Reinforcement Learning of Risk-Constrained Policies in Markov Decision Processes

Tomáš Brázdil,1 Krishnendu Chatterjee,2 Petr Novotný,1 Jiří Vahala1
1Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbrazdil, petr.novotny, xvahala1}@fi.muni.cz
2Institute of Science and Technology Austria, Klosterneuburg, Austria
Krishnendu.Chatterjee@ist.ac.at

Abstract
Markov decision processes (MDPs) are the de facto 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 106 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, Thiebaux, 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, Thiebaux, 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 (Undurdi 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 RAPlp 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, Komárová, 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 $\mathcal{M} = (S, A, \delta, rew, s_0, \gamma)$ where $S$ is a set of states, $A$ is a set of actions, $\delta : S \times A \rightarrow D(S)$ is a probabilistic transition function that given a state $s \in S$ and an action $a \in A$ gives the probability distribution over the successor states, $rew : S \times A \rightarrow \mathbb{R}$ 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 $\pi$, 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, $\pi$ selects an action $A_i \in A$ 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 $s, a, h, \ldots$ 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 $P^\pi(E)$ the probability of an event $E$ under policy $\pi$, and by $E^\pi[X]$ the expected value of a random variable $X$ under $\pi$.

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

Risk-Constrained Optimization. To encompass the notion of an undesirable event, we equip each MDP $\mathcal{M} = (S, A, \delta, rew, s_0, \gamma)$ with a set $F_M \subseteq S$ of failure states. A risk of a policy $\pi$ is then the probability that a failure state is encountered: $Risk(\pi) = P^\pi_s \left( \bigcup_{i=0}^{\infty} \{ S_i \in F_M \} \right)$. We assume that each $s \in F_M$ is a sink, i.e. $\delta(s | s, a) = 1$ and...
\( \text{rew}(s,a) = 0 \) 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 \( \pi \) which maximizes \( \text{Payoff}(\pi) \) subject to the constraint that \( \text{Risk}(\pi) \leq \Delta \). If there is no feasible policy, i.e. a policy s.t. \( \text{Risk}(\pi) \leq \Delta \), 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 \( \theta \)-parameterized function \( f_\theta : S \rightarrow \mathbb{R} \times [0,1]^{|\mathcal{A}|} \) assigning to each state \( s \) the tuple \( f_\theta(s) = (v, r, p) \) which predicts the parameters of some policy \( \pi \): \( v \) is the expected predicted payoff of \( \pi \) from \( s \), \( r \) is the predicted risk of \( \pi \) from \( s \), and \( p \) 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 \( \text{RAlph-episode} \) 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 \( \text{Train} \), 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 \( \text{RAlph-episode} \) (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 \( T \) by iterating the usual 4-phase MCTS simulations (node selection, tree expansion, leaf evaluation, backpropagation, see procedure \( \text{Simulate} \)). We follow the spirit of (Silver et al. 2018) and use the predictor \( f_\theta \) 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.**

```plaintext
1 procedure RAlph-train
   Input: MDP \( \mathcal{M} \) (with a horizon \( H \)), risk bound \( \Delta \),
   no. of training episodes \( m \), batch size \( n \)
   episodes ← 0; mod ← “train”; initialize \( f_\theta \);
   while episodes < m do
     batch ← 0; Data ← φ;
     while batch < n do
       \( E \leftarrow \text{RAlph-episode}(\mathcal{M},H,f_\theta,\Delta,\text{mod}) \);
       batch ← batch + 1;
       episodes ← episodes + 1;
       Data ← Data \cup \{E\};
     \thetag ← \text{Train}(\theta, Data);
   \while true do \text{RAlph-episode}(\mathcal{M},H,f_\theta,\text{mod})
```

**Algorithm 2: Simulation procedure.**

```plaintext
1 procedure \text{Simulate}(h)
   Input: \( \mathcal{M} \) (with a horizon \( H \)), risk bound \( \Delta \),
   \( \mathcal{A} \) \( \text{natural}\text{.}
```

---

**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 \( \Delta \) (line 9; described below). The distribution \( \xi_t \) 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 \( T \), 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 \( h.a \), 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 \( f_\theta \) 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 \( f_\theta \) in \( last(h) \).
We also have the following derived attributes: \( h.V_{\min} = \min_{a \in A} h.V_a \); and \( h.V_{\max} = \max_{a \in A} h.V_a \). 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 \( \text{root}(T) \) the root of \( T \) and by \( \text{leaf}(T) \) the set of leafs of \( T \). Also, for a node \( h \) we denote by \( T(h) \) the sub-tree rooted in \( h \).

