Reinforcement Learning of Risk-Constrained Policies in Markov Decision Processes

Tomáš Brázdil,[1] **Krishnendu Chatterjee,**[2] **Petr Novotný,**[1] **Jiří Vahala**[1]
[1]Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbrazdil, petr.novotny, xvahala1}@fi.muni.cz
[2]Institute of Science and Technology Austria, Klosterneuburg, Austria
Krishnendu.Chatterjee@ist.ac.at

Abstract

Markov decision processes (MDPs) are the defacto framework for sequential decision making in the presence of stochastic uncertainty. A classical optimization criterion for MDPs is to maximize the expected discounted-sum payoff, which ignores low probability catastrophic events with highly negative impact on the system. On the other hand, risk-averse policies require the probability of undesirable events to be below a given threshold, but they do not account for optimization of the expected payoff. We consider MDPs with discounted-sum payoff with failure states which represent catastrophic outcomes. The objective of *risk-constrained* planning is to maximize the expected discounted-sum payoff among risk-averse policies that ensure the probability to encounter a failure state is below a desired threshold. Our main contribution is an efficient risk-constrained planning algorithm that combines UCT-like search with a predictor learned through interaction with the MDP (in the style of AlphaZero) and with a risk-constrained action selection via linear programming. We demonstrate the effectiveness of our approach with experiments on classical MDPs from the literature, including benchmarks with an order of 10^6 states.

1 Introduction

MDPs with discounted-sum objectives. A classical problem in artificial intelligence is sequential decision making under uncertainty. The standard model incorporating both decision-making choices and stochastic uncertainty are Markov decision processes (MDPs) (Howard 1960; Puterman 1994). MDPs have a wide range of applications, from planning (Russell and Norvig 2010), to reinforcement learning (Kaelbling, Littman, and Moore 1996), robotics (Kress-Gazit, Fainekos, and Pappas 2009), and verification of probabilistic systems (Baier and Katoen 2008), to name a few. The objective in decision making under uncertainty is to optimize a payoff function. A fundamental payoff function is the *discounted-sum payoff*, where every transition of the MDP is assigned a reward, and for an infinite path (that consists of an infinite sequence of transitions) the payoff is the discounted-sum of the rewards of the transitions.

Expectation optimization and risk. In the classical studies of MDPs with discounted-sum payoff the objective is to obtain policies that maximize the expected payoff. However, this ignores that low probability failure events can have highly negative impact on the system. In particular, in safety critical systems, or systems with high cost for failures, policies with high expected reward can be associated with risky actions with undesirable chances of failure.

CCMDPs and risk-reward tradeoff. Chance- (or risk-) constrained MDPs (CCMDPs) introduce *chance constraint* or *risk bound* which provides a bound on the allowed probability of failure of a policy (Rossman 1977; Santana, Thiébaux, and Williams 2016; Ayton and Williams 2018). In particular, we consider MDPs equipped with a set of failure states which represent catastrophic outcomes. The probability to encounter any failure state represents the risk. Given a desired probability threshold for the risk bound, a risk-averse policy ensures that the probability of failure does not exceed the given bound. On one hand, policies with low-risk may ensure little expected payoff; on the other hand, policies with high expected payoff can be associated with high risk. Thus the relevant question to study is the interplay or the tradeoff of risk and expected payoff. In this work we study the following *risk-constrained* planning problem: given a risk bound, the objective is to maximize the expected payoff among all risk-averse policies that ensure the failure probability is at most the risk bound.

Motivating scenarios. Risk-constrained planning is natural in several scenarios. For example, in planning under uncertainty (e.g., autonomous driving) certain events (e.g., the distance between two cars, or the distance between a car and an obstacle, being less than a specified safe distance) must be ensured with low probability. Similarly, in scenarios such as a robot exploring an unknown environment for natural resources a significant damage of the robot ends the mission, and must be ensured with low probability. However, the goal is to ensure effective exploration within the specified risk bounds, which naturally gives rise to the risk-constrained planning problem we consider.

Our contributions. The risk-constrained planning problem (or CCMDPs) have been considered in previous works such as (Santana, Thiébaux, and Williams 2016; Ayton and Williams 2018). However these works consider only deterministic policies, and randomized (or mixed) policies are strictly more powerful for the risk-constrained planning problem (Altman 1999). A possible approach for the risk-

constrained planning problem is via linear programming or dynamic programming methods, however, they scale poorly and are unsuitable for large state spaces (Ayton and Williams 2018). Our main contribution is an efficient risk-constrained planning algorithm that combines UCT-like search with a predictor learned through interaction with the MDP and with a risk-constrained action selection via linear programming over a *small sampled* tree-shaped MDP. Since the linear programming is over a sampled sub-MDP, our algorithm is scalable as compared to linear programming over the entire MDP, while the use of predictor significantly enhances the search. By using the predictor we lose formal guarantees on the solution, but gain in performance. We also show that despite the lack of guarantees, our method converges to well-behaved policies in practice. We demonstrate this with experiments on classical MDPs from the literature, including benchmarks with an order of 10^6 states.

