Model-Based Reinforcement Learning in Dynamic Environments

Marco A. Wiering
marco@cs.uu.nl
Intelligent Systems Group
Institute of Information and Computing Sciences
Utrecht University

Abstract

We study using reinforcement learning in particular dynamic environments. Our environments can contain many dynamic objects which makes optimal planning hard. One way of using information about all dynamic objects is to expand the state description, but this results in a high dimensional policy space. Our approach is to instantiate information about dynamic objects in the model of the environment and to replan using model-based reinforcement learning whenever this information changes. Furthermore, our approach can be combined with an a-priori model of the changing parts of the environment, which enables the agent to optimally plan a course of action. Results on a navigation task in a Wumpus-like environment with multiple dynamic hostile spider agents show that our system is able to learn good solutions minimizing the risk of hitting spider agents. Further experiments show that the time complexity of the algorithm scales well when more information is instantiated in the model.

Keywords: Reinforcement Learning, Dynamic Environments, Model-based RL, Instantiating Information, Replanning, POMDPs, Wumpus

1 Introduction

Reinforcement learning. Reinforcement learning (Sutton and Barto, 1998; Kaelbling et al., 1996) can be used to learn to control an agent by letting the agent interact with its environment and learn from the obtained feedback (reward signals). Using a trial-and-error process, a reinforcement-learning (RL) agent is able to learn a policy (or plan) which optimizes the cumulative reward intake of the agent over time. Reinforcement learning has been applied successfully in particular stationary environments such as in checkers (Samuel, 1959), backgammon (Tesauro, 1992), and chess (Baxter et al., 1997). Reinforcement learning has also been applied to find good solutions for difficult multi-agent problems such as elevator control (Crites and Barto, 1996), network routing (Littman and Boyan, 1993), and traffic light control (Wiering, 2000). RL has only been used few times in single agent non-stationary environments, however. Path-planning problems in non-stationary environments are in fact partially observable Markov decision problems (POMDPs) (Lovejoy, 1991), which are

known to be hard to solve exactly. Dayan and Sejnowski (1996) concentrate themselves on the dual control or exploration problem where there is the need of detecting changes in a changing environment, while the agent should act to gain as much reward as possible. Boyan and Littman (2001) use a temporal model to take changes of the environment into account when computing a policy. In this paper we are interested in applying RL to learn to control agents in dynamic environments.

Dynamic environments. Learning in dynamic environments is hard, since the agent needs to stay informed about the status of all dynamic objects in the environment. This can be done by augmenting the state space with a description of the status of all dynamic objects, but this may quickly cause a state space explosion. Furthermore, the agent may not exactly know the status of an object and therefore has to deal with uncertain information. Using uncertain information as part of the state space is hard, since it makes the state space continuous and high dimensional.

Instantiating information in the model. There exists another method for using knowledge about dynamic objects: *instantiate* the information about the dynamic objects in the world model and then use the revised world model to compute a new policy. E.g. if a door can be open or closed, and we know whether the door is closed, we can set new transition probabilities between states in the world model such that this information can be used by the agent. Once the model is updated using the currently available information, dynamic programming-like algorithms (Bellman, 1957; Moore and Atkeson, 1993) can be used to compute a new policy. In this way, we have an adaptive agent which takes currently known information into account for computing actions, and which replans once the dynamic information changes. This is hard to do with other planning methods, especially for closed loop control in stochastic dynamic environments. Furthermore, the agent could also instantiate information received by communication which can be useful for multi-agent reinforcement learning. Although sharing policies (Tan, 1993) is one way for cooperative multi-agent learning, communication with instantiating information can also be used for non-cooperative or semi cooperative environments.

Using prior knowledge. Often reinforcement learning is used to learn control knowledge from scratch, i.e. without using a-priori knowledge. We know, however, that the use of some kind of a-priori knowledge can be very beneficial. For example, if particular actions are heavily punished we do not want to explore those actions, but rather reason about the consequences of these actions using an a-priori designed model. A-priori knowledge can also be used to model a dynamic environment so that this knowledge can be presented to the RL agent. This enables the agent to reason about the dynamics of the environment which may be necessary to solve a particular problem, where problems may arise one after the other. As an example think about an agent which is walking in a city and uses RL to learn a map of the city. After some time, the agent may have the desire to drink something in a bar. Once the agent enters some bar, it could use an a-priori model of bars to understand which dynamic entities, such as a barkeeper, other customers, tables and chairs etc. play a role in the bar-setting. So it can use this model, fill in the actual situation using sensor

data (e.g., vision) and compute a policy (or plan) to attain its current goal. If the agent discovers more information about particular (dynamic) entities, it can again instantiate this in the model of the current bar situation and recompute a policy. In this way, the RL agent does not need to try all its actions, but can efficiently compute a particular plan suited for the situation at hand. In this paper, we will also study using a-priori knowledge for learning to solve problems in dynamic partially observable environments.

Outline of this report. We will describe model-based RL in Section 2. Then using instantiated information is described in Section 3. Then we describe the experimental setup and results in Section 4. Section 5 provides a discussion which relates our framework to POMDPs and describes the limitations of the current approach and proposes possible extensions. Finally, Section 6 concludes this paper.

2 Model-Based Reinforcement Learning

In this section we describe model-based reinforcement learning (MBRL), and in particular prioritized sweeping (Moore and Atkeson, 1993; Wiering and Schmidhuber, 1998) which is used in our current experiments. The main reason for using model-based RL is that instantiating information is possible with these algorithms, whereas it is not possible to combine instantiating information with direct model-free RL algorithms such as Q-learning (Watkins, 1989). The reason for using prioritized sweeping is that this algorithm is very efficient in managing the necessary updates of the Q-function.

