Planning in the Presence of Cost Functions Controlled by an Adversary

H. Brendan McMahan MCMAHAN@CS.CMU.EDU
Geoffrey J Gordon GGORDON@CS.CMU.EDU
Avrim Blum AVRIM@CS.CMU.EDU
Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213 USA

Abstract

We investigate methods for planning in a Markov Decision Process where the cost function is chosen by an adversary after we fix our policy. As a running example, we consider a robot path planning problem where costs are influenced by sensors that an adversary places in the environment. We formulate the problem as a zero-sum matrix game where rows correspond to deterministic policies for the planning player and columns correspond to cost vectors the adversary can select. For a fixed cost vector, fast algorithms (such as value iteration) are available for solving MDPs. We develop efficient algorithms for matrix games where such best response oracles exist. We show that for our path planning problem these algorithms are at least an order of magnitude faster than direct solution of the linear programming formulation.

1. Introduction and Motivation

Imagine a robot in a known (previously mapped) environment which must navigate to a goal location. We wish to choose a path for the robot that will avoid detection by an adversary. This adversary has some number of fixed sensors (perhaps surveillance cameras or stationary robots) that he will position in order to detect our robot. These sensors are undetectable by our robot, so it cannot discover their locations and change its behavior accordingly. What path should the robot follow to minimize the time it is visible to the sensors? Or, from the opponent's point of view, what are the optimal locations for the sensors?

We assume that we know the sensors' capabilities. That is, given a sensor position we assume we can calculate what portion of the world it can observe. So, if we know where our opponent has placed sensors, we can compute a cost vector for our MDP: each entry represents the number of sensors observing a particular world state. In that case, we can apply efficient planning algorithms (value iteration in stochastic environments, A* search in deterministic environments) to find a path for our robot that minimizes its total observability. Of course, in reality we don't know where the sensors are; instead we have a set of possible cost vectors, one for each allowable sensor placement, and we must minimize our expected observability under the worst-case distribution over cost vectors.

In this paper we develop efficient algorithms to solve problems of the form described above, and we experimentally demonstrate the performance of these algorithms. In particular, we use Benders' decomposition (Benders, 1962) to capitalize on the existence of best-response oracles like A* and value iteration. We generalize this technique to the case where a best-response oracle is also available for the adversary. Before developing these algorithms in detail, we discuss different ways to model the general problem we have described, and discuss the variation we solve.

Our algorithms are practical for problems of realistic size, and we have used our implementation to find plans for robots playing laser tag as part of a larger project (Rosencrantz et al., 2003). Figure 1 shows the optimal solutions for both players for a particular instance of the problem. The map is of Rangos Hall at Carnegie Mellon University, with obstacles corresponding to overturned tables and boxes placed to create an interesting environment for laser tag experiments. The optimal strategy for the planner is a distribution over paths from the start (s) to one of the goals (g), shown in 1A; this corresponds to a mixed strategy in the matrix game, that is, a distribution over

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

the rows of the game matrix. The optimal strategy for the opponent is a distribution over sensor placements, or equivalently a distribution over the columns of the game matrix. This figure is discussed in detail in Section 4.

2. Modeling Considerations

There are a number of ways we could model our planning problem. The model we choose, which we call the *no observation, single play formulation*, corresponds to the assumptions outlined above. The opponent is restricted to choosing a cost vector from a finite though possibly large set. The planning agent knows this set, and so constructs a policy that optimizes worst-case expected cost given these allowed cost vectors. Let Π_D be the set of deterministic policies available to the planning agent, let $K = \{c_1, \ldots, c_k\}$ be the set of cost vectors available to the adversary, and let $V(\pi, c)$ be the value of policy $\pi \in \Pi_D$ under cost vector $c \in K$. Let $\Delta(\cdot)$ indicate the set of probability distributions over some set. Then our algorithms find the value

$$\min_{p \in \Delta(\Pi_D)} \max_{q \in \Delta(K)} E_{\pi \sim p, c \sim q}[V(\pi, c)],$$

along with the distributions p and q that achieve this value. In the next section we discuss the assumptions behind this formulation of the problem in more detail, and examine several other possible formulations.