Related Work. Discounted-payoff MDPs are a well-established model (Puterman 1994; Filar and Vrieze 1997). The notion of ensuring risk constraints is also well-studied (Rossman 1977; Hou, Yeoh, and Varakantham 2016). Moreover, CCMDPs can be considered as a special case of constrained MDPs (CMDPs) (Altman 1999). CMDPs are often solved using linear programming approaches which do not scale to large MDPs (Ayton and Williams 2018). The works most closely related to the problem we consider are as follows: First, the risk-constrained planning for partially-observable MDPs (POMDPs) with deterministic policies has been considered in (Santana, Thiébaux, and Williams 2016), and risk-constrained MDPs with deterministic policies have been considered in (Ayton and Williams 2018). In contrast, we consider randomized policies, which are more powerful for risk-constrained planning. Another related approach for POMDPs are *constrained POMDPs* (Undurti and How 2010; Poupart et al. 2015), where the objective is to maximize the expected payoff ensuring that the expected payoff of another quantity is bounded. Risk-constrained MDP optimization with randomized policies was considered in (Teichteil-Königsbuch 2012). There they consider optimization under formally guaranteed PCTL constraints via an iterative linear programming (LP) over the whole state space. The largest benchmark reported in the referenced paper has 75^2 states, while we report MDPs with up to ca. $6.5 \cdot 10^6$ states. Hence, the method of (Teichteil-Königsbuch 2012) is preferable where guarantees are a priority while RAlph is preferable where scalability is a priority. The paper (Baumgartner, Thiébaux, and Trevizan 2018) considers *stochastic shortest path* under PLTL constraints, i.e. the rewards are positive costs and we minimize the expected cost of reaching a target. In contrast, we consider arbitrary rewards under safety constraints.

Several problems related to risk-constrained planning with other objectives have been considered, such as: (a) risk threshold 0 for long-run average and stochastic shortest path problems MDPs (Bruyère et al. 2014; Randour, Raskin, and Sankur 2015); (b) general risk threshold for long-run average payoff in MDPs (Chatterjee, Komárková, and Kretínský 2015). (c) risk bound 0 for discounted-sum POMDPs (Chatterjee et al. 2017); and (d) general risk bound for discounted-sum POMDPs (Chatterjee et al. 2018). In all these works the risk is formulated as risk of the payoff being below a given value rather than of reaching failure states. Moreover, these works (apart from d)) focus on dynamic programming methods, rather than scalable algorithms for large MDPs. Although d) also uses linear programming over a sampled sub-MDP, it does not use predictors and its tree-search procedure is closer to the original UCT (Kocsis and Szepesvári 2006) than to its more sophisticated version used by AlphaZero (Silver et al. 2017; 2018). While the algorithm of d) can be adapted to risk-constrained MDPs with reachability risk, our experiments show that our new algorithm scales much better.

2 Preliminaries

Definition 1 *A* Markov decision process (MDP) *is a tuple* $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \delta, rew, s_0, \gamma)$ *where* $\mathcal{S}$ *is a set of* states*,* $\mathcal{A}$ *is a set of* actions*,* $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{D}(\mathcal{S})$ *is a* probabilistic transition function *that given a state* $s \in \mathcal{S}$ *and an action* $a \in \mathcal{A}$ *gives the probability distribution over the successor states,* $rew : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *is a* reward function*,* s_0 *is the* initial state*, and* $\gamma \in (0, 1]$ *is the* discount factor*. We abbreviate* $\delta(s, a)(s')$ *by* $\delta(s'|s, a)$*.*

Policies. The interaction with an MDP starts in the initial state s_0 and proceeds sequentially through a *policy* π, a computable function which acts as a blueprint for selecting actions, producing longer and longer *history* of actions and observations. Formally, a history is an alternating sequence of states and actions starting and ending with a state. The initial history is $H_0 = s_0$. In every time step $i \in \{0, 1, 2, \ldots\}$ the interaction already produced some history H_i whose last state $last(H_i)$ is the current state S_i of the system. In such a situation, π selects an action $A_i \in \mathcal{A}$ to play in step i. The choice may depend on the whole past history, and it might also be randomized, i.e. $A_i \sim \pi(H_i)$. The agent then gets an immediate reward $Rew_i = rew(S_i, A_i)$ and proceeds to the next state S_{i+1}, which is sampled according to the transition function, i.e. $S_{i+1} \sim \delta(S_i, A_i)$. Thus, the current history is now $H_{i+1} = H_i A_i S_{i+1}$, obtained from the previous history by appending the last selected action and the resulting state. Throughout the text we denote by S_i, A_i, H_i the random variables returning the state, action, and current history in step i, while the notation s, a, h, etc. is reserved for concrete states/actions/histories (i.e. elements of the co-domains of $S_i/A_i/H_i$).

We denote by $\mathbb{P}^\pi(E)$ the probability of an event E under policy π, and by $\mathbb{E}^\pi[X]$ the expected value of a random variable X under π.

Payoffs. The expected payoff of a policy π from state s is the value $Payoff(\pi, s) = \mathbb{E}^\pi_s[\sum_{i=0}^\infty \gamma^i \cdot Rew_i]$.

Risk-Constrained Optimization. To encompass the notion of an undesirable event, we equip each MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \delta, rew, s_0, \gamma)$ with a set $F_\mathcal{M} \subseteq \mathcal{S}$ of *failure states*. A *risk* of a policy π is then the probability that a failure state is encountered: $Risk(\pi) = \mathbb{P}^\pi_s\Big(\bigcup_{i=0}^\infty \{S_i \in F_\mathcal{M}\}\Big)$.We assume that each $s \in F_\mathcal{M}$ is a *sink,* i.e. $\delta(s|s, a) = 1$ and

$rew(s, a) = 0$ for all $a \in \mathcal{A}$. Hence, $F_{\mathcal{M}}$ models failures after which the agent has to cease interacting with the environment (e.g. due to being destroyed).

The risk-constrained planning problem is defined as follows: given an MDP $\mathcal{M}$ and a *risk threshold* $\Delta \in [0, 1]$, find a policy π which maximizes $Payoff(\pi)$ subject to the constraint that $Risk(\pi) \leq \Delta$. If there is no *feasible* policy, i.e. a policy s.t. $Risk(\pi) \leq \Delta$, then we want to find a policy that minimizes the risk and among all such policies optimizes the expected payoff.