2.1 Markov Decision Problems

Although we study dynamic environments, we use the well-known Markov decision process framework as a model of the environment and task. This means that policies will be computed based on the current state of the Markov decision problem. If there are dynamic changes in the environment, we change the underlying transition and reward functions, and recompute the policy. This will be explained in section 3.

We consider a finite set of states $S = \{S_1, S_2, \ldots, S_n\}$, a finite set of actions A, and discrete time steps $t = 1, 2, 3, \ldots$. Let s_t denote the state at time t, and $a_t = \Pi(s_t)$ the action, where Π represents the agent's policy mapping states to actions. The transition function P with elements $P_{ij}(a) := p(s_{t+1} = j | s_t = i, a_t = a)$ for $i, j \in S$ defines the transition probability to the next state s_{t+1} given s_t and a_t. A reward function R maps state/action/state tupels $(i, a, j) \in S \times A \times S$ to scalar reinforcement signals $R(i, a, j) \in \mathbb{R}$. A discount factor $\gamma \in [0, 1]$ discounts later against immediate rewards. The agent's goal is to select actions which maximize the expected long-term cumulative discounted reinforcement, given an arbitrary state $\in S$. For this goal, the agent learns two different value functions. The value $V^{\Pi}(i)$ is a prediction of the expected discounted cumulative reward to be received in the future, given that the agent

is currently in state i and policy Π will be used in the future:

$$V^{\Pi}(i) = E(\sum_{k=0}^{\infty} \gamma^k R(s_k, \Pi(s_k), s_{k+1}) | s_0 = i)$$

Action evaluation functions (Q-functions) $Q^{\Pi}(i,a)$ return the expected future discounted reward for selecting action a in state i, and subsequently executing policy Π:

$$Q^{\Pi}(i,a) = \sum_j P_{ij}(a)(R(i,a,j) + \gamma V^{\Pi}(j))$$

where V^{Π} is defined as: $V^{\Pi}(i) = \max_a Q^{\Pi}(i,a)$. By setting:

$$\Pi(i) = argmax_a Q^{\Pi}(i,a)$$

for all states i we then iteratively improve the policy.

2.2 Estimating a Model

In reinforcement learning we often do not initially possess a model containing the transition and the reward functions, and therefore we have to learn these from the observations received during the interaction with the environment. Inducing a model from experiences can be done by counting the frequency of observed experiences. For this the agent uses the variables:
$C_{ij}(a)$:= number of transitions from state i to j after executing action a.
$C_i\ (a)$:= number of times the agent has executed action a in state i.
$R_{ij}(a)$:= sum of all immediate rewards received after executing action a in state i and stepping to state j.

A maximum likelihood model (MLM) is computed as:

$$\hat{P}_{ij}(a) := \frac{C_{ij}(a)}{C_i(a)} \text{ and } \hat{R}(i,a,j) := \frac{R_{ij}(a)}{C_{ij}(a)} \quad (1)$$

After each experience the variables are adjusted and the MLM is updated. In deterministic environments one experience per state/action pair (SAP) is sufficient to infer the true underlying model. In stochastic environments, however, we need to explore the effects of an action in a state ad infinitum.

2.3 Prioritized Sweeping (PS)

Dynamic programming (DP) techniques (Bellman, 1957) could immediately be applied to the estimated model, but online DP, which updates the complete value function with value iteration after each step, tends to be computationally very expensive. Although offline DP which recomputes the policy after a complete trial, would be more efficient, online updating is much better for efficient exploration and is especially needed in dynamic environment. To speed up online DP algorithms in complex environments, some sort of efficient update-step management should be performed.

This can be done by prioritized sweeping (PS) (Moore and Atkeson, 1993) which assigns priorities to updating the Q-values of different state/action pairs (SAPs) according to their relative update sizes. Following the update of a state-value, the state's predecessors are inserted in a priority queue. Then the priority queue is used recursively for backpropagating the update of the states with highest priority.

Moore and Atkeson's PS (M+A's PS) calculates the priority of some state by checking all transitions to updated successor states and identifying the one whose update contribution is largest. Our variant allows for computing the *exact* size of updates of state values since they have been used for updating the Q-values of their predecessors, and yields more appropriate priorities. Unlike our PS, M+A's PS cannot detect large state-value changes due to many small update steps, and will forget to process the corresponding states.

Our implementation (Wiering, 1999) uses a set of predecessor lists $Preds(j)$ containing all predecessor states of state j. We denote the priority of state i by $|\Delta(i)|$, where the value $\Delta(i)$ equals the change of $V(i)$ since the last time it was processed by the priority queue. To calculate it, we constantly update all Q-values of predecessor states of currently processed states, and track changes of $V(i)$. The details are given below.

Our Prioritized Sweeping:

1) Promote the most recent state k to the top of the priority queue
2) $\forall$ a do:
   3 $Q(k,a) := \sum_j \hat{P}_{kj}(a)(\hat{R}(k,a,j) + \gamma V(j))$
4) While $n < U_{max}$ AND the queue is not empty
   5 Remove the top state s from the queue
   6 $\Delta(s) := 0$
   7 $\forall$ Predecessor states i of s do:
      8 $V'(i) := V(i)$
      9 $\forall$ a do:
         10 $Q(i,a) := \sum_j \hat{P}_{ij}(a)(\hat{R}(i,a,j) + \gamma V(j))$
      11 $V(i) := \max_a Q(i,a)$
      12 $\Delta(i) := \Delta(i) + V(i) - V'(i)$
      13 If $|\Delta(i)| > \epsilon$
         14 Promote i to priority $|\Delta(i)|$
   15 $n := n + 1$

The parameter U_{max} is the maximal number of updates to be performed per update-sweep. The parameter $\epsilon \in I\!R^+$ controls update accuracy.