2.1. Possible Problem Formulations

Our most limiting assumption is that our planning agent cannot observe the adversary's affect on the cost vector. In our example domain, the robot incurs fixed, observable costs for moving, running into objects, etc.; however, it cannot determine when it is being watched and so it cannot determine the cost vector selected by the adversary. This is a reasonable assumption for some domains, but not others. If the assumption does not hold, our algorithms will produce suboptimal policies: for example, we would not be able to plan to check whether a path was being watched before following it.

The no-observation assumption, while sometimes unrealistic, is what allows us to find efficient algorithms. Without this assumption, the planning problem becomes a partially-observable Markov decision process: the currently-active cost vector is the hidden state and the costs incurred are observations. POMDPs are known to be difficult to solve (Kaelbling et al., 1996); on the other hand, the planning problem without observations has a poly-time solution.

In addition to the POMDP formulation, our problem can also be framed in an online setting where the MDP must be solved multiple times for different cost vectors. The planning agent must pick a policy for the nth game based on the cost vectors it has seen in the first $n-1$ games. The goal is to do well in total cost, compared to the best fixed policy against the opponent's sequence of selected cost vectors. To obtain tractable algorithms we still make the no-observation assumption, but it is not necessary to assume the opponent chooses cost vectors from a finite set. When this formulation is applied to shortest path problems on graphs, it is the online shortest path problem for which some efficient algorithms are already known (Takimoto & Warmuth, 2002). We hope to explore this formulation in more detail in the future.

It is worth noting the relationship between our problem and stochastic games (Filar & Vrieze, 1992). Our setting is more general in some ways and less general in others: we allow hidden state (the cost function), but stochastic games allow players to make a sequence of interdependent moves while we require both players to select their policies simultaneously at the outset.

Our algorithm is similar in approach to Bagnell et al. (2001), but in that work the hidden information is the exact dynamics model, not the cost function.

2.2. Example Problem Domains

In this section we describe some additional domains where our formulation is useful, and also some different interpretations of the model.

In general, the no-observation assumption is applicable in two cases: when observations are actually impossible, and when observations are possible, but once they have been made there is nothing to be done. The way we initially phrased our robot path-planning problem, it falls in the first case: the sensors cannot be detected. On the other hand, if we can detect a sensor but have already lost the game once we detect it, the problem falls in the second case.

It is easy to show that having the opponent initially pick some cost vector from a distribution q is equivalent in expectation to letting the opponent independently pick a cost vector from q at each timestep. Consider again the question of monitoring an area with security cameras, but suppose the cameras have already been placed. Further, suppose we have only a single video monitor so the surveillance operator can observe the output from only one camera at each timestep. The operator should choose a camera to monitor from distribution q at each timestep in order to maximize the time that an intruder is observed.

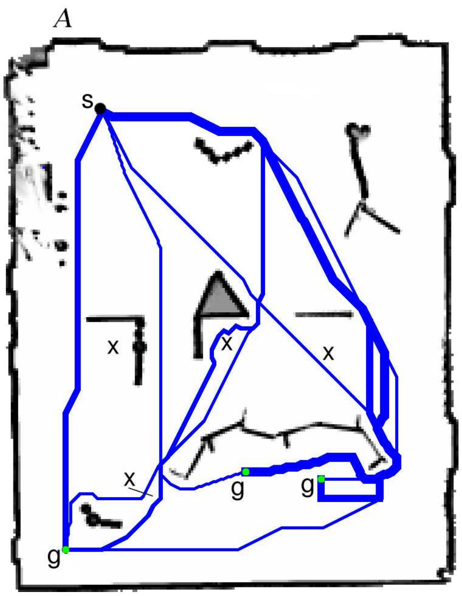

Figure 1. Planning in a robot laser tag environment. Part A: A mixture of optimal trajectories for a robot traveling from start location (s) to one of three goals (g). The opponent can put a sensor in one of 4 locations (x), facing one of 8 directions. The widths of the trajectories correspond to the probability that the robot takes the given path. Parts B,C,D,E: The optimal opponent strategy randomizes among the sensor placements that produce these four fields of view.

