Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling

# Risk-Variant Policy Switching to Exceed Reward Thresholds

**Breelyn Kane** and **Reid Simmons**
Robotics Institute
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
{breelynk,reids}@cs.cmu.edu

**Abstract**

This paper presents a decision-theoretic planning approach for probabilistic environments where the agent's goal is to win, which we model as maximizing the probability of being above a given reward threshold. In competitive domains, second is as good as last, and it is often desirable to take risks if one is in danger of losing, even if the risk does not pay off very often. Our algorithm maximizes the probability of being above a particular reward threshold by dynamically switching between a suite of policies, each of which encodes a different level of risk. This method does not explicitly encode time or reward into the state space, and decides when to switch between policies during each execution step. We compare a risk-neutral policy to switching among different risk-sensitive policies, and show that our approach improves the agent's probability of winning.

## 1 Introduction

Many probabilistic planners seek to maximize expected reward, and do little to incorporate the variance of the reward distribution when developing a plan for an agent. Therefore, many planners assume the agent has a risk-neutral attitude. In competitions, however, one often sees people behave differently (e.g., take more risks) when they believe they may end up losing. For instance, a sports team may play more aggressively when losing, but more defensively when trying to maintain a lead. This reflects the idea that it does not matter by how much one wins or loses, as long as the score is in the agent's favor. We model this as maximizing the probability of being above a given reward threshold (e.g. a competitor's current top score).

An agent needs to adjust its risk attitudes dynamically to exceed a threshold and win. For example, a campaign manager may appeal to particular advocacy groups or change the tone of the candidate's speech based on the candidate's position in the polls. In hockey, if a team is losing, they often remove the goalie in hopes that having an additional offensive player will increase the chances of a tying goal. This strategy is risky since it increases the chances of losing by more goals. If the hockey team were just trying to maximize its expected goal differential over the season it may never chose to remove the goalie. Intuitively, the idea of an agent needing to be more aggressive or conservative while executing a policy parallels a human's risk-seeking and risk-averse preferences.

A straightforward approach is to encode cumulative reward explicitly in the state space of a standard Markov Decision Process (MDP). Doing so, however, explodes the state space since the number of different cumulative reward values is typically very large. We propose an alternate approach in which multiple policies are generated offline. Then, an online algorithm decides which policy to use based on which is more likely to achieve the threshold constraint.

The algorithm reasons about the complete distribution of rewards, not just mean and variance, to make fine-grained decisions about which policy is most applicable for a given situation. In particular, we provide an algorithm that decides when to switch between strategies, at run-time, by estimating non-parametric distributions of the reward functions for each of these policies. Agents with risk-sensitive policies to choose from now have the ability to switch to a policy with a higher variance in hopes of increasing their chances of meeting the given threshold. We show that by switching policies in such a manner, the agent will end up doing better (with respect to the goal of finishing above a certain threshold of reward) than if it just followed the risk-neutral policy.

## 2 Background

### 2.1 Markov Decision Processes

Initially, our algorithm requires a model of the environment. We formulate this model as a Markov Decision Process, a mathematical representation of a sequential decision problem. An MDP is a four-tuple $\{S, A, T(S,A), r(S,A)\}$ consisting of:

- $S$ : set of states $\{s_0 \dots s_\infty\}$
- $A$ : set of actions $\{a_0 \dots a_\infty\}$
- $T(s_t, a_t)$ : transition function yielding $s_{t+1}$, with $P(s_{t+1}|s_t, a_t)$
- $r(s_t, a_t)$ : immediate reward function

Solving an MDP produces a policy, $\pi$, that maps states to actions $\pi : S \rightarrow A$. One approach is to use *value-iteration* to find a policy using the value-update rule below. This value function is also used to estimate the distribution of future discounted reward, as described in section 4.2.

$$V^{\pi}(s) = \max_{a \in A}\{R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V(s')\} \qquad (1)$$

## 2.2 Utility and Risk

Incorporating risk attitudes captures the trade-off between variance and mean in the distribution of reward outcomes. For instance, risk-seeking policies tend to have a lower mean and larger variance (more upside, but also more downside) than risk-neutral policies. Utility theory provides structure for making decisions of varying risk attitudes (Pratt 1964). A utility function maps an agent's value to a plan of wealth that represents the agent's rational choice. Linear utility functions maximize expected cumulative reward and represent risk-neutral decision makers, while exponential functions model risk-sensitive decisions. Concave utility functions reflect risk-averse decisions, and convex utility functions characterize risk-seeking decisions. The convexity of these functions changes for different risk factors.