3 Instantiating Information

For particular environments with dynamic objects, the agent should have information about the status (e.g., position) of these objects. One way of using this information is to expand the state space to include the state of all dynamic objects. However, suppose that we possess information about a dynamic object

in the form of occupancy probabilities. Clearly it is not desirable to include these occupancy probabilities in the state space, since this would result in a high dimensional continuous state space which makes planning and the use of dynamic programming-like algorithms hard.

Instantiating information. Another way is to *instantiate* the information about the dynamic object in the world model. E.g. if we have information about occupancy probabilities of robots in a soccer game, we may adjust the model's transition probabilities to account for possible hits with obstacles. Thus, expected occupancy probabilities of a hostile agent can be used for setting transition probabilities to a (possibly terminal) encounter with the hostile agent. In this paper we only study replanning where collision avoidance plays the main role. Therefore we model all collisions with other agents as highly punished transitions to terminal states. In case an agent could pick up objects such as a tool (e.g., a hammer) and can perform actions with that tool, the problem becomes more complex. In that case, we need to include the objects as part of the state space, or we need to have multiple temporal branches between world models. This may quickly become intractable, and therefore in this paper we only consider terminal hits with dynamic objects.[1]

An example of instantiating information. Suppose that the agent receives new information that the hostile agent has probability $p(j)$ to occupy a specific state j. If the agent makes a step after which she meets the hostile agent, she dies. How do we then change the model to incorporate the information about occupancy probabilities of the hostile agent? Clearly we have to reset the transition counters and reward variables, since this is what our model consists of. We define the transition counter from state i to some terminal state (H for hit) which is occupied by a hostile agent if action A is executed as $C_{iH}(a)$. Now if some state action pair (i, a) can make a transition to state j with probability $\hat{P}_{ij}(a)$ and we know the probability that a hostile agent occupies state j is $p(j)$, we set the transition counter for modelling transitions from i to the terminal state H (hit) to:[2]

$$C_{iH}(a) := \frac{p(j)\hat{P}_{ij}(a)(C_i^{old}(a) - C_{iH}^{old}(a))}{1 - p(j)\hat{P}_{ij}(a)}$$

and

$$C_i(a) := C_i^{old}(a) - C_{iH}^{old}(a) + C_{iH}(a)$$

In this way the new probability $\hat{P}_{iH}(a)$ will become $p(j)\hat{P}_{ij}(a)$. We set the reward $R_{iH}(a)$ to R_{hit}.

General algorithm. In case an action a from a state i can result in multiple states j all with a different probability of being occupied by the hostile agent, we cannot set the transition counter to one of these transitions immediately, but have to sum the transition counter over all transitions to states which may be occupied by a hostile agent. For this we first reset all counters

[1] In principle any state transition could be modelled, as long as there are not novel objects involved which were not already included in the state description.

[2] If $p(j)\hat{P}_{ij}(a) = 1$, we set the transition counter to a very large number.

to hostile states to 0, and then recompute the counters using the occupancy probabilities. The following algorithm does this:

Instantiating information :

1) For all state-action pairs (i, a) which are in a possible area of the hostile agent do:
    2) $C_i(a) := C_i(a) - C_{iH}(a)$
    3) $C_{iH}(a) := 0$
4) For all hostile areas D do:
    5) For all (i, a) pairs which can lead to some successor state k in D do:
      6) For all successor states j of (i, a)
        7) If state j falls inside D with prob.$p(j)$
          8) $C_H := C_{ij}(a)p(j)$
        9) $C_T := C_{ij}(a)$
        10) $p_{old} := C_{iH}(a)/C_i(a)$
        11) $p_{new} := \hat{P}_{ij}(a)C_H/C_T$
        12) $\Delta C := \frac{C_i(a)*(p_{new}+p_{old})-C_{iH}(a)}{(1-p_{new}-p_{old})}$
        13) $C_{iH}(a) := C_{iH}(a) + \Delta C$
        14) $C_i(a) := C_i(a) + \Delta C$
      15) $\hat{R}(i, a, H) := R_{hit}$

This algorithm exactly recomputes the desired probabilities:

$$P(H|i, a) = \sum_s P(i, a, s)P(H|s)$$

for transitions from a state/action pair to a hit with some hostile agent (dynamic obstacle) and renormalized the other probabilities. Note that rewards $\hat{R}(i, a, H)$ are set to R_{hit} which is predefined in the reward function.

Using prioritized sweeping to replan. After we instantiated all newly available information, we store all changed states at the top of the priority queue and use prioritized sweeping to recompute the policy. This ensures that the new information is immediately used.

4 Experiments

We have executed two sets of experiments (Wumpus II and Wumpus III) to validate the usefulness of our method. In both experiments we have to deal with partial observability of the dynamic hostile spider agents. In the first set of experiments, the agent uses an a-priori model to reason where the dynamic hostile agents might be. In the second set of experiments the agent uses a limited sensor to observe where the hostile agents are. For the second set of experiments we also study time complexity issues for updating the model after instantiating new information.

4.1 Wumpus II

The first set of experiments consists of a number of different experiments. In the first experiment we have a small maze and one hostile spider agent, and in the second experiment we have a larger maze and 5 (10 or 20) moving spider agents. The agent needs to find the goal in the least number of steps without hitting a spider. The agent cannot see the spider, however. If the agent hits a spider it dies and knows the region where the spider was. A spider agent occupies a particular nest and moves randomly around the nest so that all states in the region of an active spider nest have the same occupancy probability. If the agent finds an active nest, the agent can smell whether there is an active spider in the region or not. Figure 1 shows the first environment used in the experiments which consists of two spider nests. For the region around the spider nest, we use 25 states. During a trial, the spider moves randomly in the region around the nest. After a trial, the spider may move to a different nest or goes to its current nest.