In this paper, we present RAlph (a portmanteau of "Risk" and "Alpha"), an online algorithm for risk-constrained planning. Inspired by the successful approach of AlphaZero, RAlph combines a UCT-like tree search with evaluation of the leaf nodes via a suitable *predictor* learned through a repeated interaction with the system. On top of this, we augment the algorithm's action-selection phase with a risk-constrained mechanism based on evaluation of a linear program over the constructed search tree.

3 The Algorithm

Predictor. First we formally define the notion of a predictor. A predictor is a θ-parameterized function $f_\theta \colon \mathcal{S} \to \mathbb{R} \times [0, 1] \times [0, 1]^{|\mathcal{A}|}$ assigning to each state s the tuple $f_\theta(s) = (v, r, \mathbf{p})$ which predicts the parameters of some policy π: v is the predicted expected payoff of π from s, r is the predicted risk of π from s, and $\mathbf{p}$ is the vector of *prior probabilities* over the set $\mathcal{A}$ in s. We defer the details of the predictor implementation, its parameters, and the learning technique used to update them, to Subsection 3.2.

RAlph: Overall Structure. The main training and evaluation loops of RAlph are given in Algorithm 1. As in other algorithms based on search through the search tree, termination is ensured by searching only up to a given finite horizon H. In the training phase, RAlph repeatedly samples episodes of the agent-environment interaction, using the `RAlph-episode` procedure described in Subsection 3.1. After each batch of episodes is sampled, the gathered data are used to retrain the predictor via the procedure `Train`, described in Subsection 3.2. Once the training is finished, we fix the predictor and continue to the evaluation phase.

3.1 Risk-Constrained Tree Search

In this subsection we describe the procedure `RAlph-episode` (Algorithm 2). We first describe the conceptual elements of the algorithm, then the data structures it operates on, and finally the algorithm itself.

Overview. The algorithm interacts with the MDP for H steps, each step i resulting in the (randomized) choice of some action a_i to be played. In every step, RAlph first expands the search tree $\mathcal{T}$ by iterating the usual 4-phase MCTS simulations (node selection, tree expansion, leaf evaluation, backpropagation, see procedure `Simulate`). We follow the spirit of (Silver et al. 2018) and use the predictor f_θ to evaluate the leaf nodes. Using the data stored within the tree, we then compute the distribution from which a_i is sampled.

To accommodate the risk, we extend the AlphaZero-like MCTS with several conceptual changes, outlined below.

Algorithm 1: Training and evaluation of RAlph.

```
1  procedure RAlph-train
       Input: MDP M (with a horizon H), risk bound Δ,
              no. of training episodes m, batch size n
2      episodes ← 0; mod ← "train"; initialize f_θ;
3      while episodes < m do
4          batch ← 0; Data ← ∅;
5          while batch < n do
6              E ← RAlph-episode (M,H,f_θ,Δ,mod);
7              batch ← batch + 1;
8              episodes ← episodes + 1;
9              Data ← Data ∪ {E};
10         θ ← Train (θ, Data)
11 procedure RAlph-evaluate
       Input: MDP M (with a horizon H), risk bound Δ,
              pre-trained predictor f_θ
12     mod ← "eval";
13     while true do RAlph-episode (M,H,f_θ,mod)
```

Risk-constrained sampling of a_i. In the action selection phase, we solve a linear program (LP) over $\mathcal{T}$, which yields a local policy that maximizes the estimated payoff while keeping the estimated risk below the threshold Δ (line 9; described below). The distribution ξ_i used by the local policy in the first step is then used to sample a_i.

Risk-constrained exploration. Some variants of AlphaZero enable additional exploration by selecting each action with a probability proportional to its exponentiated visit count (Silver et al. 2017). Our algorithm use a technique which perturbs the distribution computed by the LP while keeping the risk estimate of the perturbed distribution below the required threshold (line 11; described below).

Risk predictor. In our algorithm, the predictor is extended with risk prediction.

Estimation of alternative risk. The risk threshold must be updated after playing an action, see Example 2, since each possible outcome of the action has a potential contribution towards the global risk. We use linear programming and the risk predictor to obtain an estimate of these contributions.

Data Structures. The search tree (Silver and Veness 2010), denoted by $\mathcal{T}$, is a dynamic tree-like data structure whose nodes correspond to histories of $\mathcal{M}$. We name the nodes directly by the corresponding histories. Each child of a node h is of the form hat, where $a \in \mathcal{A}$ and $t \in \mathcal{S}$ are s.t. $\delta(t|last(h), a) > 0$. Each node h has these attributes:

- $h.N$, the visit count of h;
- $h.v$ and $h.r$; the last predictions of payoff and risk obtained by f_θ for $last(h)$.

Moreover, for each action $a \in \mathcal{A}$ we have the attributes:

- $h.N_a$, counting the number of uses of a during visits of h;
- $h.V_a$, the average payoff accumulated by past simulations after using a in h;
- $h.p_a$, the last prediction of a prior probability of a obtained by f_θ in $last(h)$.

We also have the following derived attributes: $h.V_{\min} = \min_{a \in \mathcal{A}} h.V_a$; and $h.V_{\max} = \max_{a \in \mathcal{A}} h.V_a$. These are recomputed to match the defining formulae whenever some $h.V_a$ is changed. Every newly created node is initialized with zero attributes.

We denote by $root(\mathcal{T})$ the root of $\mathcal{T}$ and by $leaf(\mathcal{T})$ the set of leafs of $\mathcal{T}$. Also, for a node h we denote by $\mathcal{T}(h)$ the sub-tree rooted in h.