When an agent's utility function is constant for the function duration (such as linear or exponential) the risk measure is constant, and this is known as a zero-switch utility function. Zero-switch utility functions are unrealistic, since decisions often change as wealth level changes. (Liu and Koenig 2008) take this a step further in defining MDPs that have a one-switch utility function. In the one-switch case, an agent acting risk-neutral may switch to being more conservative, which entails one switch from a linear function to a concave exponential function. While this is closer to realistic decision making, it seems more natural to allow the agent to switch multiple times between many utility functions, which is what our approach supports.

## 3 Related Work

While previous work has investigated switching between strategies (policies) to achieve different subgoals (Comanici and Precup 2010), our work instead considers adapting a strategy with the assertion of risk for a single goal – that of winning. In defining what it might mean to win, other works have discussed the idea of using thresholded reward objective functions (McMillen and Veloso 2007), (Wagman and Conitzer 2008). Our work differs by not requiring an adversary, by focusing on the use of risk attitudes, not requiring a threshold to be known ahead of time, and by having the ability to switch strategies during run-time. These works focus on solving variants of the MDP's objective function, and produce a single static policy. For instance, in (Geibel and Wysotzki 2005), risk is incorporated into the estimated objective function as a weighting factor.

By not focusing on altering the MDP's objective function, our work also trades off computation at execution time for creating policies more efficiently during planning time. This tradeoff was also a goal of work done by (Roth, Simmons, and Veloso 2005) in limiting communication for distributed agents modeled as DEC-POMDPs.

Another distinguishing characteristic of our work is that we reason about the complete distribution of a policy's reward, rather than just the expectation: in particular the reward is modeled as a non-parametric distribution. Other work that estimates the variance of an MDP (Tetreault, Bohus, and Litman 2007) does so by adding uncertainty to the MDP model's parameters. This is done by modeling the transition probabilities as a Dirichlet distribution and mapping confidence intervals over transition counts. Our approach better handles the tails of the distribution, which is very important for distinguishing the effects of different risk-sensitive policies.

## 4 Run-Time Policy Switching

A utility function that maximizes the probability of being over a given threshold, while representing the entire domain, is difficult to know ahead of time. This paper assumes that function is unknown, and emulates a multi-switch utility function by deciding which policy to follow (corresponding to different risk attitudes) during run-time. To accomplish this, we generate a suite of policies, including a risk-neutral policy (linear utility function), and risk sensitive policies (see section 4.1). A risk factor, $\delta$, controls the amount of convexity for the exponential utility functions.

Then, for each policy, we estimate the complete reward distribution. This is done by executing (offline) a sufficient number of trajectories in the original MDP and collecting statistics on the cumulative rewards achieved at each state. The distribution of rewards is then modeled as a Cumulative Distribution Function (CDF) (see section 4.2). Finally, at each step during run-time, the agent determines which policy has the highest probability of exceeding the (user specified) reward threshold, given the current state and cumulative reward, so far. The agent then picks the action associated with the current state for that policy, executes it, then determines again which policy to use (see section 4.3).

### 4.1 Creating Policies

Creating a suite of policies that allow a variety of strategies for the agent to employ, requires a model of the world in the form of an MDP. While different techniques may be used to generate the various risk-sensitive policies, we use the transformation algorithm described in (Koenig and Simmons 1994) and (Liu 2005). This transformation does not affect the state space, but merely changes the structure of the MDP to choose actions based on probabilities that now form an exponential utility rather than a linear utility. The reason exponential utility functions are used is because they maintain the Markov property, preserve the decomposability of planning functions, and they are one of the most common risk functions. Convex utility functions (Figure 1a) are of the form:

$$U(r) = \delta^r, \delta > 1$$

and concave functions (Figure 1b) are of the form:

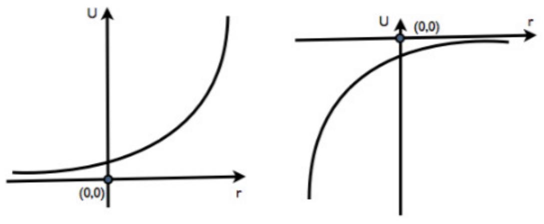
$$U(r) = -\delta^r, 0 < \delta < 1.$$

where $\delta$ is the risk factor, and $r$ is a reward. (Liu 2005) further simplifies the function as :

$$U_{exp}(r) = \iota\delta^r,$$

where

$$\iota = sgn\ ln\ \delta.$$



(a) Convex (risk seeking) (b) Concave (risk averse)

Figure 1: Convex and concave exponential utility functions

To summarize the approach described in (Koenig and Simmons 1994), the transition probabilities of the original MDP are transformed by multiplying them by a function of risk and the immediate rewards are also transformed. Specifically, all non-goal state transition probabilities are converted to $P(s'|s,a)\delta^{r(s,a)}$. The original probabilities add up to one, but transforming the probabilities cause them to decrease depending on the risk factor. Since the transformed probabilities no longer add up to one, an additional sink state is introduced where the probability is $1 - \sum_{s' \in S} P(s'|s,a)\delta^{r(s,a)}$. The larger the risk factor, the greater chance the agent has of falling into this sink state, and the more it is encouraged to take riskier actions. Therefore, increasing the risk parameter, $\delta$, generates policies that select increasingly riskier actions. Models with mixed positive and negative rewards, known as arbitrary rewards, require certain properties to hold, described in (Liu 2005). Positive rewards cause the initial MDP probabilities to be scaled to a value less than or equal to the reward, which might be greater than one, so arbitrary rewards are transformed to [0,1].