So far we have imagined an adversary selecting one cost vector from a set of cost vectors; however, our formulation applies to the case where the actual cost is given by the highest cost of the chosen policy with respect to any of the cost vectors. For example, suppose there is a competition to control a robot performing an industrial welding task. In the first round the robots will be evaluated by three human judges, each of which has the ability to remove a robot from consideration. It is known that one judge will prefer faster robots, another will be more concerned with the robots' power consumption, and another with the precision with which the task is performed. If the task is formulated as an MDP, then each judge's preference can be turned into a cost vector, and our algorithm will find the policy that maximizes the lowest score given by any of the three judges. The policy calculated will be optimal if all three judges evaluate the policy, or if an adversary picks a distribution from which only one judge will be chosen.

In general, our techniques apply any time we have a set of cost vectors K, a set of policies Π_D, a fast algorithm to solve the problem given a particular cost vector $c \in K$, and we can randomize over the set of policies Π_D. Here the term "policy" means only a possible solution to the given problem. We now proceed with some background on MDPs and then present our algorithms for solving these problems.

3. Background

Consider an MDP M with a state set $\mathcal{S}$ and action set $\mathcal{A}$. The dynamics for the MDP are specified for all $s, s' \in \mathcal{S}$ and $a \in A$ by $\mathcal{P}^a_{ss'}$, the probability of moving to state s' if action a is taken from state s. In order to express problems regarding MDPs as linear programs, it is useful to define a matrix E as follows: E has one row for every state-action pair and one column per state. The entry for row (s, a) and column s' contains $\mathcal{P}^a_{ss'}$ for all $s \neq s'$, and $\mathcal{P}^a_{ss} - 1$ for $s = s'$. A cost function for the MDP can be represented as a vector c that contains one entry for each state-action pair (s, a) indicating the cost of taking action a in state s. A stochastic policy for an MDP is a mapping $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, so that $\pi(s, a)$ gives the probability an agent will take action a in state s. Thus, for all s we must have $\sum_{a \in A} \pi(s, a) = 1$. A deterministic policy is one that puts all probability on a single action for each state, so that it can be represented by $\pi : \mathcal{S} \rightarrow \mathcal{A}$. The Markov assumption implies that we do not need to consider history[1] dependent policies; the policies we consider are stationary, in that they depend only on the current state. For an MDP with a fixed cost function c there is always an optimal deterministic

[1] We assume the standard definition of the history, where it contains only states and actions. If costs incurred appear in the history then our formulation does not apply, as we are in the POMDP case.

policy, and so stochastic policies play a lesser role. In our adversarial formulations, however, optimal policies may be stochastic.

We are primarily concerned with undiscounted shortest path optimality: that is, all states have at least one finite-length path to a zero-cost absorbing state, and so undiscounted costs can be used. Our results can be adapted to discounted infinite horizon problems by multiplying all the probabilities $\mathcal{P}^a_{ss'}$ by a discount factor γ when the matrix E is formed. The results can also be extended to an average reward model, but this requires slightly more complicated changes to the linear programs introduced below.

There are two natural representations of a policy for a MDP, one in terms of frequencies and another in terms of total costs or values. Each arises naturally from a different linear programming formulation of the MDP problem. For any policy we can compute a cost-to-go function,[2] $v_\pi : \mathcal{S} \to \mathbb{R}$, that associates with each state the total cost $v_\pi(s)$ that an agent will incur if it follows π from s for the rest of time. If π is optimal then the policy achieved by acting greedily with respect to v_π is optimal.

The optimal value function for an MDP with cost vector c and fixed start state[3] s can be found by solving the following linear program:

$$\max_v v(s) \;\; \text{subject to} \qquad (1)$$
$$Ev + c \geq 0.$$

The set of constraints $Ev + c \geq 0$ is equivalent to the statement that $v(s) \leq c(s,a) + \sum_{s' \in S} \mathcal{P}^a_{ss'} v(s')$ for all $s \in S$ and $a \in A$.