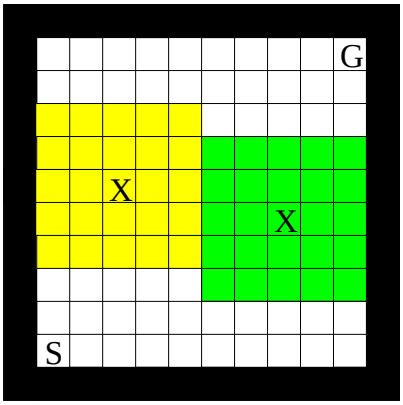

Figure 1: The maze environment containing two spider nests X which may be used by the spider. The start and goal positions are indicated by S and G. The regions around the spider nests denote which states the spider may occupy if it uses a particular nest. The agent and the spider can move in 4 directions.

4.1.1 The Model of the Environment

The agent knows its exact (X, Y) location at all times, but cannot observe the spider agent. It only knows the exact spider location when it hits the spider, but then it dies so that information is only partially useful, since a new trial starts and the spider is reset to the position of the new active, possibly neighboring, spider nest. After each trial the spider has a particular probability $P_{move} = 1 - P_{stay}$ of moving to one of the neighboring nests.

Modelling occupancy probabilities. In the beginning the agent does not know where the spider nests are. It has to discover these for itself, but once it hits a spider nest, it remembers its location in (X, Y) coordinates. The

a-priori knowledge of the agent consists of its knowledge of the size of the region of a spider nest in which the spider moves randomly, and the probabilities that the spider makes a transition to a new neighboring spider nest after each trial[3]. The agent uses a probabilistic model of the spider's location. The occupancy probabilities $P(S\_spider)$ are computed by:

$$P(S\_spider) = \sum_i P(Active_i)P(S\_spider|Active_i)$$

Here, the agent uses for hostile area i the probabilities: $P(Active_i)$ and the conditional probabilities $P(S\_spider|Active_i)$ to compute the probabilities $P(S\_spider)$. The probability $P(S\_spider|Active_i)$, that a spider occupies a specific state if it travels through an active region, is set to $\frac{1}{M}$ for the M (25) states surrounding a spider region and 0 to other states. Note that we model the stationary distribution of the spider's location.

Computing active nest probabilities. The first probability $P(Active_i)$ models the probability that a spider agent occupies a particular region i and follows from the properties of the dynamic stochastic system. There are several ways to get new information about the state of the system: (1) The agents finds a nest and observes whether it is used by a spider or not, (2) The agent hits the spider around some nest, (3) A new trial starts, and the agent knows that the spider may have migrated to a neighboring nest. In case the agent finds a spider nest, it can see whether the nest is active or not. In case the nest is active it sets the probability $P(Active_i)$ to 1.0, and sets the probabilities for the other nests to 0. If the nest is not active the agent sets the probability $P(Active_i)$ to 0.0 and renormalizes the probabilities of the other nests.

Hitting the spider. In the same way, in case the agent hits the spider, the agent knows that the region in which it was walking contained the active nest, and sets the probability $P(Active_i)$ to 1.0 and the other nest probabilities to 0.0. After an encounter with the spider, the agent dies and a new trial is started.

Transition probabilities between nests. After each trial, the spider may change its nest. In the small maze given above, the spider has probability $P_{stay} = 0.9$ of staying in the same nest and probability 0.1 of moving to the other nest after each trial. Therefore we recompute the active nest probabilities of the model after each trial by:

$$P(Active_1) = 0.9P(Active_1) + 0.1P(Active_2)$$

And vice versa for the other nest.

Discovering nests. Since the agent can only set probabilities to non-zero for nests it has discovered and knows that the spider can only move between nests which are closer than a particular distance Manhattan D (which is set to 7 and defines the neighbourhood relation between nests), the probability cannot be exactly computed in case the agent has not yet discovered all spider nests. Therefore exploration is important to ensure all spider nests have been found.

[3]Although learning this information is possible, it would require many interactions with the spider or with spider nests and therefore take a very long time.

Using additional a-priori information. Finally, we use a-priori information in the form of an initial state transition model. For each action in a state (X, Y), we set the transition counter to 1 for the successor state (i.e. state $(X + 1, Y)$ for action East) as if actions were deterministic and no states are blocked. This initial information can be easily obtained and used in case of maze environments, and makes it easier to implement the instantiating information procedure (to deal with unvisited states which may contain a spider). Of course, initially the position of the goal and spider's nests are unknown and should be discovered by the agent. Furthermore, in case of maze-like environments as in the second maze (see Figure 3), the initial transition model in the maze is less helpful, since maze-locations may be occupied. For this maze, we therefore initialize the transition counters to 0.0001.

4.1.2 Experiments with the Small Maze

First we have executed experiments with the small maze given in Figure 1.

Systems. We compare using the a-priori spider model using instantiated information to using model-based RL without using the spider model and instantiated information. The second algorithm computes probabilities of hitting the spider based on previous experiences resulting in a confrontation with the spider. It does not use any kind of a-priori knowledge. With each system we perform 10 simulations. The algorithm using prior knowledge has access to the following information: the number of states in a spider region, the distance D between neighboring spider nests, the probability $P_{move} = 1 - P_{stay}$ of spider transitions between nests, the knowledge that a spider moves randomly in its active region, and finally the initial state transition information to navigate in deterministic empty mazes. Using this information, it keeps track of the spider model which after a change in spider occupancy probabilities is instantiated in the agent's world model.

Problem description. The reward for hitting the spider is -5000, the reward for reaching the goal state is 1000. The reward for an individual step is -1. We experimented with a deterministic environment, with 10% noise, and with 25% noise. If a noisy action is executed, the agent has probability 25% of executing each of its actions. Spiders move randomly to neighboring squares and can cross blocked states (except for the borders of the maze of course).

Parameters. After a coarse search through parameter space to find the best learning parameters, we used the following setup: We use max-random exploration with $P_{exp} = 0.5 \rightarrow 0.0$ (we anneal the exploration probability). The discount factor γ when using the spider model is set to 0.9999, the discount factor without spider model is set to 0.95. The update accuracy ϵ is set to 1.0. Finally, the maximal number of updates $U_{max} = 500000$ (which we used to make almost optimal use of the instantiated information possible).