## 4.2 Estimating the Reward Distribution

The next step is to empirically estimate the (non-parametric) distribution of reward for every state of each policy. We do this by executing the policies in the original MDP. Each trajectory run will result in a cumulative discounted reward value. These values make up the distribution.

The cumulative discounted reward is given by:

$$R(s) = \sum_{i=0}^{\infty} \gamma^i r(s_i, a_i) \tag{2}$$

where $\gamma$ is the discount factor.

More explicitly:
$a = \pi(s)$
$R(s) = \gamma^0 r(s_0, a_0) + \gamma \sum_{i=0}^{\infty} \gamma^i r(s_i, a_i)$
$R(s) = r(s_0, a_0) + \gamma(r(s_1, a_1) + \gamma(r(s_2, a_2) + \ldots))$
where T(s,a) is the transition function for generating all of the next states:

$$R(s) = r(s,a) + \gamma(R(T(s,a))) \tag{3}$$

Note that the discount factor decreases the contribution of the future reward term over time. Therefore, there is a point where the discount factor causes the future reward to be arbitrarily small $\gamma^{Tr} * maxR \leq \epsilon_1$, where $maxR$ is the maximum possible immediate reward and $\epsilon_1$ is a small constant. Trajectory length ($Tr$), or required number of time-steps, is then calculated as:

$$Tr = \frac{\log(\epsilon_1) - \log(maxR)}{\log(\gamma)} \tag{4}$$

A trajectory is a sequence of state and action transitions $s_0 \rightarrow^{a_1} s_1 \rightarrow^{a_2} s_2$ generated by following the known policy in the environment, and may visit the same state multiple times. The stopping criteria, for how many trajectories to run, is based on the convergence of a fourth-order statistic. The statistic needs to be scale invariant, since our approach is domain-independent. Convergence occurs when the variance of the distribution variance divided by the mean of the variance is less than some small value, $\epsilon_2$. This statistic states that convergence occurs when the spread between numbers in sample distributions (obtained from the overall distribution) is arbitrarily small, and then this is scaled by the mean.

After collecting the distribution of values for each policy, we convert them into the corresponding Cumulative Distribution Function (CDF).

$$F(x) = \int_{-\infty}^{x} f(t)\, \mathrm{d}t = P(V \leq x) \tag{5}$$

The CDF, $F(x)$, gives the probability of achieving reward less-than-or-equal to some cumulative discounted reward, $x$. Note that for each policy we need a separate CDF for every state. Example CDFs are shown in Figure 4 and pseudocode for estimating the reward function is presented in Algorithm 1. For every state, the trajectory length is calculated by equation (4). Then, while the statistic has yet to converge, lines 7-10 go through an entire trajectory sequence saving off the immediate reward that corresponds with each state. The next *for* loop is used to discount all the states' corresponding reward values at once.

To increase efficiency, we simultaneously collect rewards for every state visited along a trajectory. In particular, the cumulative value of $s_t$ is $R(s_t)$ as given in equation (2). Trajectories can visit the same state multiple times, so one trajectory run may collect multiple values for that state. We run trajectories of length $2 * Tr$ (line 6 of Algorithm 1), but do not include values of states $s_t$ where $t > Tr$. Even with having to run each trajectory twice as long, collecting multiple values per trajectory is a huge win in our experiments, at least an order of magnitude faster.

## 4.3 The Switching Criteria

It is straightforward to calculate the maximum probability over a threshold using the CDF. The probability of being above a discrete threshold is a matter of subtracting the CDF from one.

$$1 - F(x) = \int_{x}^{\infty} f(t)\, \mathrm{d}t = P(V > x) \tag{6}$$

**Algorithm 1** generates reward distributions given a policy. For each starting state, a trajectory is run for the calculated trajectory size. The second nested for loop is used to discount the rewards by backtracking.

```
1:  for $i = 1 \to stateSz$ do
2:      $Tr = getTrajectorySize(\epsilon_1)$
3:      do
4:      $vals = getAllValsSavedForStartState(s_i)$
5:      $statistic = var(var(vals))/mean(var)$
6:      for $j = 1 \to 2 * Tr$ do
7:          $a_j = \pi(s_j)$
8:          $r = r(s_j, a_j)$
9:          $saveOff(r, s_j, 0, j)$
10:         $s_{j+1} = getNextState(T(s_j, a_j))$
11:     end for
12:
13:     //backtrack to discount the values
14:     $R = 0$
15:     for $x = savedOffValsSz \to 1$ do
16:         $r = getSavedOff(x).ImmedRew$
17:         $R = r + \gamma * R$
18:         if $x < Tr$ then
19:             $s = getSavedOff(x).s$
20:             $saveOff(r, s, R, x)$
21:         end if
22:     end for
23:     while($statistic > \epsilon$)
24: end for
```