**Episode Sampling: Overall Structure.** In Algorithm 2, a single search tree \( T \) 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 \( T \) defined below (line 9). This gives us a distribution \( \xi_i \) over actions as well as a risk distribution \( \tau_i \) over the child nodes of the root node \( \nu \). Formally, \( \tau(v(b)) \) 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 \( r_i \). The risk threshold is then updated to be the formula on lines 16–18, where \( \text{alt}risk \) is the probability placed by the risk distribution on all the histories not consistent with the current history \( v(a_i,s_{i+1}) \). Finally, we prune away all parts of the tree not consistent with the current history and continue into the 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 \( a = \arg \max_{a \in A} \text{UCT}(h,a) \), where

\[
\text{UCT}(h,a) = \frac{h.V_a - h.V_{\min}}{h.V_{\max} - h.V_{\min}} + C \cdot h.p_n \cdot \sqrt{\frac{\ln(h.N)}{h.N_a + 1}}.
\]

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

Upon reaching a leaf node \( h \), we expand \( T \) 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 \( F_M \), in which case the risk and future payoff are both 0.

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

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

The LP \( L \) encodes a probabilistic flow induced by some policy (constraints (1)–(4), which we together denote by \( \text{Flow}(T) \)), \( x_h \) 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 \( \Delta \) (constraint (5)). Hence, the procedure \( \text{Solve-LP} \) returns an action distribution \( \xi_i \) s.t. \( \xi_i(a) = x_{\text{root}(T),a} \) for each \( a \in A \).

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 \( M = (S,A,\delta,\text{rew},s,\gamma) \) with \( S = \{s,t,u\} \) and \( A = \{a,b\} \) s.t. \( \delta(s,a) = \frac{1}{2}, \delta(u|s,b) = 1 \). The states \( t, u \) are sinks, \( F_M = \{t\} \), and \( \text{rew}(s,a) = 1 \) (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:

![Tree Diagram](image)

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 \( L \) asks to maximize \( x_{s,a} \cdot 1.95 + x_{s,t} \) under the following constraints: \( x_1 = 1 \),

\[
x_s = x_{s,a} + x_{s,b}, x_{s,a} = 0.5 \cdot x_{s,a}, x_{s,t} = 0.5 \cdot x_{s,a},
\]

\[
x_{s,u} = x_{s,b}, 0.4 \cdot x_{s,a} + x_{s,t} + 0.1 \cdot x_{s,u} \leq 0.6 \text{ (and all variables in } [0,1]) \]

**Risk Distribution.** The choice of actions according to \( \xi_i \) 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 \( M \) 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 \( r_0 = \frac{1}{3} \cdot 1 + \frac{2}{3} \cdot r_1 \), where \( r_1 \) is the probability of hitting a failure state after continuing the play from \( s \). To ensure that \( r_0 \leq \Delta \), we must ensure \( r_1 \leq 0.2 \). Hence, in the next step, \( \Delta \) must be updated to 0.2.

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

Figure 1: The Linear program \( L \).
Algorithm 2: The episode sampling of RAalph.