Results. Figure 2(A) shows the average cumulative reward in 100 test trials after each 50 steps during the first 1000 training steps in the deterministic environment. Within 1000 steps, both methods have learned to find good solutions, but using the model results quickly in near optimal performance. Figure 2(B) shows the obtained cumulative reward intake during each 100 test trials

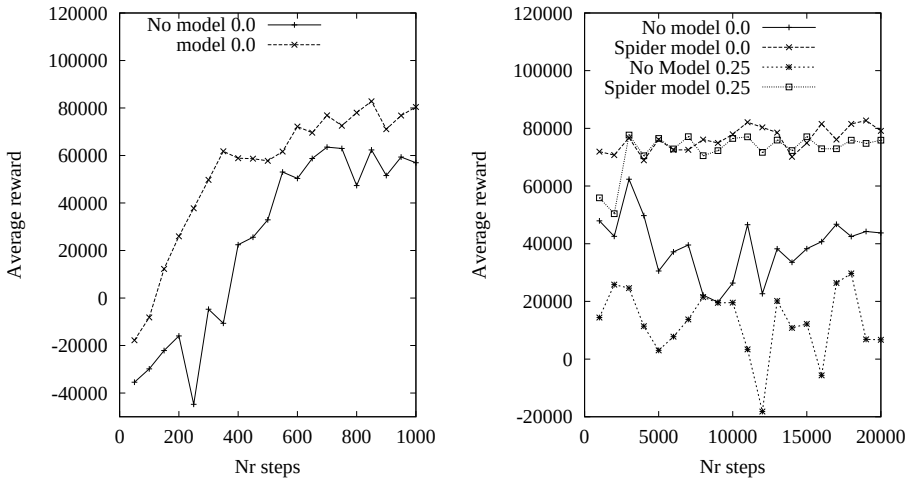

Figure 2: The results for the small maze environment containing two spider nests. (A) shows learning results for the first 1000 steps. (B) shows the results for much longer simulations. Results are averages over 10 simulations.

after each 1000 learning steps of the two different algorithms for the small maze with deterministic actions and with 25% randomness in the action selection. The figure clearly shows that using the spider-model outperforms not using the model. Basically, the agent can reason about the spider's location and use its marginal information to compute optimal dynamic policies. Thus, the agent always prefers to go through the region with the smallest probability of containing the spider. This is impossible to learn without using the spider model, although it is clearly beneficial in particular dynamic environments. The simulation for 20,000 steps costs 358 seconds for using the spider model and 12 seconds for not using it.

Table 1: Results for the systems with (With) and without (No) the spider model. Noise refers to the amount of noise in the action execution. Goal/Spider hit refers to the number of test trials resulting in a hit with the goal/spider. Final reward denotes average reward of the last 100 test trials.

Model (noise)	Goal hit	Spider hit	Final reward
With (0.0)	1927 ± 8	73 ± 8	79K ± 10K
With (0.1)	1917 ± 12	83 ± 12	68K ± 9K
With (0.25)	1910 ± 41	80 ± 19	76K ± 11K
No (0.0)	1802 ± 20	198 ± 20	44K ± 31K
No (0.1)	1782 ± 20	218 ± 20	36K ± 23K
No (0.25)	1717 ± 24	283 ± 24	7K ± 25K

Table 1 shows the total number of goal hits and spider hits in a total of 2000 test trials (100 test trials after each 1000 steps in a 20,000 step simulation) and

the average reward intake during the last 100 test trials. The table clearly shows that using the spider model leads to fewer hits (4% vs. 11%) with the spider (and therefore a larger number of times the goal was reached). It should be mentioned that during the initial trials most hits with spiders are made, and that it is impossible to avoid spider hits completely — the position of the spider can never remain completely known. The table also shows that additional randomness does not lead to significantly more hits with the spider when the spider model is used, The reason is that the agent learns to circumvent the dangerous region in most trials, and therefore does not suffer much from random actions which let the agent stay there longer. Additional randomness decreases the performance of the RL agent without spider model drastically, however. An interesting phenomena when using the spider model is that in particular simulations, the agent has learned a path traversing the spider nest's location so that it is able to get more information whether the current path is safe (nest is not active). If not, the agent plans a new path. In our current work we have not exploited this information gain, however, see chapter 5 for a possible extension of the algorithm which can be used for dual control.

4.1.3 Experiments with a Large Maze with Multiple Spiders

We have also experimented with a larger maze of size 50 $\times$ 50 (see Figure 3) containing 30 possible locations for spider nests, and 5 spiders traversing 30% of the maze. The maze also contains about 20% randomly distributed blocked states and 20% penalty states.

Reward function. For hitting the goal, the agent receives a reward of 2500. For hitting the spider, the reward is -10,000. For hitting a blocked state, the reward is -2, for hitting a penalty state, the reward is -10, and other steps are rewarded by -1.

Parameters. The discount factor when using the model is set to 0.99999 which was used to make almost optimal use of the model possible. The discount factor without spider model which worked best is 0.99. The exploration rule Max-Random is used where the probability of selecting a random action is annealed from 0.5 to 0.0. The maximum number of updates per step is 50,000. The accuracy parameter ϵ is set to 0.5.

Simulation set-up. We perform 10 simulations with each system. The number of steps in a simulation is 200,000. After each 1,000 steps the systems are tested a single trial using a maximum number of 10,000 test actions. Thus, in total there are 200 tests. For these tests we compute the total number of times the goal has been found and the number of times one of the spiders is hit.