Episode Sampling: Overall Structure. In Algorithm 2, a single search tree $\mathcal{T}$ is used as a global dynamic structure. In this paragraph, we provide a high-level description; the following paragraphs contain details of individual components of the algorithm. The main loop (lines 4–19) has a UCT-like structure. In every decision step i, the tree is extended via a sequence of simulations (described below); the number of simulations being either fixed in advance or controlled by setting a timeout. After that, we solve a linear program over $\mathcal{T}$ defined below (line 9). This gives us a distribution ξ_i over actions as well as a risk distribution τ_i over the child nodes of the root node ν. Informally, $\tau(\nu bt)$ is the estimated future risk of hitting a failure state after playing b and transitioning into t. After solving the program, we sample an action a_i to play and then the corresponding successor state s_{i+1}, obtaining an immediate reward ρ_i. The risk threshold is then updated to by the formula on lines 16–18, where $altrisk$ is the probability placed by the risk distribution on all the histories not consistent with the current history $\nu a_i s_{i+1}$. Finally, we prune away all parts of the tree not consistent with the current history and continue into a next iteration.

Simulations & UCT Selection. Simulations also follows the standard UCT scheme. In every simulation, we traverse the tree from top to bottom by selecting, in every node h, an action $a = \arg\max_{a \in \mathcal{A}} \texttt{UCT}(h, a)$, where

$$\texttt{UCT}(h,a) = \frac{h.V_a - h.V_{\min}}{h.V_{\max} - h.V_{\min}} + C \cdot h.p_a \cdot \sqrt{\frac{\ln(h.N)}{h.N_a + 1}}.$$

Here C is a suitable *exploration constant,* a parameter fixed in advance of the computation.

Upon reaching a leaf node h, we expand $\mathcal{T}$ by adding *all* possible child nodes of h (lines 28–31). Finally, we perform a bottom-up traversal from h to the root, updating the node and action statistics with the data from the current simulation (lines 35–40). Note that the payoff and risk from h (which was not visited by a simulation before) is estimated via the predictor, unless h corresponds either to being trapped in a failure state (in which case its risk is clearly one and future payoff 0) or to running out of the horizon without hitting $F_{\mathcal{M}}$, in which case the risk and future payoff are both 0.

Linear Program. We first fix some notation. For a history $h = s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ we define its length $len(h)$ to be n and its payoff to be $Payoff(h) = \sum_{i=0}^{len(h)-1} \gamma^i \cdot rew(s_i, a_i)$.

The procedure $\texttt{Solve-LP}(\mathcal{T}, \Delta)$ constructs a linear program $\mathcal{L}$, which has variables $x_h, x_{h,a}$ for every node $h \in \mathcal{T}$ and every $a \in \mathcal{A}$, and is pictured in Figure 1.

The LP $\mathcal{L}$ encodes a probabilistic flow induced by some policy (constraints (1)–(4), which we together denote by $Flow(\mathcal{T})$), x_h being the probability that the policy produces a history h and $x_{h,a}$ the probability that h is produced and afterwards a is selected. We aim to maximize the expected payoff of such a policy (with payoffs outside the tree estimated by predictions stored in $h.v$) while keeping the (estimated) risk below Δ (constraint (5)). Hence, the procedure `Solve-LP` returns an action distribution ξ_i s.t. $\xi_i(a) = x_{root(\mathcal{T}),a}$ for each $a \in \mathcal{A}$.

$$\textit{max} \sum_{h \in leaf(\mathcal{T})} x_h \cdot (Payoff(h) + \gamma^{len(h)} \cdot h.v) \textit{ subject to}$$

$$x_{root(\mathcal{T})} = 1 \tag{1}$$

$$x_h = \sum_{a \in \mathcal{A}} x_{h,a} \qquad \text{for } h \in \mathcal{T} \setminus leaf(\mathcal{T}) \tag{2}$$

$$x_{hbt} = x_{h,b} \cdot \delta(t \mid last(h), b) \quad \text{for } h, hbt \in \mathcal{T} \tag{3}$$

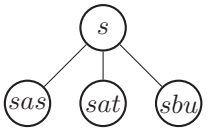
$$0 \le x_h \le 1, \quad 0 \le x_{h,a} \le 1 \qquad \text{for } h \in \mathcal{T}, a \in \mathcal{A} \tag{4}$$

$$\sum_{h \in leaf(\mathcal{T})} x_h \cdot h.r \le \Delta \tag{5}$$

Figure 1: The Linear program $\mathcal{L}$.

If $\Delta = 1$, there is no need for constrained sampling. Hence, in such a case we omit the LP step altogether and make the selection based on the action visit count.

Example 1 *Consider an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \delta, rew, s, \gamma)$ with $\mathcal{S} = \{s, t, u\}$ and $\mathcal{A} = \{a, b\}$ s.t. $\delta(s|s,a) = \delta(t|s,a) = \frac{1}{2}$, $\delta(u|s,b) = 1$. The states t, u are sinks, $F_{\mathcal{M}} = \{t\}$, and $rew(s,a) = 1$ (all other rewards are 0). We put $\gamma = 0.95$, and $\Delta = 0.6$. Assume, for the sake of simplicity, that we have just one simulation per step, which, in the initial step, yields the following tree:*



Next, assume that the current predictor predicts risk 0.4 for s, and 0.1 for u, while the predicted payoffs are 0 for t, u and 1 for s. Then $\mathcal{L}$ asks to maximize $x_{sas} \cdot 1.95 + x_{sat}$ under the following constraints: $x_s = 1$, $x_s = x_{s,a} + x_{s,b}$, $x_{sas} = 0.5 \cdot x_{s,a}$, $x_{sat} = 0.5 \cdot x_{s,a}$, $x_{sbu} = x_{s,b}$, $0.4 \cdot x_{sas} + x_{sat} + 0.1 \cdot x_{sbu} \le 0.6$ (and all variables in $[0,1]$).

Risk Distribution. The choice of actions according to ξ_i is randomized, as is the subsequent sample of the successor state. Each outcome of these random choices contributes some risk to the overall risk of our randomized policy.