1. procedure RAalph-episode (M, H, fθ, Δ, mod)
2.   global T;
3.   initialize T to one node s₀; E ← empty sequence;
4.   for i ← 0 to H − 1 do
5.     ν ← root(T); s₁ ← last(ν);
6.     repeat
7.        Simulate (M, H − i, T); // build T
8.        until timeout;
9.     ξ, τi ← Solve-LP(T, Δ);
10.    if mode = "train" then
11.       [ξi, RiskAwareExplore(T, ξi)]
12.      a₁ ← sample from ξi;
13.      s₁+i ← sample from δ(s₁, a₁);
14.     append (s₁, ξi, s₁+i) to E;
15.     alt ← {ν' ∈ T | ν' child of ν s.t. ν' ≠ νa₁s₁+i};
16.     altrisk ← ∑ν′∈alt τi(ν′);
17.     Δ ← (Δ − altrisk)/τi(νa₁s₁+i);
18.     T ← sub-tree of T rooted in νa₁s₁+i
19. return E
20. procedure Simulate(M, steps, T)
21.   [h ← root(T); depth ← 0;
22.   while h is not a leaf of T do
23.     a ← arg maxₐ∈A UCT(h, a);
24.     s ← sample from δ(last(h), a);
25.     h ← has; depth ← depth + 1;
26.     if last(h) ∉ FM then h.r ← 1;
27.   else h.r ← 0;
28.   return hbt]
29.   [initialize a new leaf hbt, add it to T as a child of h ;
30.     Predict (hbt)]
31.   else if last(h) ∈ FM then h.r ← 1;
32.   else h.r ← 0;
33.   val ← h.v; h.N ← h.N + 1;
34.   while h ≠ root(T) do
35.     let h = hbt where b ∈ A, t ∈ S;
36.     h′.N ← h′.N + 1; h′.N₀ ← h′.N₀ + 1;
37.     val ← rew(last(hbt), b) + γ · val;
38.     h′.V₀ ← h′.V₀ + (val − h′.V₀)/h′.N₀;
39.     h ← h′
40. procedure Predict (h, fθ)
41.   [(h.v, h.r, (h.pq₁)a∈A) ← fθ(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 fθ). Formally, we set τi(h) to be the optimal value of a linear program Lrisk(h) with constraints Flow(T(h)) and with the objective to minimize ∑h′∈leaf(T(h)) xh′ ⋅ h′.r.

Infeasible LP. The linear program 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 Lrisk(root(T)). (Note that Lrisk(h) is feasible for each node h.) We then again solve 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 ξj); or to explore, in which case we modify ξj in a way depending on whether the computation of ξj required just one call of the linear solver (i.e. if L was feasible without relaxing Δ) or not.

If L was feasible on the first try, we perturb ξj 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 ξj might be too risky, which is indicated by violating the risk constraint ∑b∈A,τ∈S τi(root(T)hb) ⋅ ξj(root(T)hb) ≤ Δ. 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 ξj; such a distribution is then output by RiskAwareExplore.

If we needed to relax Δ to solve L, we assume the predictions to be too pessimistic and opt for a more radical exploration. Hence, we ignore L altogether and instead select actions proportionally to their UCT values, i.e. we put ξi(a) = UCT(root(T), a)/∑b∈A UCT(root(T), b).

3.2 Predictor & Training

In principle any predictor (e.g. a neural net) can be used with RAalph. 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. θ(s) = fθ(s)). Each episode produces a data element η = (s₀, ξ₀, ρ₀) ⋯ (sH−1, ξH−1, ρH−1), where sᵢ, ξᵢ, ρᵢ 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 Gᵢη = ∑j=₁ H−1 γ̂j−1 · ηj from that step on; similarly, for risk we set Pᵢη to 1 if some sⱼ ∈ FM for j ≥ i, and to 0 otherwise; for action probabilities we denote Pᵢη = ξᵢ. For each state s encountered on η we put Iᵦ(s) = {i | 0 ≤ i ≤ H − 1 ∧ sᵢ = 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) = ∑η∈Data Iᵦ(s) (the total visit count of s), G(s) = ∑η∈Data, i∈Iᵦ(s) Gᵢη, R(s) =
common code with RAlph as possible. In particular, both al-
comparison, our implementation of RAMCP shares as much
risk-averse exploration) on the performance. To get a fair
effect of RAlph’s crucial features (learning and prediction,
modified (per suggestion in the source paper) so as to al-
line RAMCP algorithm from (Chatterjee et al. 2018) slightly

\[ \sum_{\eta \in \text{Data}, i \in I_{\text{data}}(s)} R^i_{\eta}, \text{ and } P(s) = \sum_{\eta \in \text{Data}, i \in I_{\text{data}}(s)} P^i_{\eta}. \]