Results. Table 2 shows the number of times the goal has been found and the number of times a spider has been hit for different noise levels when P_{stay} is set to 0.4. The table clearly shows that using the spider model leads to many fewer hits with the spider. The number of hits with a spider is reduced by a factor of 3 when the spider model is used. It is clear that the agent is able to discover spider nests and to use the acquired information to plan paths which circumvent going through locations with a large probability of containing a spider. We can again observe that more randomness does not lead to worse

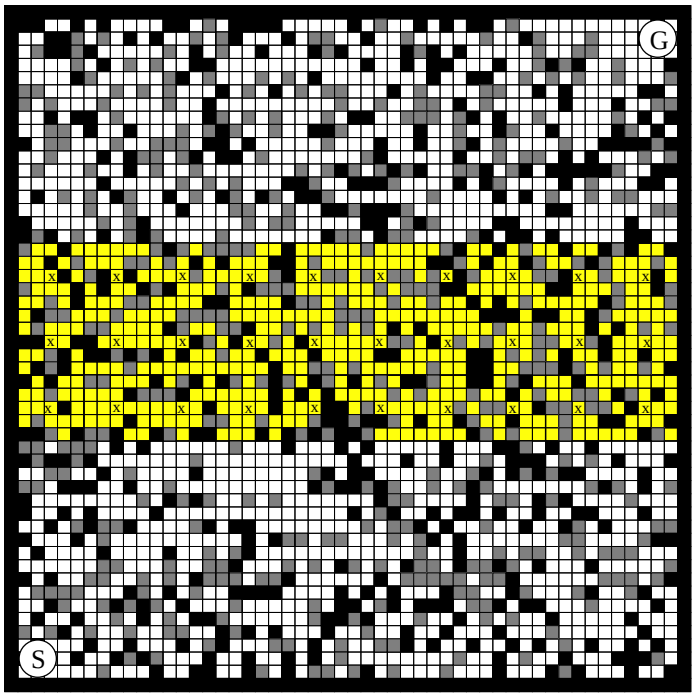

Figure 3: The large maze environment containing 30 spider nests (indicated by an X in the shaded area) and 5 active spiders. Black fields denote impassable walls. Dark grey fields denote penalty fields.

results when the spider model is used (the agent circumvents dangerous regions altogether), whereas the results become much worse when the spider model is not used.

Table 2: Results for the system with and without using the spider model. Noise refers to the amount of noise in the action execution. Spider hits refers to the number of test trials resulting in a hit with the spider.

System	Noise	Goal hits	Spider hits
With Model	0.0	173 ± 3	9 ± 2
With Model	0.1	173 ± 5	12 ± 4
With Model	0.25	173 ± 5	11 ± 3
No Model	0.0	163 ± 6	26 ± 6
No Model	0.1	160 ± 6	31 ± 7
No Model	0.25	152 ± 7	42 ± 7

Table 3 shows the number of times the goal has been found and the number of times a spider has been hit for values of P_{stay} where the noise is set to 0. It shows that our approach works better when the environment is more predictable. This indicates that the agent makes efficient use of the model. Both systems find very good solutions to the more deterministic task in which

spiders only move around their same unique nests.

Table 3: Results for the system with using the spider model for different values of the P_{stay} parameter. Evidently, using a larger value for P_{stay} leads to more predictable environments so that the spider model is more accurate.

Model (P_{stay})	Goal hits	Spider hits	Time (min)
With (0.1)	167 ± 6	12 ± 5	122 ± 25
With (0.4)	173 ± 3	9 ± 2	117 ± 21
With (0.9)	183 ± 3	4 ± 2	84 ± 6
With (1.0)	196 ± 2	0 ± 0	31 ± 15
No (0.1)	165 ± 5	23 ± 5	0.5 ± 0.2
No (0.4)	163 ± 6	26 ± 6	0.5 ± 0.2
No (0.9)	168 ± 5	19 ± 7	0.5 ± 0.3
No (1.0)	184 ± 4	4 ± 2	0.2 ± 0.0

Although instantiating information and replanning works very well, the computational time is significantly larger, since after each trial large portions of the policy have to be updated. We have not explored using other learning parameters to speed up the learning time, however. In the second set of experiments we study time complexity issues in more detail.

We finally performed experiments with different numbers of spiders. Table 4 show the results for both competitors when there are 5, 10, or 20 spiders, $P_{stay} = 0.9$ and the randomness in the action selection is 0.1.

Table 4: Results for the systems for different numbers of spiders.

Model	Nr. of spiders	Goal hits	Spider hits
With	5	190 ± 4	4 ± 2
With	10	169 ± 5	16 ± 3
With	20	132 ± 6	37 ± 5
No	5	164 ± 3	29 ± 5
No	10	121 ± 41	69 ± 35
No	20	85 ± 14	112 ± 14

Table 4 shows that when the spider model is used, the agent can circumvent hitting one of the moving spiders in a much better way than when the model is not used. This is also true with a large amount of dynamic hostile agents.

4.2 Wumpus III

In the second set of experiments we study complexity issues of the updating algorithm in more detail. For this we use an agent with a limited sensor for observing spiders which are close to the agent. When the sensor range is enlarged, the agents receives more information. A sensor range of 5 means that the agent observes the spider when the Manhattan distance is less than 5. If a spider is seen, the state and its neighboring states are assumed to be occupied

with 20% probability. This information is then instantiated in the model. We again use the large maze from Figure 3, and the same parameters as in the previous experiment. The number of spider agents in these experiments is set to 20. We computed the number of Bellman backups performed per time step with prioritized sweeping and the number of items (changed states) which are instantiated per time step. For plotting figure 4, we performed one simulation with one sensor range (from 5 to 35) in which only 50,000 steps were executed.