We define $R^t(s)$ as the *running* cumulative discounted reward for time $t$, starting in state $s$.

$$R^t(s) = \sum_{i=0}^{t} \gamma^i r(s, a) \tag{7}$$

[Note that $R(s)$ in equation [2] is then equal to $R^\infty(s)$.]

Now, $P(V > x)$, becomes:

$$P(R(s_0) > thresh|\pi)$$

$$R(s_0) = \sum_{i=0}^{t-1} \gamma^i r(s, a) + \sum_{i=t}^{\infty} \gamma^t \gamma^{i-t} r(s_i, a_i)$$

$$R(s_0) = R^{t-1}(s_0) + \gamma^t R(s_t)$$

so to maximize the probability of being greater than threshold $thresh$, we have:

$$\max_\pi P(R^{t-1}(s_0) + \gamma^t R(s_t) > thresh|\pi)$$

$$\max_\pi P(R(s_t) > \frac{thresh - R^{t-1}(s_0)}{\gamma^t}|\pi)$$

$$valtofind = \frac{thresh - R^{t-1}(s_0)}{\gamma^t}$$

The pseudocode in Algorithm 2 details the process of selecting actions, by choosing the policy that maximizes the probability of being above *valtofind*. Note that, if the policies are equal, the algorithm defaults to following the risk-neutral policy. Also note that the CDF for each policy is not required to be smooth, and in some cases may resemble a step function. It is necessary to interpolate the values to retrieve the probability since the CDF is not continuous. This occurs on lines 7 and 8 (of Algorithm 2) in the $cdfGet$ function.

**Algorithm 2** executes actions starting in some initial state. It switches between selecting the actions from a set of policies based on a threshold.

```
1:  Given threshold $thresh$, start state $s_0$
2:  $Tr = getTrajectorySize(\epsilon)$
3:  $r_0 = 0$
4:  for $i = 0 \to Tr - 1$ do
5:      $r_i = r(s_i, a_i)$
6:      $cdfVal = (thresh - R_i)/\gamma^i$
7:      $max = 0$
8:      $\pi_{currBest} = \pi_{risk-neutral}$
9:      for $0 \to allcdfs$ do
10:         $cCurr = 1 - cdfGet(cdfVal, cdf.this)$
11:         if $cCurr > max$ then
12:             $max = cCurr$
13:             $\pi_{currBest} = \pi_{currCDF}$
14:         end if
15:     end for
16:     $a_{i+1} = \pi_{currBest}(s_i)$
17:     $R_{i+1} = R_i + \gamma^i * r_i$
18:     $s_{i+1} = getNextState(T(s_i, a_{i+1}))$
19: end for
```

## 4.4 Changing the Reward Threshold

Our formulation assumes that the reward threshold is given as an input. In some cases, threshold values can correlate to interpretations of the world, such as cut-off times for delivering items or the current high score of a video game. A threshold could also be a percentage line in the CDF, such that 60% of the time the distribution is better than some value. The threshold depends on the problem one is trying to solve. The algorithm does not care how the threshold is chosen or what it represents. The algorithm attempts to maximize the probability of being over the threshold, regardless.

Note, however, that as the threshold shifts to the lower end of the reward distribution, the agent chooses policies that are more risk-averse; as the threshold shifts to the other extreme, the agent chooses more aggressive policies. Depending on the uncertainty in the environment, it may need to switch to risky policies earlier, rather than later, or scale back to a less risky policy when it is performing well.

## 5 Evaluation

We have tested our algorithm in two domains: *Super Mario Bros* (described in section 5.1) and a simpler pizza delivery domain (described in section 5.4).

## 5.1 Mario Domain

The *Super Mario Bros* domain uses the Infinite Mario simulator [1]. Previous work using this domain (Mohan and Laird 2011) compared a learning agent to the agent provided with the simulator for one world. Our agent generalizes a small state space over many worlds.

Infinite Mario includes a trainer for generating episodes of varying difficulty and type. The agent (Mario) must move through the environment collecting objects, fighting monsters, all while getting to a goal line without being killed. The environment is made up of observations and actions. Mario receives the visual scene as a 16 x 22 array of tiles. Each tile contains a character or a number. If the tile is a character, it represents the tile type (coin, block, etc); if it is an integer, it indicates how Mario can travel through that tile. There is also an observation vector that includes the location and types of monsters in the environment. The primitive actions Mario can take correspond to buttons on a Nintendo game controller: direction, jump, and speed.