Fixing an arbitrary stochastic policy π and start state s uniquely determines a set of state-action visitation frequencies $f_s(\pi)$. We omit the s when it is clear from context. The dual of (1) is the linear program whose feasible region is the set of possible state-action visitation frequencies, and is given by

$$\min_f f \cdot c \;\; \text{subject to} \qquad (2)$$
$$E^T f + \vec{s} = 0, \quad f \geq 0$$

where $\vec{s}$ is the vector of all zeros except for a 1 in the position for state s. The constraints $E^T f + \vec{s} = 0$ require that the sum of all the frequencies into a state x equal the sum of all the frequencies out of x. The objective $f \cdot c$ represents the value of the starting state under the policy π which corresponds to f. For any cost vector c' we can compute the value of π as $f(\pi) \cdot c'$. We also use the fact that every stochastic policy (when represented as state-action visitation frequencies) can be represented as a convex combination of deterministic policies, and every convex combination of deterministic policies corresponds to some stochastic policy. A detailed proof of this fact along with a good general introduction to MDPs can be found in (Puterman, 1994, Sec. 6.9).

[2] The term cost-to-go function is more natural when we think of costs for actions; the term value function is used when rewards are specified.

[3] This can easily be generalized to a start-state distribution.

4. Solving the Single-Play Formulation

Suppose we have an MDP with known dynamics and fixed start state s, but an adversary will select the cost function from $K = \{c_1, \ldots, c_k\}$. We can formulate the problem as a zero-sum matrix game: the possible pure strategies for the planning player (our robot) are the set of stationary deterministic policies Π_D for the MDP, and the pure strategies for the opponent are the elements of K (possible sensor configurations). Note that the game matrix is finite because, by assumption, K is finite, and there are a finite number of stationary deterministic policies for a given MDP. If we fix a policy π_i and cost vector c_j, then the value of the game is $f(\pi_i) \cdot c_j$.

A mixed strategy given by a distribution $p : \Pi_D \to [0, 1]$ for the row player is equivalent to the stochastic policy with state-action visitation frequencies $f = \sum_{\pi \in \Pi_D} f(\pi) p(\pi)$. Similarly, a mixed strategy $q : K \to [0, 1]$ for the column player corresponds to a point in the convex set Q whose corners are the elements of K.

Let Π be the set of all mixed strategies (stochastic policies). Our goal is to find a $\pi^* \in \Pi$ and $c^* \in Q$ such that

$$\min_{\pi \in \Pi} \max_{c \in Q} f(\pi) \cdot c = \max_{c \in Q} \min_{\pi \in \Pi} f(\pi) \cdot c = f(\pi^*) \cdot c^*.$$

That is, we wish to find a minimax equilibrium for a two player zero sum matrix game.

We can extend (2) in a straightforward way to solve this problem: we simply introduce another variable z which represents the maximum cost of the policy f under all possible opponent cost vectors. Letting C be the matrix with columns $c_1, \ldots, c_k$, we have:

$$\min_{z,f} z \;\text{subject to} \qquad (3)$$
$$E^T f + s = 0$$
$$\mathbf{1} \cdot z + C^T f \leq 0, \quad f \geq 0$$

The primal variables f of (3) give an optimal mixed strategy for the planning player. Taking the dual of (3), we have

$$\max_{v,q} v \cdot s \quad \text{subject to} \tag{4}$$
$$Ev + Cq \geq 0,$$
$$\mathbf{1} \cdot q = 1, \quad q \geq 0$$

where q gives the optimal mixture of costs for the adversary, and v is a cost-to-go function when playing against this distribution. Note that v induces a deterministic policy that gives a best response if the opponent fixes the distribution q over cost vectors, but in general this pair of strategies will not be a minimax equilibrium.

Figure 1 shows a solution to the robot path planning problem formulated in this way. The left portion, A, shows the minimax strategy for the planner, that is a distribution p over the start - goal paths that make up Π_D for this problem. The four right-hand panels, B, C, D, and E correspond to elements of K, cost vectors shown as the fields of view of sensor placements. These four are the most likely cost vectors selected by the opponent; they receive weights 0.18, 0.42, 0.11, and 0.28 respectively. The remainder of the probability mass is on other sensor placements.