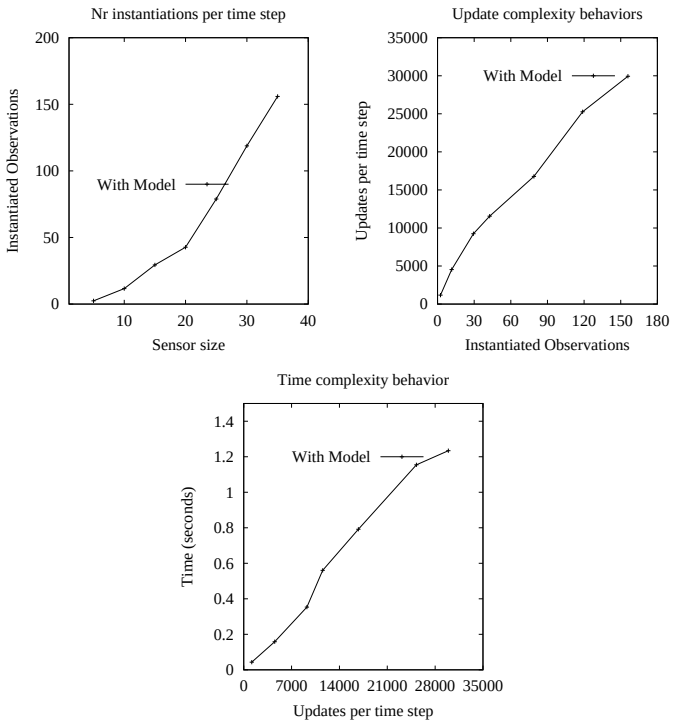

Figure 4: The complexity results for the large maze when using a limited sensor. Results are shown per step (performed action). (A) shows the number of instantiated items (changed states) for different ranges of the sensor. (B) shows how many updates are performed given the number of instantiated items. (C) shows the time needed to perform all updates per step.

Figure 4 shows three figures which display two measured variables against each other. The variables are: (a) The size of the limited sensor, (b) the average number of instantiated observations (changed states) per step, (c) the average number of updates performed per step, and (d) the average time (seconds) needed to perform the updates for one step.[4] We can see three things: (1) With a larger sensor range, the number of instantiated items increases faster than linearly. This can be explained by the fact that the number of states which the agent observes scales quadratically with the sensor range. (2) With more instantiated items, the number of performed updates to recompute the

[4]The simulations were run on a Ultra Sparc 5, 333MHz.

policy increases more or less linearly (although we should mention that the maximal number of updates is set to 50,000). (3) With more updates per step, the time increases approximately linearly (since it is the most important factor in a simulation). The longest experiments take about 1.2 seconds per step on our computer. Thus, the figures show that the algorithm scales more or less linearly with more instantiated information.

More detailed results are shown in Table 5. For these experiments we performed 10 simulations in which 200,000 steps were executed. The table shows that a larger sensor range than 10 does not need to lead to better performance, although the smallest sensor range of 5 performs worst. The reason why very large sensors do not work better, is that spiders move around, so it is hard to compute a path for a large number of steps without needing to revise it afterwards. Furthermore, the agent usually focuses on its current path. Thus, a limited sensor range can already perform quite well on its own. This is useful, since a smaller sensor requires much less computational time. Finally, one can observe that the agent still sometimes hits one of the spiders. The reason is that squares are only considered dangerous if they are the neighboring or current square of an observed spider. This can cause the agent to get trapped easily, especially since it performs 10% noisy actions.

Table 5: Results for the system with using the spider model for different sizes of the sensor range. The complete simulation lasts for 200,000 steps. The number of trials is the total number of test and learning trials.

Sensor Range	Goal hits	Spider hits	Trials
With (5)	183±4	16±4	1335±47
With (10)	190±5	9±5	1409±39
With (15)	192±3	7±3	1362±103
With (20)	192±2	7±2	1423±64
With (25)	191±2	8±2	1518±52

If we compare it to using the spider model from the previous experiments (132 goal hits - 37 spider hits) in 200 trials versus (190 goal hits - 9 spider hits) in 200 trials with the sensor of range 10, we can see that the limited sensor performs much better than the a-priori reasoning module. The first problem was much harder, however, since in the previous experiments spiders could not be seen at all, and the agent purely had to rely on previous encounters with a spider or spider nest. Still, both methods perform much better than not using instantiating information at all (85 goal hits - 112 spider hits).

5 Discussion

POMDPs. Path-planning problems in environments with dynamic obstacles are partially observable Markov decision problems (POMDPs), since the transition and reward functions are non-stationary, and there is uncertainty about the true state of the world. Usually POMDPs are solved by using a belief vector

which models the probabilities an agent is in each of the possible states. In case of an environment with dynamic agents, we can use a belief vector modelling probabilities of being in each possible world (with locations of other agents). Solving POMDPs exactly can be done by particular dynamic programming algorithms (Lovejoy, 1991; Kaelbling et al., 1998; Littman, 1996) which compute the best action given each possible belief vector. However, this approach is intractable when the number of possible worlds is quite large (as in our second experimental environment).

Using the underlying MDP. There exist a number of heuristic algorithms trying to find sub-optimal solutions to POMDPs more quickly. The most relevant to our current algorithm is the Q_{MDP} value method (Littman et al., 1995). Here, first the MDP is solved, and then the optimal action is selected by computing the sum of the Q-values of possible states times the occupancy probabilities. This algorithm can perform very well (Littman et al., 1995), but is not able to perform actions to obtain information.

Our approach. We model the POMDP using a single MDP (possible world). Although the dynamic agents may be at different places, and in reality there are multiple possible worlds, we use the certainty equivalence assumption and set transition probabilities to account for all possible worlds. In this way we can use dynamic programming techniques on the single world, otherwise we would need to solve each possible world, which would become quickly intractable. As with Q_{MDP}, our method does not take into account that actions can be used for gaining information about the environment. In principle, the MDP is unchanged as long as no additional information is acquired.