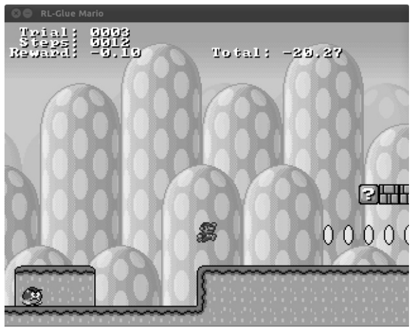


Figure 2: Mario world (screen capture from the Infinite Mario simulator)

The action space for our model is made up of nine macro-actions (see Table 1), inspired by Function Level Operators in (Mohan and Laird 2010). A macro-action tries to find a path to its object of interest using A*. The A* path goes around anything Mario is not able to travel through on his way to the object. The macro-action executes the A* path until the end condition is met, using the appropriate combination of primitive actions.

The state space for our MDP is based on Mario's relationship to objects in the environment. A symbolic state representation for Mario was presented in (Mohan and Laird 2010). Our state vector contains seven dimensions {mario, block, coin, mushroom, pipe, finish, goomba}. Each dimension takes on two possible values; either the object is near (in the visual scene) or far (not in the visual scene). The "mario" dimension indicates if Mario is big or small.

In Infinite Mario, each episode is generated based on a seed value. The rewards Mario receives are -0.01 for each time step, -10 for dying, 1 for collecting a coin or killing a monster, and 100 for reaching the finish line. We ran 1,000 trials over different starting episodes to estimate the transition probabilities and cumulative rewards for macro-actions.

[1]http://2009.rl-competition.org/mario.php

| Macro-actions | | |
|---|---|---|
| Action Name | Description | End Condition |
| grabCoin | go to nearest coin | past coin |
| grabCoinForever | go to nearest coin | no coins |
| avoidMonster | go past monster | past monster |
| tackleMonster | go to above monster (to smash) | past monster |
| tackleMonster Forever | go to above monster (to smash) | all monsters smashed |
| searchBlock | hit nearest question block | past block |
| searchBlock Forever | hit nearest question block | blocks searched |
| getMushroom | find hidden mushroom | past mushroom |
| moveSmart | use A-Star to move right 4 | past move position |

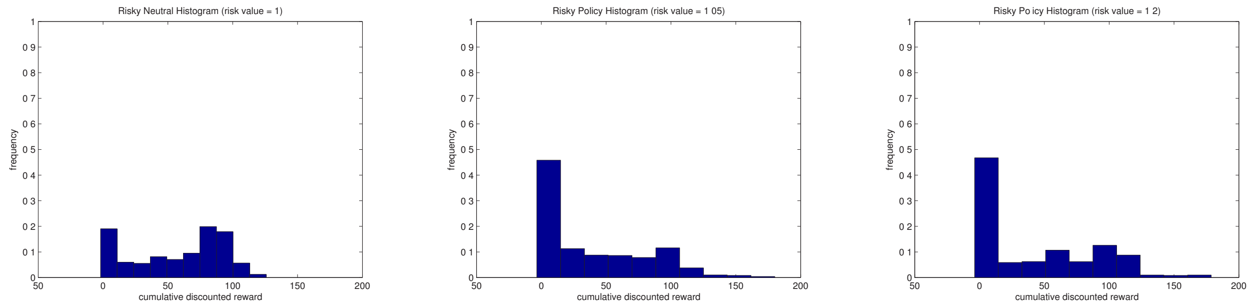
Table 1: Macro-actions of the MDP model

Mario chooses randomly from the set of macro-actions at each time step (biased to moving towards the finish line, in order to avoid getting stuck too often). While the macro-action is executing, immediate rewards are accumulated, and these become the "immediate" reward of the macro-action. Similarly, the state when the macro-action is started and the state when it completes are used to update the transition probability for that pair of states.

The MDP is solved using value-iteration, where the immediate reward and transition functions return values based on the information captured by sampling the Mario world. To generate each risky policy, the probability changes to $P(s'|s,a)\delta^{r(s,a)}$, where $P(s'|s,a)$ is the transition probability obtained from earlier testing, $\delta$ is a risk factor greater than one, and $r(s,a)$ is the average immediate reward for this state and action from the recorded rewards. The Mario world contains both negative and positive rewards, so the probability returned must be linearly transformed to a range between 0 and 1. In other words, the interval $[0, 1 * \delta^{maxReward}]$ needs to map to $[0, 1]$. Exponential functions maintain their convexity for affine transforms. The linear transformation for the probability mapping is just a mapping from $[A, B]$ to $[C, D]$ where $x' = ((D - C)/(B - A))x + C$.