4.1. An Algorithm using Benders' Decomposition

Our iterative algorithm is an application of a general method for decomposing certain linear programs first studied by Benders (1962). We focus on the application of this technique to the problem at hand, and refer the reader to other sources for a more general introduction (Bazaraa et al., 1990).

Our algorithm is applicable when we have an oracle $\mathcal{R} : Q \rightarrow \Pi$ that for any cost vector c provides an optimal policy π. The algorithm requires that π be represented by its state-action frequency vector $f(\pi)$, and so we write $\mathcal{R}(c) = f$ for $c \in Q$. If the actual oracle algorithm provides a policy as a cost-to-go function we can calculate $f(\pi)$ with a matrix inversion or by iterative methods.

If we let $\theta(q)$ be the optimal value of (1) for a fixed q and $c = Cq$, then we can rewrite (4) as the program

$$\max_{q} \theta(q) \quad \text{subject to} \tag{5}$$
$$\mathbf{1} \cdot q = 1, q \geq 0$$

Unfortunately, $\theta(q)$ is not linear so we cannot solve this program directly. However, it can be solved via a convergent sequence of approximations that capitalize on the availability of our oracle $\mathcal{R}$.

The algorithm performs two steps for each iteration. On iteration i, first we solve for an optimal mixture of costs q_i under the assumption that the planner is only allowed to select a policy from a restricted set $F = \{\pi_1, \pi_2, \ldots, \pi_i\}$. Then, we use our oracle to compute $\mathcal{R}(Cq_i) = \pi_{i+1}$, an optimal deterministic policy with respect to the fixed cost vector $c = Cq_i$. The policy π_{i+1} is added to F, and these steps are iterated until an iteration where π_{i+1} is already in F.

Let v be the value of (2). Given a policy $\pi \in F$, for any cost mixture q' we have the bound $\theta(q') \leq f(\pi) \cdot Cq'$, the value of the game when we always play π. Thus, each π gives an upper bound $\forall q',\ v \leq f(\pi) \cdot Cq'$. Similarly, fixing a cost mixture q, we know $\mathcal{R}(q) = f$ is the optimal response, and thus we have a bound $\forall f',\ \theta(q) \geq f' \cdot Cq$, because we cannot do better against q than playing f. Thus, $v \geq f \cdot Cq$. In practice we check for the convergence of the algorithm by monitoring these two bounds, allowing us to halt on any iteration with a known bound on how far the current solution might be from v.

All that remains is to show how to find the optimal cost mixture q_i given that the planner will select a policy from the set $F = \{f(\pi_1), f(\pi_2), \ldots, f(\pi_i)\}$ of feasible solutions to (2). This problem can be solved with the linear program

$$\max_{q} \theta \quad \text{subject to} \tag{6}$$
$$\theta \leq f(\pi_j)Cq \quad \text{for } 1 \leq j \leq i,$$

which is essentially the same program as (3), where f is restricted to be a member of F rather than an arbitrary stochastic policy. It is also the linear program for solving the matrix game that is defined by the submatrix of the overall game containing only the rows corresponding to elements of F. This is the "master" program of the Benders' decomposition.

The algorithm can be summarized as follows:

- Pick an initial value for q, say the uniform distribution, and let $F = \{\}$.
- Repeat:
  - Use the row oracle $\mathcal{R}$ to find an optimal policy f for the fixed cost vector $c = Cq$. If $f \in F$, halt. Otherwise, add f to F and continue.
  - Solve the linear program (6), and let q be the new optimal cost mixture determined.

The optimal cost mixture is given by the q found in the last iteration. The optimal policy for the planning player can be expressed as a distribution over the policies in F. These values are given by the dual variables of (6). The convergence and correctness of this algorithm are immediate from the corresponding proofs for Benders' decomposition. We show in Section 5 that the sequence of values converges quickly in practice. We refer to this algorithm as the Single Oracle algorithm because it relies only on a best-response oracle for the row player.