Example 2 *Consider $\mathcal{M}$ as in Example 1, with $\Delta = 0.6$. If the agent selects action a and the system transitions into the non-failure state s, the agent made the risk in the root node s equal to $r_0 = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot r_1$, where r_1 is the probability of hitting a failure state after continuing the play from s. To ensure that $r_0 \le \Delta$, we must ensure $r_1 \le 0.2$. Hence, in the next step, Δ must be updated to 0.2.*

Hence, when making a step, we need to compute a risk distribution τ_i which assigns to each possible outcome (i.e. each child of $root(\mathcal{T})$) an estimate of its risk contribution.

Algorithm 2: The episode sampling of RAlph.

```
1  procedure RAlph-episode (M,H,f_θ,Δ,mod)
2      global T;
3      initialize T to one node s_0; E ← empty sequence;
4      for i ← 0 to H − 1 do
5          ν ← root(T); s_i ← last(ν);
6          repeat
7              Simulate (M, H − i,T) ;    // build T
8          until timeout;
9          ξ_i, τ_i ← Solve-LP(T, Δ);
10         if mode = "train" then
11             ξ_i ← RiskAwareExplore(T, ξ_i)
12         a_i ← sample from ξ_i;
13         ρ_i ← rew(s_i, a_i);
14         s_{i+1} ← sample from δ(s_i, a_i);
15         append (s_i, ξ_i, ρ_i) to E;
16         alt ← {ν' ∈ T | ν' child of ν s.t. ν' ≠ ν a_i s_{i+1}};
17         altrisk ← Σ_{ν'∈alt} τ_i(ν');
18         Δ ← (Δ − altrisk)/τ_i(ν a_i s_{i+1}) ;
19         T ← sub-tree of T rooted in ν a_i s_{i+1}
20     return E
21 procedure Simulate(M, steps, T)
22     h ← root(T); depth ← 0;
23     while h is not a leaf of T do
24         a ← arg max_{a∈A} UCT(h, a);
25         s ← sample from δ(last(h), a);
26         h ← has; depth ← depth + 1;
27     if last(h) ∉ F_M ∧ depth < steps then
28         foreach b ∈ A do
29             foreach t ∈ S s.t. δ(t|last(h), b) > 0 do
30                 initialize a new leaf hbt, add it to T as a
                    child of h ;
31                 Predict (hbt)
32     else if last(h) ∈ F_M then h.r ← 1;
33     else h.r ← 0;
34     val ← h.v; h.N ← h.N + 1;
35     while h ≠ root(T) do
36         let h = h'bt where b ∈ A, t ∈ S;
37         h'.N ← h'.N + 1; h'.N_b ← h'.N_b + 1;
38         val ← rew(last(h'), b) + γ · val;
39         h'.V_b ← h'.V_b + (val − h'.V_b)/h'.N_b;
40         h ← h'
41 procedure Predict (h, f_θ)
42     (h.v, h.r, (h.p_a)_{a∈A}) ← f_θ(last(h))
```

This distribution used to update the risk threshold Δ after a concrete outcome of the choices is observed (lines 16 – 18). In our experiments, we use the *optimistic* risk estimate, which assigns to each child h of the root the minimal risk achievable in the sub-tree rooted in h (the risk of leafs being estimated by f_θ). Formally, we set $\tau_i(h)$ to be the optimal value of a linear program $\mathcal{L}_{risk}(h)$ with constraints $Flow(\mathcal{T}(h))$ and with the objective to minimize $\sum_{h' \in leaf(\mathcal{T}(h))} x_{h'} \cdot h'.r$.

Infeasible LP. The linear program $\mathcal{L}$ might be infeasible, either because there is no policy satisfying the risk threshold or because the risk estimates are too imprecise (and pessimistic). In such a case, we relax the overall risk constraint while trying to stay as risk-averse as possible. Formally, we reset Δ to be the minimal risk achievable in the current tree, i.e. the optimal value of $\mathcal{L}_{risk}(root(\mathcal{T}))$. (Note that $\mathcal{L}_{risk}(h)$ is feasible for each node h.) We then again solve $\mathcal{L}$, which is guaranteed to be feasible under the new Δ.

Exploration. The exploration-enhancing procedure `RiskAwareExplore` (line 11) uses a pre-set function $expl$ which, given an integer j, returns a value from $[0, 1]$. When called, the procedure performs a Bernoulli trial with parameter $expl(j)$, where j is the number of past calls of the procedure. Depending on outcome, it either decides to not explore (entailing no change of ξ_i); or to explore, in which case we modify ξ_i in a way depending on whether the computation of ξ_i required just one call of the linear solver (i.e. if $\mathcal{L}$ was feasible without relaxing Δ) or not.

If $\mathcal{L}$ was feasible on the first try, we perturb ξ_i using the standard Boltzmann (softmax) formula (Kaelbling, Littman, and Moore 1996), i.e. the perturbed probabilities are proportional to an exponential function of the original probabilities. The perturbed distribution $\tilde{\xi}_i$ might be too risky, which is indicated by violating the risk constraint $\sum_{b \in \mathcal{A}, t \in \mathcal{S}} \tau_i(root(\mathcal{T})bt) \cdot \tilde{\xi}_i(root(\mathcal{T})bt) \leq \Delta$. If this is the case, we find, using the method of Lagrange multipliers, a distribution which satisfies the risk constraint and minimizes the squared distance from $\tilde{\xi}_i$; such a distribution is then output by `RiskAwareExplore`.

If we needed to relax Δ to solve $\mathcal{L}$, we assume the predictions to be too pessimistic and opt for a more radical exploration. Hence, we ignore $\mathcal{L}$ altogether and instead select actions proportionally to their UCT values, i.e. we put $\xi_i(a) = \texttt{UCT}(root(\mathcal{T}), a) / \sum_{b \in \mathcal{A}} \texttt{UCT}(root(\mathcal{T}), b)$.