The policies produced for the risk-seeking transformation tend to choose the *forever* macro-actions more (see Table 1). For these actions, there is a chance of getting a higher score by retrieving all the objects and an increased chance of dying, since Mario is in the environment longer. The *moveSmart* macro-action also is chosen more often in the risky policies. This action may be chosen more because it produces more variance in the environment than just going after a particular object. The histograms for various policies are displayed in Figure 3. Not that, as the risk value increases, the distributions have a larger variance but lower mean.

The reward functions for each state in the MDP are estimated according to the algorithm described in section 4.2.



(a) risk-neutral policy, risk value 1 (b) risk value 1.05 (c) risk value 1.2

Figure 3: Histograms for varying risk values

Two variations of the domain were used. The Mario world modeled as an MDP and the actual Infinite Mario simulator. The difference between the two can be thought of as an example where the model is well known and one where things may not be modeled perfectly.

Reward distributions are estimated for each state in these domains. For the Infinite Mario simulator, estimations are taken over multiple random seeds (different episodes) and, for the MDP model, domain trajectories are sampled based on the transition probabilities constructed previously. The number of samples needed depend on the convergence statistic described in section 4.2. CDFs are then constructed for all of these states. The CDF for various policies in start state 0 {mario small, block far, coin far, mushroom far, pipe far, finish far, goomba far} is shown for the MDP domain in Figure 4a and the actual Mario simulator in Figure 4b. The MDP modeled Mario is slightly more optimistic, partly because the model was constructed using the average immediate rewards collected in the environment. Also, it is possible the MDP assumes that there are more transitions to states with higher reward (such as more coins) than actually exist in the real environment.

## 5.2 Mario Results

Different threshold values affect how often the switching strategy chooses more risky actions. The rewards are discounted so the total reward received is lower than the exact values returned in the Mario simulator. First, the results are displayed for exploiting the policy in the modeled Mario MDP (Tables 2 and 3). Each group of results compares 1,000 trajectory runs, all with start state 0, using just the risk-neutral policy versus 1,000 runs using the switching strategy (switching between the risk-neutral policy and a risky policy with $\delta = 1.2$). In order to compare trajectories fairly, random probabilities are generated offline on a per-state basis. As the policy is being exploited in the original MDP, the probability that corresponds to the current state, and number of times visited, is retrieved. This probability is then the same for both trajectories, and is used to determine the next state.

Table 4 shows results in the actual Mario world for the risk-neutral policy and the switching strategy (switching between the risk-neutral policy and a risky policy with $\delta =$ 1.05). Each group of results compares runs over 1,000 different worlds, generated using different seeds.

| Mario MDP model for a threshold of 30: | | |
|---|---|---|
| | Switching Wins | Switching Loses |
| Risk-neutral Wins | 754 | 18 |
| Risk-Neutral Loses | **59** | 169 |
| No switching lost : 228 times | | |
| Switching strategy lost: 187 times. | | |

Table 2: Fails 4.1% less using switching strategy; reduces losses by 18%.

| Mario MDP model for a threshold of 100: | | |
|---|---|---|
| | Switching Wins | Switching Loses |
| Risk-neutral Wins | 40 | 41 |
| Risk-Neutral Loses | **222** | 697 |
| No switching lost : 919 times | | |
| Switching strategy lost: 738 times. | | |

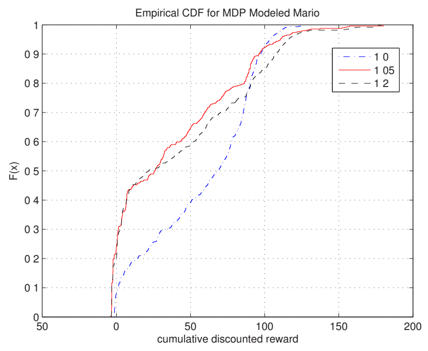
Table 3: Fails 18.1% less using switching strategy; reduces losses by 19.7%

| Mario simulator for a threshold of 30: | | |
|---|---|---|
| | Switching Wins | Switching Loses |
| Risk-neutral Wins | 72 | 90 |
| Risk-Neutral Loses | **107** | 731 |
| No switching lost : 838 times | | |
| Switching strategy lost: 821 times. | | |

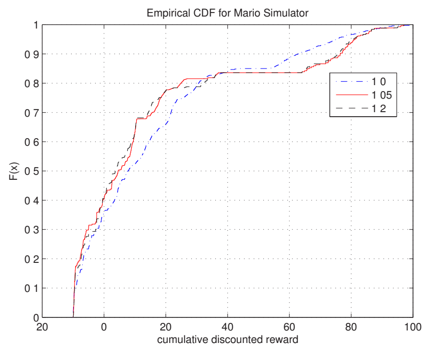
Table 4: Fails 1.7% less using switching strategy; reduces losses by 2%