4.2. A Double Oracle Algorithm

The above algorithm is sufficient for problems when the set K is reasonably small; in these cases solving the master problem (6) is fast. For example, we use this approach in our path planning problem if the opponent is confined to a small number of possible sensor locations and we know that he will place only a single sensor. However, suppose there are a relatively large number of possible sensor locations (say 50 or 100), and that the adversary will actually place 2 sensors. If the induced cost function assigns an added cost to all locations visible by one or more of the sensors, then we cannot decouple the choice of locations, and so there will be $\binom{100}{2}$ possible cost vectors in K. The Single Oracle algorithm is not practical for a problem with this many cost vectors; simply the memory cost of representing them all would be prohibitive.

We now derive a generalization of the Single Oracle algorithm that can take advantage of best response oracles for both the row and column players. We present this algorithm as it applies to arbitrary matrix games. Let M be a matrix defining a two player zero-sum matrix game. An entry $M(r, c)$ is the cost to the row player given that the row player played row r and the column player played c. Let V_G be the value of the game, and let $V_M(p, q)$ be the value of the game under strategies p and q. We omit the M when it is clear from context.

Our algorithm assumes the existence of best response oracles $\mathcal{R}$ and $\mathcal{C}$ for the row and column player respectively. That is, given a mixture q on the columns of M, $\mathcal{R}(q) = r$ computes the row r of M that is a best response, and given a mixture p on the rows of M, $\mathcal{C}(p) = c$ computes a column c of M that is a best response. For an arbitrary matrix such an oracle may be achieved (say for the row player) by finding the minimum entry in Mq. Perhaps surprisingly, we show that even using such a naive oracle can yield performance improvements over the Single Oracle algorithm. If sensor fields of view have limited overlap, then a fast best response oracle for multiple sensor placement can also be constructed by considering each sensor independently and using the result to bound the cost vector for a pair of sensors.

The algorithm maintains sets $\bar{R}$ of all strategies the row player has played in previous iterations of the algorithm (this set corresponds to the set F from the previous algorithm), and similarly maintains a set $\bar{C}$ of all the columns played by the column player. We initialize $\bar{R}$ with an arbitrary row, and $\bar{C}$ with an arbitrary column. The algorithm then iterates over the following steps. On iteration i:

- Solve the matrix game where the row player can only play rows in $\bar{R}$ and the column player can only play columns of $\bar{C}$, using linear programming or any other convenient technique. This provides a distribution p_i over $\bar{R}$ and q_i over $\bar{C}$.
- The row player assumes the column player will always play q_i, finds an optimal pure strategy $\mathcal{R}(q_i) = r_i$ against q_i, and adds r_i to $\bar{R}$. Let $v_\ell = V(r_i, q_i)$. Since r_i is a best response we conclude that $\forall p\ V(p, q_i) \geq v_\ell$, and so we have a bound on the value of the game, $V_G = \min_p \max_q V(p, q) \geq v_\ell$.
- Similarly, the column player picks $\mathcal{C}(p_i) = c_i$, and adds c_i to $\bar{C}$. We let $v_u = V(p_i, c_i)$ and conclude $\forall q\ \ V(p_i, q) \leq v_u$, and hence $V_G = \max_q \min_p \leq v_u$.

The bounds given above are a direct consequence of von Neumann's minimax theorem and the definition of the value of a game. If on some iteration i, r_i is already in $\bar{R}$ and $c_i \in \bar{C}$, then the algorithm terminates. As in the Single Oracle algorithm, it will be more convenient to simply iterate until $v_u - v_\ell < \epsilon$, where ϵ is a parameter.

Theorem 1 *The Double Oracle algorithm converges to a minimax equilibrium.*

Proof: Convergence of the algorithm is immediate because we assume a finite number of rows and columns. (Eventually $\bar{R}$ and $\bar{C}$ include all rows and columns of M, and the corresponding linear program is simply the linear program for the whole game.) Now we prove correctness. Suppose on iteration j we add neither a new row nor a new column. Since the algorithm does not add a new row to $\bar{R}$ then it must be the case that $v_\ell = V(p_j, q_j)$, and similarly $v_u = V(p_j, q_j)$, so $v_\ell = v_u$. It suffices to show p_j is a minimax optimal solution; the fact that q is a maximin optimal solution follows from an analogous argument. Let $v = v_\ell = v_u$. Since $\forall p\ V(p, q_j) \geq v$, we know $\forall p\ \ \max_q V(p, q) \geq v$. Since $\forall q\ V(p_j, q) \leq v$ we know