These are then averaged to \( \bar{G}(s) = G(s)/N(s), \) \( \bar{R}(s) = R(s)/N(s), \) and \( \bar{P}(s) = P(s)/N(s) \) (operations on probability distributions are componentwise). Together, these averages form a target table \( \theta \) such that \( \tilde{\theta}(s) = \left( \bar{G}(s), \bar{R}(s), \bar{P}(s) \right). \) Finally, we perform the update \( \theta \leftarrow \theta + \alpha(\tilde{\theta} - \theta) \), where \( \alpha \) is a pre-set learning rate.

This scheme can be generalized to more sophisticated pre-
dictors, 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, per-
factly observable version of Hallway (Pineau et al. 2003;
Smith and Simmons 2004) where we control a robot nav-
igating 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 posi-
tive 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 ran-
dom 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 on-
line RAMCP algorithm from (Chatterjee et al. 2018) slightly
modified (per suggestion in the source paper) so as to al-
low 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 al-
gorithms employ UCT-like simulations. We denote by \( \text{sim} \)
the number of these simulations invoked per decision.

**Evaluation.** We evaluate RAlph and RAMCP on four in-
stances of the Hallway (called Hallway 1, 2, 3, 4) of dimen-
sions 2x3, 3x5, 5x8, 5x5. The corresponding MDPs have
state-spaces of sizes \(|S|\) 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, evalua-
tion 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 aver-
age 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 train-
ing 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 hits-
walls or blunder in circles. Also, RAlph’s risk-averse explo-
ration 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 solu-
sions 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
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.