## 5.3 Mario Discussion

The results for trajectories run in the MDP model demonstrate how beneficial the switching strategy is when the model is well known. As the threshold value increases the



(a) State 0, Mario MDP model CDF



(b) State 0, Mario simulator model CDF (collected for the same start state)

Figure 4: CDFs for the Mario MDP model and the Infinite Mario simulator for start state 0. The graphs show the estimated reward function of the risk-neutral policy ($\delta$=1) compared to risky policies ($\delta = 1.05, \delta = 1.2$) for each domain.

switching strategy is more beneficial because the higher threshold takes riskier actions sooner, which allows the resulting CDF to reside in between the risk-neutral and risky policy at higher values. Low thresholds may never take risky actions soon enough to reach the higher cumulative discounted reward values.

The results for trajectories in the Mario simulator show that even in a world that may not be modeled perfectly the switching strategy can provide some benefit. There is no improvement when using a threshold of 80, but in Figure 4b one can see that a threshold this high is approaching the upper bound for what the agent can achieve in practice.

## 5.4 Pizza Domain

We also evaluated our algorithm in a navigation domain, where a vehicle drives through a non-deterministic world for the purpose of delivering a pizza. The idea is that the delivery driver may need to be risk-sensitive in order to make the delivery on time. The state space has three dimensions: an x,y location and a Boolean value indicating whether the driver has a pizza. For a ten by ten grid, the world contains 200 states (refer to Figure 5).

The world contains the following actions: (PICKUP,

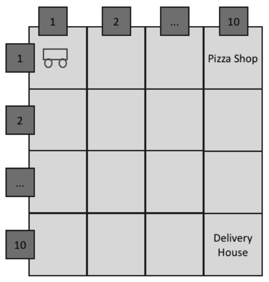


Figure 5: Navigation grid for pizza delivery world

| Reward Mappings | | |
|---|---|---|
| Action Name | State | Reward |
| DROPOFF | x,y=delivery location, have pizza | 50 |
| PICKUP | x,y=pizza shop, no pizza | -1 |
| RISKMOVE{N,S,E,W} | any | -2 |
| MOVE{N,S,E,W} | any | -6 |

Table 5: Reward mappings of the MDP model

DROPOFF, MOVE{N,S,E,W}, RISKMOVE{N,S,E,W}). The MOVE{N,S,E,W} actions are more deterministic and have a higher cost. Riskier actions (RISKMOVE{N,S,E,W}) are cheaper and have a chance of traveling further, but have a lower probability of actually progressing. For action MOVE{N,S,E,W}, there is an 80% chance of moving one square and a 20% of staying in the same square. The probabilities for the action RISKMOVE{N,S,E,W} are a 9% chance of moving one square, a 7% chance of moving two squares, and an 84% of staying put. Table 5 presents the immediate rewards for this domain.

## 5.5 Pizza Domain Results

The results in Tables 6 and 7 compare the risk-neutral policy and a risky policy with risk factor $\delta = 1.2$, at two different threshold values. The policies generated are run in the original MDP domain. Each group of results compares 10,000 trajectory runs between the risk-neutral policy versus the switching strategy. As before, we generate random probabilities offline on a per-state basis to compare the policies fairly.

As stated in the introduction, a straightforward approach to the problem of exceeding reward thresholds is to encode cumulative reward explicitly in the state space. Since the pizza delivery domain is small enough, we can feasibly do that and compare the results against our approach. The state space of the pizza domain was augmented to include an additional dimension of cumulative reward. We capped the maximum and minimum cumulative reward from 0 to -150 and removed discounting. This inflates the 200 states

| For a 30% threshold (threshold = -100): | | |
|---|---|---|
| | Switching Wins | Switching Loses |
| Risk-neutral Wins | 6422 | 458 |
| Risk-Neutral Loses | **1412** | 1708 |
| No switching lost : 3120 times | | |
| Switching strategy lost: 2166 times. | | |

Table 6: Fails 9.5% less using switching strategy; reduces losses by 30.6%

| For a 60% threshold (threshold = -88.5): | | |
|---|---|---|
| | Switching Wins | Switching Loses |
| Risk-neutral Wins | 2885 | 830 |
| Risk-Neutral Loses | **3271** | 3014 |
| No switching lost : 6285 times | | |
| Switching strategy lost: 3844 times. | | |

Table 7: Fails 24.4% less using switching strategy; reduces losses by 38.8%

to 30,000 states. The MDP reward function now returns a large positive reward if the driver delivers a pizza at the goal and the cumulative reward exceeds the reward threshold and a small positive reward for delivering the pizza while not exceeding the threshold, to encourage the planner to achieve the goal regardless. An additional cost is also added for states that are not the goal and are under the threshold value. This parallels the MDP transformation necessary to use thresholded reward objective functions as explored in (McMillen and Veloso 2007).