Table 1. Sample problem discretizations, number of sensor placements available to the opponent, solution time using Equation 4, and solution time and number of iterations using the Double Oracle Algorithm.

	grid size	k	LP	Double	iter
A	54 x 45	32	56.8 s	1.9 s	15
B	54 x 45	328	104.2 s	8.4 s	47
C	94 x 79	136	2835.4 s	10.5 s	30
D	135 x 113	32	1266.0 s	10.2 s	14
E	135 x 113	92	8713.0 s	18.3 s	30
F	269 x 226	16	-	39.8 s	17
G	269 x 226	32	-	41.1 s	15

$\max_q V(p_j, q) \leq v$. Combining these two facts, we conclude $\forall p,\ \max_q(p_j, q) \leq \max_q(p, q)$, and so p_j is minimax optimal. □

If the algorithm ends up considering all of the rows and columns of M, it will be slower than just solving the game directly. However, it is reasonable to expect in practice that many pure strategies will never enter $\bar{R}$ and $\bar{C}$. For example, in our path-planning example we may never need to consider paths which take the robot far out of its way, and we certainly never need to consider paths with possible loops in them. We might be able to remove these bad pure strategies ahead of time in an ad-hoc way, but this algorithm gives us a principled approach.

5. Experimental Results

We model our robot path planning problem by discretizing a given map at a resolution of between 10 and 50 cm per cell, producing grids of size 269×226 to 54×45. We do not model the robot's orientation and rely on lower level navigation software to move the robot along planned trajectories. Each cell corresponds to a state in the MDP.

The transition model we use gives the robot 16 actions, corresponding to movement in any of 16 compass directions. Each state s has a cost weight $m(s)$ for movement through that cell; in our experiments all of these are set to 1.0 for simplicity. The actual movement costs for each action are calculated by considering the distance traveled (either 1, $\sqrt{2}$, or $\sqrt{5}$) weighted by the movement costs assigned to each cell. For example, movement in one of the four cardinal directions from a state u to a state v incurs cost $0.5m(u) + 0.5m(v)$.

Cells observed by a sensor have an additional cost given by a linear function of the distance to the sensor. An additional cost of 20 is incurred if observed by an adjacent sensor, and cost 10 is incurred if the sensor is at the maximum distance. The ratio of movement cost to observation cost determines the planner's relative preference for paths that provide a low probability of observation versus shortest paths. We assume a fixed start location for our robot in all problems, so pure strategies can be represented as paths.

We have implemented both the Single Oracle algorithm and the Double Oracle algorithm and applied them to this domain. Both algorithms use Dijkstra's algorithm as the row oracle, and the column oracle for the Double Oracle algorithm is the naive method outlined in Section 4.2. Our implementation is in Java, with an external call to CPLEX 7.1 (ILOG, 2003) for solving all linear programs. For comparison, we also used CPLEX to solve the linear program 4 directly (without any decompositions).

All results given in this paper correspond to the map in Figure 1. We performed experiments on other maps as well, but we do not report them because the results were qualitatively similar. We solved the problem using various discretizations and different numbers of potential cost vectors to demonstrate the scaling properties of our algorithms. These problem discretizations are shown in Table 5, along with their Double Oracle and direct LP solution times. The letters in the table correspond to those in Figure 2, which compares the Double and Single Oracle algorithms. All times are wall-clock times on a 1 GHz Pentium III machine with 512M main memory. Results reported are the average over 5 runs. Standard deviations were insignificant, less than 1/10th of total solve time in all cases. For the Single and Double Oracle algorithms, non-algorithmic overhead involved in calling CPLEX is not included in these times; an implementation in C or using a version of CPLEX with Java bindings could completely bypass this overhead.

Our results indicate that both the Double and Single Oracle algorithms significantly outperform the direct LP algorithm. This improvement in performance is possible because our algorithms take advantage of the fact that 4 is "almost" an MDP: the row oracle is implemented with Dijkstra's algorithm, which is much faster than general LP solvers. The particularly lopsided times for problems C, D, and E were partly caused by CPLEX running low on physical memory; we didn't try solving the LPs for problems F and G because they are even larger. One of the benefits of our decomposition algorithms is their lower memory usage (they never need to construct the LP 4 explicitly), but even when memory was not an issue our algorithms were significantly faster than CPLEX.