3.2 Predictor & Training

In principle any predictor (e.g. a neural net) can be used with RAlph. In this paper, as a proof of concept, we use a simple table predictor, directly storing the estimates for each state s (the parameter θ can then be identified with the table, i.e. $\theta(s) = f_\theta(s)$).

Each episode produces a data element $\eta = (s_0, \xi_0, \rho_0) \cdots (s_{H-1}, \xi_{H-1}, \rho_{H-1})$, where s_i, ξ_i, ρ_i are the current state, the distribution on actions used, and the reward obtained in step i, respectively. For every step i of this episode we compute the discounted accumulated payoff $G^i_\eta = \sum_{j=i}^{H-1} \gamma^{i-j} \cdot \rho_j$ from that step on; similarly, for risk we set R^i_η to 1 if some $s_j \in F_\mathcal{M}$ for $j \geq i$, and to 0 otherwise; for action probabilities we denote $P^i_\eta = \xi_i$. For each state s encountered on η we put $I_\eta(s) = \{i \mid 0 \leq i \leq H-1 \wedge s_i = s\}$.

The state statistics across all episodes in $Data$ are gathered in an every-visit fashion (Sutton and Barto 2018). I.e., we compute the quantities $N(s) = \sum_{\eta \in Data} |I_\eta(s)|$ (the total visit count of s), $G(s) = \sum_{\eta \in Data, i \in I_\eta(s)} G^i_\eta$, $R(s) =$

```
1 1 1 1 1 1
1 A B x   1
1 D C E g 1
1 1 1 1 1 1
```

Figure 2: Example of a Hallway MDP. Symbols '1', 'x', 'g' represent wall/trap/gold cell respectively; the other symbols are empty cells. The agent starts in B facing east.

$\sum_{\eta \in Data, i \in I_\eta(s)} R^i_\eta$, and $P(s) = \sum_{\eta \in Data, i \in I_\eta(s)} P^i_\eta$. These are then averaged to $\widetilde{G}(s) = G(s)/N(s)$, $\widetilde{R}(s) = R(s)/N(s)$, and $\widetilde{P}(s) = P(s)/N(s)$ (operations on probability distributions are componentwise). Together, these averages form a target table $\widetilde{\theta}$ such that $\widetilde{\theta}(s) = (\widetilde{G}(s), \widetilde{R}(s), \widetilde{P}(s))$. Finally, we perform the update $\theta \leftarrow \theta + \alpha(\widetilde{\theta} - \theta)$, where α is a pre-set learning rate.

This scheme can be generalized to more sophisticated predictors, which only requires replacing the final update with a gradient descent in the parameter space. The implementation and evaluation of these predictors is left for future work.

4 Experiments

Benchmarks. We implemented RAlph and evaluated it on two sets of benchmarks. The first one is a modified, perfectly observable version of Hallway (Pineau et al. 2003; Smith and Simmons 2004) where we control a robot navigating a grid maze using three possible moves: forward, turn right, turn left. Depending on the instance, the forward movement might be subject to random perturbations (the robot shifted to the right or left of the target cell). For every step the robot incurs a (fixed) negative penalty. Some cells of the maze contain "gold," collection of which yields a positive reward. Cells may also contain traps. Entering a trap entails a small chance of destroying the robot (i.e. going to a failure state). Each gold piece can be only collected once, so each additional gold cell doubles the size of the state space.

As a second benchmark, we consider a controllable *random walk (RW)*. The state space here are integers in a fixed 0-containing interval, representing the agent's wealth. At each step, the agent can chose between two actions - safer and riskier. Each action has a probabilistic outcome: either the wealth is increased or lost. The riskier action has higher expected wealth gain, but greater chance of loss. We start with a small positive wealth, and the failure states are those where the wealth is lost, i.e. non-positive numbers. In each step, the agent receives a reward/penalty equal to wealth gained/lost. The goal is to surpass a given wealth level L as fast as possible: the agent incurs a small penalty for every step up to the first step when she surpasses L.

Comparison. For comparison, we reimplemented the online RAMCP algorithm from (Chatterjee et al. 2018) slightly modified (per suggestion in the source paper) so as to allow for state-based risk. This should allow us to evaluate the effect of RAlph's crucial features (learning and prediction, risk-averse exploration) on the performance. To get a fair comparison, our implementation of RAMCP shares as much common code with RAlph as possible. In particular, both algorithms employ UCT-like simulations. We denote by sim the number of these simulations invoked per decision.

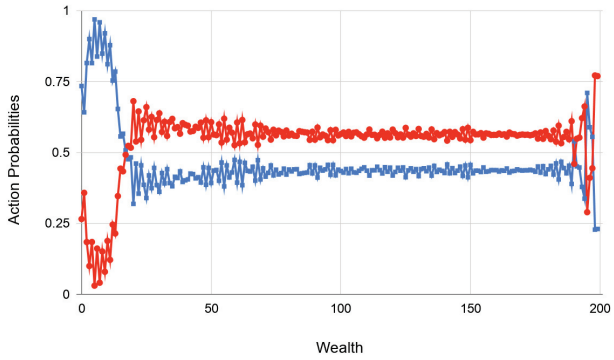


Figure 3: Action probabilities learned by RAlph for each wealth level of the RW benchmark. Blue line with boxes - safe action; red line with circles - unsafe action.

Evaluation. We evaluate RAlph and RAMCP on four instances of the Hallway (called Hallway 1, 2, 3, 4) of dimensions 2x3, 3x5, 5x8, 5x5. The corresponding MDPs have state-spaces of sizes $|\mathcal{S}|$ equal to 20, 44, 1136, 6553600, respectively. For the random walk, we consider benchmarks with 50 and 200 wealth levels (i.e. states).