As expected, the offline planning times for the reward-augmented policy were significantly greater than for our algorithm. Solving for the policy with 30,000 states took approximately 18 hours, while solving for the 200 state policy took a few seconds (using an Intel 3.15 Ghz processor). Even though our algorithm must solve multiple policies, generate offline reward distributions for each state (which took 5 to 10 minutes per policy) and construct the corresponding CDFs (which took about 1 minute per policy), the processing is still significantly less than it takes to solve for the augmented-reward policy.

The difference in run time computation is small. Our approach must evaluate a point on the CDF for each policy, which is a straightforward linear interpolation. There is a cost for reading in the CDFs (which are generated offline) for each state of each policy, but that is done just once, at start up.

Comparing the results of the policies did not show a significant difference between the reward-augmented and switching policies. Both performed better than the risk-neutral policy, but their similarity could be attributed to the fact that in such a simple domain the switching strategy is approaching optimal. In general, the reward-augmented policy is expected to perform better (and should behave optimally, with respect to the objective of exceeding the reward threshold).

### 5.6 Pizza Domain Discussion

Besides performing significantly better than the risk-neutral policy, it is interesting to note that the average trajectory lengths are higher for the switching strategy versus the following the risk-neutral policy. Also, the trajectory lengths for the switching strategy increase as the threshold increases because as more risky actions are taken there is a higher chance of getting stuck in the same state.

Policies generated contained all MOVE actions for the risk-neutral case and more RISKMOVE actions depending on the convexity of the risk factor. The reward-augmented policy chose more risky actions as the cumulative reward got closer to the threshold, and returned to MOVE actions once the threshold was exceeded.

While increasing the state space to include cumulative reward creates an optimal policy for thresholded rewards, there is a tradeoff with longer execution time and less flexibility for setting the threshold. The optimal policy must be re-generated for every threshold that needs to be tested. This can take days depending on the size of the state space. Our algorithm allows a threshold to be re-set during the switching stage, and does not affect the offline policy generation.

## 6 Conclusion

For these specific domains, there was not a large difference between the risk-seeking levels, so results are shown only comparing risk-neutral with one risky policy. This algorithm allows for more complex domains to compare with multiple risk-sensitive policies based on the architect's preferences.

We presented a domain-independent algorithm that aims to maximize the probability of exceeding a threshold at execution time using risk-sensitive policies. This was demonstrated on two domains showing the benefits of taking more risks to win. For future work, we would like to continue to explore additional ways an agent adapts and operates reliably in a dynamic environment. More specifically, having the agent gather more contextual awareness on whether it was winning or losing is useful. The long term goal is to apply these principles to enhance the robustness of real robotic systems.

## 7 Acknowledgments.

This work is supported by AFOSR grant #FA2386-10-1-4138. We thank Sven Koenig and Yaxin Liu for their help with the risky policy transformations, and thank John Laird and Shiwali Mohan for their assistance with the Super Mario domain and simulator.

## References

Comanici, G., and Precup, D. 2010. Optimal policy switching algorithms for reinforcement learning. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 709–714. International Foundation for Autonomous Agents and Multiagent Systems.

Geibel, P., and Wysotzki, F. 2005. Risk-sensitive reinforcement learning applied to control under constraints. *Journal of Artificial Intelligence Research* 24(1):81–108.

Koenig, S., and Simmons, R. 1994. How to make reactive planners risk-sensitive. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, volume 293298.

Liu, Y., and Koenig, S. 2008. An exact algorithm for solving mdps under risk-sensitive planning objectives with one-switch utility functions. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, 453–460. International Foundation for Autonomous Agents and Multiagent Systems.

Liu, Y. 2005. *Decision-theoretic planning under risk-sensitive planning objectives*. Ph.D. Dissertation, Citeseer.

McMillen, C., and Veloso, M. 2007. Thresholded rewards: Acting optimally in timed, zero-sum games. In *Proceedings of the national conference on artificial intelligence*, volume 22, 1250. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Mohan, S., and Laird, J. 2010. Relational reinforcement learning in infinite mario. *Ann Arbor* 1001:48109Y2121.

Mohan, S., and Laird, J. 2011. An object-oriented approach to reinforcement learning in an action game. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Pratt, J. 1964. Risk aversion in the small and in the large. *Econometrica: Journal of the Econometric Society* 122–136.

Roth, M.; Simmons, R.; and Veloso, M. 2005. Reasoning about joint beliefs for execution-time communication decisions. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, 786–793. ACM.

Tetreault, J.; Bohus, D.; and Litman, D. 2007. Estimating the reliability of mdp policies: a confidence interval approach. In *Proceedings of NAACL HLT*, 276–283.

Wagman, L., and Conitzer, V. 2008. Strategic betting for competitive agents. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, 847–854. International Foundation for Autonomous Agents and Multiagent Systems.