<table>
<thead>
<tr>
<th></th>
<th>Algo</th>
<th>( \Delta )</th>
<th>( \sim )</th>
<th>Avg payoff</th>
<th>Stdev payoff</th>
<th>Risk</th>
<th>Succ avg payoff</th>
<th>Succ stdev payoff</th>
<th>Training time[s]</th>
<th>Time per episode (avg)[ms]</th>
<th>Total node expansions</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1</td>
<td>RAMCP</td>
<td>0</td>
<td>25</td>
<td>12.79</td>
<td>23.93</td>
<td>0.0</td>
<td>12.79</td>
<td>29.93</td>
<td>N/A</td>
<td>252.3</td>
<td>23,837,630</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>25</td>
<td>30.78</td>
<td>34.77</td>
<td>0.082</td>
<td>35.76</td>
<td>31.74</td>
<td>N/A</td>
<td>174.7</td>
<td>16,219,205</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.25</td>
<td>25</td>
<td>45.08</td>
<td>36.86</td>
<td>0.193</td>
<td>61.80</td>
<td>14.43</td>
<td>N/A</td>
<td>80.8</td>
<td>8,405,505</td>
</tr>
<tr>
<td></td>
<td>RAlph</td>
<td>0</td>
<td>25</td>
<td>40.00</td>
<td>0.00</td>
<td>0.0</td>
<td>40.00</td>
<td>0.00</td>
<td>1.8</td>
<td>7.8</td>
<td>120,830</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>25</td>
<td>46.78</td>
<td>25.80</td>
<td>0.094</td>
<td>55.70</td>
<td>14.95</td>
<td>1.2</td>
<td>5.9</td>
<td>96,748</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.25</td>
<td>25</td>
<td>52.36</td>
<td>35.74</td>
<td>0.196</td>
<td>70.00</td>
<td>0.00</td>
<td>1.6</td>
<td>3.3</td>
<td>27,054</td>
</tr>
<tr>
<td>H 2</td>
<td>RAMCP</td>
<td>0</td>
<td>50</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Timeout</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>50</td>
<td>60.15</td>
<td>39.27</td>
<td>0.15</td>
<td>73.57</td>
<td>24.44</td>
<td>N/A</td>
<td>134,534</td>
<td>123,943,098</td>
</tr>
<tr>
<td></td>
<td>RAlph</td>
<td>0</td>
<td>50</td>
<td>61.00</td>
<td>0.00</td>
<td>0.0</td>
<td>61.00</td>
<td>0.00</td>
<td>103</td>
<td>137</td>
<td>29,739,512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>50</td>
<td>65.75</td>
<td>26.11</td>
<td>0.075</td>
<td>72.28</td>
<td>12.79</td>
<td>65</td>
<td>90</td>
<td>18,317,217</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>50</td>
<td>70.11</td>
<td>32.25</td>
<td>0.136</td>
<td>82.84</td>
<td>3.11</td>
<td>5</td>
<td>6</td>
<td>8,625,983</td>
</tr>
<tr>
<td>H 3</td>
<td>RAMCP</td>
<td>0</td>
<td>100</td>
<td>92.53</td>
<td>137.02</td>
<td>0.651</td>
<td>278.67</td>
<td>6.83</td>
<td>N/A</td>
<td>294</td>
<td>138,800,592</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>100</td>
<td>93.18</td>
<td>137.24</td>
<td>0.649</td>
<td>279.19</td>
<td>6.90</td>
<td>N/A</td>
<td>287</td>
<td>145,421,231</td>
</tr>
<tr>
<td></td>
<td>RAlph</td>
<td>0</td>
<td>100</td>
<td>21.71</td>
<td>83.49</td>
<td>0.906</td>
<td>285.70</td>
<td>2.24</td>
<td>N/A</td>
<td>59</td>
<td>39,666,989</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>100</td>
<td>281.7123</td>
<td>9.51</td>
<td>0.001</td>
<td>281.169</td>
<td>5.02</td>
<td>16</td>
<td>108</td>
<td>8,069,542</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>100</td>
<td>280.00</td>
<td>20.80</td>
<td>0.005</td>
<td>281.46</td>
<td>2.72</td>
<td>16</td>
<td>26</td>
<td>42,912,980</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>50</td>
<td>279.32</td>
<td>29.14</td>
<td>0.01</td>
<td>282.24</td>
<td>2.41</td>
<td>8</td>
<td>63</td>
<td>3,733,503</td>
</tr>
<tr>
<td>H 4</td>
<td>RAlph</td>
<td>0</td>
<td>50</td>
<td>1270.0</td>
<td>0.00</td>
<td>0.0</td>
<td>1270.0</td>
<td>0.00</td>
<td>631</td>
<td>903</td>
<td>107,845,003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.1</td>
<td>50</td>
<td>1311.11</td>
<td>149.57</td>
<td>0.062</td>
<td>1349.54</td>
<td>2.09</td>
<td>821</td>
<td>1,034</td>
<td>123,007,131</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>50</td>
<td>1127.45</td>
<td>130.61</td>
<td>0.110</td>
<td>1307.06</td>
<td>11.10</td>
<td>26</td>
<td>36</td>
<td>53,565,317</td>
</tr>
<tr>
<td>RW 1</td>
<td>RAMCP</td>
<td>0.05</td>
<td>50</td>
<td>17.78</td>
<td>19.38</td>
<td>0.234</td>
<td>27.03</td>
<td>10.75</td>
<td>N/A</td>
<td>548</td>
<td>77,713,072</td>
</tr>
<tr>
<td></td>
<td>RAlph</td>
<td>0.05</td>
<td>50</td>
<td>23.32</td>
<td>13.55</td>
<td>0.035</td>
<td>24.92</td>
<td>10.69</td>
<td>34</td>
<td>68</td>
<td>23,077,790</td>
</tr>
<tr>
<td>RW 2</td>
<td>RAMCP</td>
<td>0.05</td>
<td>50</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Timeout</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>RAlph</td>
<td>0.05</td>
<td>50</td>
<td>97.33</td>
<td>24.38</td>
<td>0.007</td>
<td>98.18</td>
<td>22.24</td>
<td>62</td>
<td>113</td>
<td>104,871,114</td>
</tr>
</tbody>
</table>

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 small \( \Delta \) 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 (\( \geq 50 \)) 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


