We find that, somewhat surprisingly, the uniform random strategy is hard to beat by sampling or learning—but the methods are effective as evidenced by their ability to find strategies that succeed at *minimising* the work done in the mine. Identifying model features pays off, but always requires care and a good understanding of the case study at hand. In comparing LSS and QL, we see that the former works better despite its more simplistic, uninformed approach. In particular, LSS preserves the constant-memory property of SMC, while QL runs into the state space explosion problem just like standard model checking. We conjecture that this effect is evidence that, for our case study, no small "core" [31] exists that suffices to obtain good strategies while disregarding a large amount of the mine's behaviour. Our results reinforce earlier data [26] hinting at learning-based approaches being inferior to LSS on a level playing field, but potentially being able to provide much better results when carefully tweaked, such as by employing neural networks instead of explicit Q-tables and initialising episodes from non-initial states if the model allows to do so [21]. This motivates us to compare with neural network-based methods in future work. Successful results of the partition-refinement learning method implemented in Uppaal Stratego [29] suggest another approach worth comparing to.

Data availability. The models and tools/scripts to reproduce our experimental evaluation are archived and available at DOI 10.5281/zenodo.13327230 [10].

Acknowledgments. We are grateful to Matías D. Lee and Joaquín Feltes for discussions and insights on early versions of the open-pit mine model.

Funding. This work was supported by Agencia I+D+i grant PICT 2022-09-00580 (CoSMoSS), the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements 101008233 (MISSION) and 101067199 (ProSVED), the Interreg North Sea project STORM_SAFE, the NextGenerationEU project D53D23008400006 (SMARTITUDE) under the MUR PRIN 2022, NWO VIDI grant VI.Vidi.223.110 (TruSTy), and SeCyT-UNC grant 33620230100384CB (MECANO).

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018). https://doi.org/10.1145/3158668
2. Alarie, S., Gamache, M.: Overview of solution strategies used in truck dispatching systems for open pit mines. International Journal of Surface Mining, Reclamation and Environment **16**(1), 59–76 (2002). https://doi.org/10.1076/ijsm.16.1.59.3408
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
4. Ashok, P., Butkova, Y., Hermanns, H., Kretínský, J.: Continuous-time Markov decisions based on partial exploration. In: Lahiri, S.K., Wang, C. (eds.) 16th International Symposium on Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 11138, pp. 317–334. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_19
5. Ashok, P., Jackermeier, M., Kretínský, J., Weinhuber, C., Weininger, M., Yadav, M.: dtControl 2.0: Explainable strategy representation via decision tree learning steered by experts. In: Groote, J.F., Larsen, K.G. (eds.) 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 12652, pp. 326–345. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_17
6. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_28
7. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT). Lecture Notes in Computer Science, vol. 3185, pp. 200–236. Springer (2004). https://doi.org/10.1007/978-3-540-30080-9_7
8. Bellman, R.: A Markovian decision process. Journal of Mathematics and Mechanics **6**(5), 679–684 (1957)
9. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. **32**(10), 812–830 (2006). https://doi.org/10.1109/TSE.2006.104
10. Budde, C.E., D'Argenio, P.R., Hartmanns, A.: Artifact for digging for decision trees: A case study in strategy sampling and learning. Zenodo (2024). https://doi.org/10.5281/zenodo.13327230

11. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: An efficient statistical model checker for nondeterminism and rare events. Int. J. Softw. Tools Technol. Transf. **22**(6), 759–780 (2020). https://doi.org/10.1007/S10009-020-00563-2
12. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Poulsen, D.B., Legay, A., Wang, Z.: Uppaal-SMC: Statistical model checking for priced timed automata. In: Wiklicky, H., Massink, M. (eds.) 10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL). EPTCS, vol. 85, pp. 1–16 (2012). https://doi.org/10.4204/EPTCS.85.1
13. Butkova, Y., Fox, G.: Optimal time-bounded reachability analysis for concurrent systems. In: Vojnar, T., Zhang, L. (eds.) 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 11428, pp. 191–208. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_11
14. Butkova, Y., Hartmanns, A., Hermanns, H.: A Modest approach to Markov automata. ACM Trans. Model. Comput. Simul. **31**(3), 14:1–14:34 (2021). https://doi.org/10.1145/3449355
15. Butkova, Y., Hatefi, H., Hermanns, H., Krcál, J.: Optimal continuous time Markov decisions. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) 13th International Symposium on Automated Technology for Verification and Analysis (ATVA). Lecture Notes in Computer Science, vol. 9364, pp. 166–182. Springer (2015). https://doi.org/10.1007/978-3-319-24953-7_12
16. Bäuerle, N., Rieder, U.: Markov Decision Processes with Applications to Finance. Springer (2011). https://doi.org/10.1007/978-3-642-18324-9
17. D'Argenio, P.R., Hartmanns, A., Sedwards, S.: Lightweight statistical model checking in nondeterministic continuous time. In: Margaria, T., Steffen, B. (eds.) 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Lecture Notes in Computer Science, vol. 11245, pp. 336–353. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_22
18. D'Argenio, P.R., Legay, A., Sedwards, S., Traonouez, L.M.: Smart sampling for lightweight verification of Markov decision processes. Int. J. Softw. Tools Technol. Transf. **17**(4), 469–484 (2015). https://doi.org/10.1007/S10009-015-0383-0
19. David, A., Jensen, P.G., Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Uppaal Stratego. In: Baier, C., Tinelli, C. (eds.) 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_16
20. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: 25th Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 342–351. IEEE Computer Society (2010). https://doi.org/10.1109/LICS.2010.41
21. Gros, T.P., Groß, J., Höller, D., Hoffmann, J., Klauck, M., Meerkamp, H., Müller, N.J., Schaller, L., Wolf, V.: DSMC evaluation stages: Fostering robust and safe behavior in deep reinforcement learning – extended version. ACM Trans. Model. Comput. Simul. **33**(4), 17:1–17:28 (2023). https://doi.org/10.1145/3607198
22. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013). https://doi.org/10.1007/S10703-012-0167-Z
23. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51

24. Hartmanns, A., Hermanns, H.: A Modest Markov automata tutorial. In: Krötzsch, M., Stepanova, D. (eds.) 15th International Reasoning Web Summer School on Explainable Artificial Intelligence (RW). Lecture Notes in Computer Science, vol. 11810, pp. 250–276. Springer (2019). https://doi.org/10.1007/978-3-030-31423-1_8
25. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: Sankaranarayanan, S., Sharygina, N. (eds.) 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 13993, pp. 469–488. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_24
26. Hartmanns, A., Klauck, M.: The Modest state of learning, sampling, and verifying strategies. In: Margaria, T., Steffen, B. (eds.) 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Lecture Notes in Computer Science, vol. 13703, pp. 406–432. Springer (2022). https://doi.org/10.1007/978-3-031-19759-8_25
27. Hatefi-Ardakani, H.: Finite horizon analysis of Markov automata. Ph.D. thesis, Saarland University, Germany (2017), http://scidok.sulb.uni-saarland.de/volltexte/2017/6743/
28. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). https://doi.org/10.1007/S10009-021-00633-Z
29. Jensen, P.G., Larsen, K.G., Mikucionis, M.: Playing wordle with uppaal stratego. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13560, pp. 283–305. Springer (2022). https://doi.org/10.1007/978-3-031-15629-8_15
30. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1-2), 99–134 (1998). https://doi.org/10.1016/S0004-3702(98)00023-X
31. Kretínský, J., Meggendorfer, T.: Of cores: A partial-exploration framework for Markov decision processes. Log. Methods Comput. Sci. **16**(4) (2020), https://lmcs.episciences.org/6833
32. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) 23rd International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
33. Law, A.M.: Simulation Modeling and Analysis. McGraw-Hill series in industrial engineering and management science, McGraw-Hill Education, 5th edn. (2015)
34. Legay, A., Sedwards, S., Traonouez, L.M.: Scalable verification of Markov decision processes. In: Canal, C., Idani, A. (eds.) Software Engineering and Formal Methods – Revised Selected Papers of the SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, and WS-FMDS. Lecture Notes in Computer Science, vol. 8938, pp. 350–362. Springer (2014). https://doi.org/10.1007/978-3-319-15201-1_23
35. Moradi Afrapoli, A., Askari-Nasab, H.: Mining fleet management systems: a review of models and algorithms. International Journal of Mining, Reclamation and Environment **33**(1), 42–60 (2019). https://doi.org/10.1080/17480930.2017.1336607

36. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real Time Syst. **53**(3), 354–402 (2017). https://doi.org/10.1007/S11241-017-9269-4
37. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
38. Sutton, R.S., Barto, A.G.: Reinforcement learning – An introduction. Adaptive computation and machine learning, MIT Press (1998)
39. Watkins, C.J.C.H., Dayan, P.: Q-learning. Mach. Learn. **8**, 279–292 (1992). https://doi.org/10.1007/BF00992698