The test configuration was: CPU: Intel Xeon E5-2620 v2@2.1GHz (24 cores); 8GB heap size; Debian 8. A training phase of RAlph is executed on 23 parallel threads, evaluation is single-threaded. Both algorithms were evaluated over 1000 episodes, with a timeout of 1 hour per evaluation. [1]

Metrics. For both RAMCP and RAlph, we report the average payoff and risk. To account for bias caused by runs that ended in failure, we also consider payoff averaged over runs that avoided a failure state ("Succ avg payoff" in Table 1). We also measured the training time of RAlph and, for both algorithms, an average time per evaluation episode. We also use the *total node expansion* metric, tracking the number of search tree nodes created throughout the whole experiment on a given benchmark. For RAlph, this includes both training and evaluation; hence it is a relevant indicator of how much searching both methods require to produce the results.

Results. The results are summarized in Table 1. Even in smaller benchmarks, RAlph is much faster and makes up to two orders of magnitude less node expansions. This is because RAMCP lacks the knowledge that RAlph acquires during the training phase and thus RAMCP often keeps hitting walls or blunder in circles. Also, RAlph's risk-averse exploration improves the chance of finding promising paths. Although the learning is an advantage to RAlph, the total number of node expansions (including the learning phase) is much smaller than in RAMCP, which tends to construct large search trees. In Hallway 3, the average payoff of solutions found by RAMCP are inferior to those found by RAlph in approximately half of the time; and while the

[1] Implementation can be found at https://github.com/snurkabill/MasterThesis/releases/tag/AAAI_release

	Algo	Δ	sim	Avg payoff	Stdev payoff	Risk	Succ avg payoff	Succ stdev payoff	Training time[s]	Time per episode (avg)[ms]	Total node expansions
H 1	RAMCP	0	25	12.79	23.93	0.0	12.79	29.93	N/A	252.3	23,837,630
		0.1	25	30.78	34.77	0.082	35.76	31.74	N/A	174.7	16,219,205
		0.25	25	45.08	36.86	0.193	61.80	14.43	N/A	80.8	8,405,505
	RAlph	0	25	40	0.0	0.0	40	0	1.8	7.8	120,830
		0.1	25	46.78	25.80	0.094	53.70	14.95	1.2	5.9	96,748
		0.25	25	52.36	35.74	0.196	70	0.0	1.6	3.3	27,054
H 2	RAMCP	0	50	N/A	N/A	N/A	N/A	N/A	N/A	Timeout	N/A
		0.1	50	N/A	N/A	N/A	N/A	N/A	N/A	Timeout	N/A
		1	50	60.15	39.27	0.15	73.57	24.44	N/A	134,534	123,943,098
	RAlph	0	50	61.0	0	0.0	61.0	0.0	103	137	29,739,512
		0.1	50	65.75	26.11	0.075	72.28	12.79	65	90	18,317,217
		1	50	70.11	32.25	0.136	82.84	3.11	5	6	8,625,983
H 3	RAMCP	0	100	92.53	137.02	0.651	278.67	6.83	N/A	294	138,800,592
		0.1	100	93.18	137.24	0.649	279.19	6.90	N/A	287	145,421,231
		1	100	21.71	83.49	0.906	285.70	2.24	N/A	59	59,666,989
		0.1	**500**	161.84	142.50	0.411	280.62	5.92	N/A	1,582	651,055,909
	RAlph	0	100	281.169	5.02	0.0	281.169	5.02	16	108	8,069,542
		0.1	100	281.723	9.51	0.001	281.169	5.02	73	154	45,766,309
		1	100	280.00	20.80	0.005	281.46	2.72	16	26	42,912,980
		0.1	**50**	279.32	29.14	0.01	282.24	2.41	8	63	3,733,503
H 4	RAlph	0	50	1270.0	0.0	0.0	1270.0	0.0	631	903	107,845,003
		0.1	50	1311.11	149.57	0.062	1349.54	2.09	821	1,034	123,007,131
		1	50	1276.45	130.61	0.110	1307.06	11.10	26	36	53,565,317
RW 1	RAMCP	0.05	50	17.78	19.38	0.234	27.03	10.75	N/A	548	77,713,072
	RAlph	0.05	50	23.32	13.55	0.035	24.92	10.69	34	68	23,077,790
RW 2	RAMCP	0.05	50	N/A	N/A	N/A	N/A	N/A	N/A	Timeout	N/A
	RAlph	0.05	50	97.33	24.38	0.007	98.18	22.24	62	113	104,871,114

Table 1: Summary of the Hallway benchmark. Here H 1,...,4 correspond to Hallway 1,...,4, respectively. RW 1 is a random walk with 50 states, RW 2 with 200 states. Parameter sim denotes the number of simulations per step.

failure-avoiding runs of RAMCP perform similarly to those of RAlph, RAMCP is not able to consistently avoid failures and its risk is well above Δ (the same holds for the RW benchmark). The reason is that RAMCP is too slow to find a competitive solution in the given time limit. Enlarging the number of expanded nodes in every step (sim) of RAMCP is not sufficient to beat RAlph. On the other hand, changing sim from 100 to 50 in RAlph does not have a significant effect on solution quality. The results for Hallway 4 show that RAlph scales well for larger state spaces. RAMCP is omitted for Hallway 4, as each of its executions timed out.

Discussion. We observed an interesting connection between RAlph and AlphaZero. The behavior of RAlph with $\Delta = 1$ is close in nature to the behavior of AlphaZero. If RAlph is invoked with Δ small or zero, it explores the state space much faster (measured by node expansion count) than with $\Delta = 1$. The reason is that the risk-averse exploration of RAlph typically visits much smaller part of the state-space. Hence, in cases when risky paths are sub-optimal, RAlph may find a solution faster than algorithms ignoring the risk.