Computing information values. We can extend our algorithm so that information gains can be computed and used by the agent. We can compute the information value of going to a state by instantiating the possible outcomes of an observation received in this state in the MDP. Given one such possible instantiation, our current policy would obtain a reward which can be computed by policy evaluation. By taking into account the instantiated information and recomputing the policy afterwards (by value iteration), we would receive the reward received with the optimal policy given the observation. By subtracting the value of the current policy (found by policy evaluation) of the value of the optimal policy (found by value iteration) and weighing these values over all possible observations, we can compute the information value of going to this state. This will be 0 if no change to the policy is made, and large if the current policy would behave quite bad compared to the optimal policy. Then, this information value can be instantiated in the reward function for this state, and the agent can act to gain information. Unfortunately this becomes intractable if the agent wants to explore sequences of observations.

Dual Control. Dayan and Sejnowski (1996) focus on the exploration problem in which barriers may block the shortest path with some probability. They also changed the transition and reward functions to account for the dynamic probabilities of the existence of each barrier. After this, they used DP to compute a new policy. Although their approach is similar, our algorithm was designed for modelling dynamic agents moving around in the environment and was made much more efficient by using prioritized sweeping. Our algorithm can

also instantiate information acquired by sensors, communication, or reasoning in the transition and reward functions, so that the approach is more general.

Instantiating Information. Instantiating information is a very useful procedure for dealing with dynamic environments such as the Wumpus environments or multi agent systems. We also used instantiating information in (Wiering, 2000) where traffic light controllers communicated with each other and with cars to determine paths through the traffic network containing the least number of waiting cars. Here, the number of cars waiting at a next traffic light was communicated and instantiated to compute the probabilities of ending up at a specific place in a queue of cars at the next traffic light.

Dynamic Replanning. Multiple researchers have designed dynamic replanning algorithms. Most relevant to our research is the D^* algorithm (Stentz, 1995), which uses A^* planning in a dynamic way and a focusing technique to backpropagate the effects of changed parts of the environment. Stentz ran experiments in deterministic 100×100 and 1000×1000 mazes and found a large improvement for only backpropagating partial state-update values which may change the agent's plan. His method used an heuristic to find the goal, however, and cannot deal with probabilistic information.

Using a-priori knowledge. A-priori knowledge can be used with RL in different ways. It can be used for constructing the initial behavior to quickly generate useful learning experiences. It can also be used for designing the structure of a function approximator, so that instead of having to solve both the structural and temporal credit assignment problems, only the temporal credit assignment problem has to be solved. A priori knowledge can be used for modelling the decision process or as a model for solving POMDPs. In this paper we have studied a new way of using a model of dynamic hostile agents and compared it to using limited sensors.

6 Conclusion

We developed a new adaptive dynamic replanning method using reinforcement learning. Our method can learn a model of the environment, and replan if it observes that the environment has changed. The method uses model-based reinforcement learning and instantiates dynamic information about the environment in the model so that the agent can reason about the current environmental state. For efficiency reasons, we used prioritized sweeping to recompute the policy. Our method was successfully tested on maze problems with partially observable dynamic obstacles. We first used an a-priori reasoning module to reason about possible locations of the hostile spider agents. This method was shown to be very effective in avoiding hitting hostile agents in a partially observable path-planning problem. Additional experiments show that the complexity of the algorithm scales well with the number of items which is instantiated in the model. Furthermore, they also show that our method can be combined effectively with a limited sensor for observing hostile agents.

Future work. For very fast changing environments, we may need to include time in the state description (Boyan and Littman, 2001), and our current

method may need too much computation. Therefore we need to make Prioritized Sweeping's update management smarter, taking into account the position and plan of the agent. Then, we want to implement and test our novel algorithm for computing information gains for more effective exploration and dual control.

References

Baxter, J., Tridgell, A., and Weaver, L. (1997). Knightcap: A chess program that learns by combining TD(λ) with minimax search. Technical report, Australian National University, Canberra.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press, Princeton, New Jersey.

Boyan, J. and Littman, M. (2001). Exact solutions to time-dependent MDPs. In *Neural Information Processing Systems (in press).* MIT Press.

Crites, R. H. and Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D., Mozer, M., and Hasselmo, M., editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge MA. MIT Press.

Dayan, P. and Sejnowski, T. J. (1996). Exploration bonuses and dual control. *Machine Learning*, 25:5–22.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134.

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

Littman, M. L. (1996). *Algorithms for Sequential Decision Making.* PhD thesis, Brown University, Providence, Rhode Island.

Littman, M. L. and Boyan, J. A. (1993). A distributed reinforcement learning scheme for network routing. In Alspector, J., Goodman, R., and Brown, T., editors, *Proceedings of the First International Workshop on Applications of Neural Networks to Telecommunication*, pages 45–51, Hillsdale, New Jersey.

Littman, M. L., Cassandra, A. R., and Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scaling up. In Prieditis, A. and Russell, S., editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 362–370. Morgan Kaufmann Publishers, San Francisco, CA.

Lovejoy, W. S. (1991). A survey of algorithms methods for partially observable Markov decision processes. *Annals of Operations Research*, 28:47–66.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229.

Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* The MIT press, Cambridge MA, A Bradford Book.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337.
http://www.cs.brandeis.edu/~aeg/papers/tan.ML93.ps.

Tesauro, G. (1992). Practical issues in temporal difference learning. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 259–266. San Mateo, CA: Morgan Kaufmann.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, England.

Wiering, M. A. (1999). *Explorations in Efficient Reinforcement Learning.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands.

Wiering, M. A. (2000). Multi-agent reinforcement learning for traffic light control. In Langley, P., editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 1151–1158.

Wiering, M. A. and Schmidhuber, J. H. (1998). Efficient model-based exploration. In Meyer, J. A. and Wilson, S. W., editors, *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats 6*, pages 223–228. MIT Press/Bradford Books.