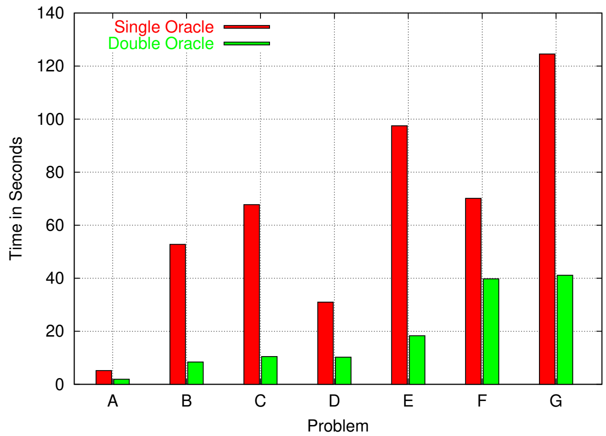

Figure 2. Double and Single Oracle algorithm performance on problems shown in Table 5.

As Figure 2 shows, the Double Oracle algorithm outperforms the Single Oracle version for all problems. The difference is most pronounced on problems with a large number of cost vectors. The time for solving the master LPs and for the column oracle are insignificant, so the performance gained by the Double Oracle algorithm is explained by its implicit preference for mixed strategies with small support. With small support we need to represent fewer cost vectors, and explicit representation of cost vectors can be a significant performance bottleneck. For example, in Problems F and G each cost vector has length $|S||A| = 269 \cdot 226 \cdot 16$, so storing 32 cost vectors uses approximately 236M of memory.

6. Conclusions and Future Work

We have described the problem of planning in an MDP when costs are influenced by an adversary, and we have developed and tested efficient, practical algorithms for our formulation of the problem.

The most important missing element is the ability to plan for future observations. It would be interesting to see if there is a formulation of the problem that allows observations of costs to be accounted for without requiring a full POMDP solution. We also hope to investigate the online version of this problem in more detail. Finally, we intend to experiment with even faster row oracles, since row oracle calls take 90% or more of the total run time in our experiments: for example, instead of Dijkstra's algorithm we could use an incremental A^* implementation (Koenig & Likhachev, 2001).

Acknowledgments

We would like to thank the reviewers for a number of helpful suggestions. This work was supported in part by NSF grant CCR-0105488 NSF-ITR CCR-0122581 and by AFRL contract F30602–01–C–0219, DARPA's MICA program. The opinions and conclusions are the authors' and do not reflect those of the US government or its agencies.

References

Bagnell, J., Ng, A. Y., & Schneider, J. (2001). *Solving uncertain markov decision problems* (Technical Report CMU-RI-TR-01-25). Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Bazaraa, M. S., Jarvis, J. J., & Sherali, H. D. (1990). *Linear programming and network flows*. John Wiley & sons.

Benders, J. F. (1962). Partitioning procedures for solving mixed-variable programming problems. *Numerische Mathematik*, *4*, 238–252.

Filar, J., & Vrieze, O. (1992). Weighted reward criteria in competitive markov decision processes. *ZOR - Methods and Models of Operations Research*, *36*, 343 – 358.

ILOG (2003). CPLEX optimization software. `http://www.ilog.com/products/cplex/index.cfm`.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1996). *Planning and acting in partially observable stochastic domains* (Technical Report CS-96-08). Brown University.

Koenig, S., & Likhachev, M. (2001). Incremental A*. *Advances in Neural Information Processing Systems 14*.

Puterman, M. L. (1994). *Markov decision processes: Discrete stochastic dynamic programming*. Willey Interscience.

Rosencrantz, M., Gordon, G., & Thrun, S. (2003). Locating moving entities in dynamic indoor environments with teams of mobile robots. *AAMAS 2003*.

Takimoto, E., & Warmuth, M. K. (2002). Path kernels and multiplicative updates. *Proceedings of the 15th Annual Conference on Computational Learning Theory*. Springer.