RAlph also exhibited interesting behavior on the Hallway instance shown in Figure 2. For $\Delta = 0$, the only way to reach the gold is by exploiting the move perturbations: since the robot cannot to move east from C without risking a shift to the trap, it must keep circling through A, B, C, D until it is randomly shifted to E. RAlph is able, with some parameter tuning, to find this policy.

In the random walk benchmark, RAlph finds a commonsense solution of playing the safe action when the wealth is low and the riskier one otherwise. Figure 3 depicts the probabilities of choosing the respective actions at all wealth levels (up to the level $L = 200$). The differences of the probabilities for larger levels (≥ 50) are due to the step penalty equal to -1. For a larger penalty, the difference would be larger as the agent would be motivated to reach the top level L faster. The wiggliness for wealths close to L is caused by the specific structure of the optimal strategy. Indeed, for some specific wealth values close to L it is beneficial to take the safer action, and RAlph exploits this peculiarity.

5 Conclusions & Future Work

We introduced RAlph, an online algorithm for risk-constrained MDPs. Our experiments show that even with a simple predictor, RAlph performs and scales significantly

better than a state-of-the-art algorithm. As an interesting future work we see extension of the method to POMDPs and incorporation of more sophisticated predictors.

Acknowledgements

Krishnendu Chatterjee is supported by the Austrian Science Fund (FWF) NFN Grant No. S11407-N23 (RiSE/SHiNE), and COST Action GAMENET. Tomáš Brázdil is supported by the Grant Agency of Masaryk University grant no. MUNI/G/0739/2017 and by the Czech Science Foundation grant No. 18-11193S. Petr Novotný and Jiří Vahala are supported by the Czech Science Foundation grant No. GJ19-15134Y.

References

Altman, E. 1999. *Constrained Markov decision processes*, volume 7. CRC Press.

Ayton, B. J., and Williams, B. C. 2018. Vulcan: A monte carlo algorithm for large chance constrained mdps with risk bounding functions. *CoRR* abs/1809.01220.

Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. Cambridge, Massachusetts: The MIT Press.

Baumgartner, P.; Thiébaux, S.; and Trevizan, F. W. 2018. Heuristic search planning with multi-objective probabilistic LTL constraints. In *KR 2018*, 415–424. AAAI Press.

Bruyère, V.; Filiot, E.; Randour, M.; and Raskin, J.-F. 2014. Meet Your Expectations With Guarantees: Beyond Worst-Case Synthesis in Quantitative Games. In Mayr, E. W., and Portier, N., eds., *STACS*, volume 25 of *LIPIcs*, 199–213. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Chatterjee, K.; Novotný, P.; Pérez, G. A.; Raskin, J.; and Zikelic, D. 2017. Optimizing expectation with guarantees in POMDPs. In *AAAI 2017*, 3725–3732. AAAI Press.

Chatterjee, K.; Elgyütt, A.; Novotný, P.; and Rouillé, O. 2018. Expectation optimization with probabilistic guarantees in pomdps with discounted-sum objectives. In *IJCAI 2018*, 4692–4699.

Chatterjee, K.; Komárková, Z.; and Kretínský, J. 2015. Unifying Two Views on Multiple Mean-Payoff Objectives in Markov Decision Processes. In *LICS*, 244–256. IEEE Computer Society.

Filar, J., and Vrieze, K. 1997. *Competitive Markov Decision Processes*. Springer-Verlag.

Hou, P.; Yeoh, W.; and Varakantham, P. 2016. Solving Risk-Sensitive POMDPs With and Without Cost Observations. In *AAAI 2016*, 3138–3144. AAAI Press.

Howard, H. 1960. *Dynamic Programming and Markov Processes*. MIT Press.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Kocsis, L., and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *ECML*, volume 4212 of *LNCS*, 282–293. Springer.

Kress-Gazit, H.; Fainekos, G. E.; and Pappas, G. J. 2009. Temporal-Logic-Based Reactive Mission and Motion Planning. *IEEE Transactions on Robotics* 25(6):1370–1381.

Pineau, J.; Gordon, G.; Thrun, S.; et al. 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI*, volume 3, 1025–1032.

Poupart, P.; Malhotra, A.; Pei, P.; Kim, K.; Goh, B.; and Bowling, M. 2015. Approximate Linear Programming for Constrained Partially Observable Markov Decision Processes. In *AAAI 2015.*, 3342–3348. AAAI Press.

Puterman, M. 1994. *Markov Decision Processes*. Wiley.

Randour, M.; Raskin, J.-F.; and Sankur, O. 2015. Variations on the Stochastic Shortest Path Problem. In *VMCAI*, volume 8931 of *LNCS*, 1–18. Springer.

Rossman, L. A. 1977. Reliability-constrained dynamic programing and randomized release rules in reservoir management. *Water Resources Research* 13(2):247–255.

Russell, S. J., and Norvig, P. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Santana, P.; Thiébaux, S.; and Williams, B. C. 2016. RAO*: An Algorithm for Chance-Constrained POMDP's. In *AAAI 2016*, 3308–3314. AAAI Press.

Silver, D., and Veness, J. 2010. Monte-Carlo planning in large POMDPs. In *NIPS 23*. Curran Associates, Inc. 2164–2172.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419):1140–1144.

Smith, T., and Simmons, R. 2004. Heuristic search value iteration for POMDPs. In *UAI*, 520–527. AUAI Press.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. MIT press.

Teichteil-Königsbuch, F. 2012. Path-constrained markov decision processes: bridging the gap between probabilistic model-checking and decision-theoretic planning. In *ECAI 2012*, 744–749. IOS Press.

Undurti, A., and How, J. P. 2010. An online algorithm for constrained POMDPs. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, 3966–3973. IEEE.