Kaleidoscope

Volume 11 Article 19

July 2014

Oswald Physical and Engineering Sciences Honorable Mention: Summer 2012 LIP6 Write Up

Josiah Hanna

Summer 2012 LIP6 Write Up

October 31, 2012

Abstract

Planning under uncertainty is a central problem in developing intelligent autonomous systems. The traditional representation for these problems is a Markov Decision Process (MDP). The MDP model can be extended to a Multi-criteria MDP (MMDP) for planning under uncertainty while trying to optimize multiple criteria. However, due to the trade-offs involved in multi criteria problems there may be infinitely many optimal solutions. The focus of this project has been to find a method that efficiently computes a subset of solutions that represents the entire set of optimal solutions for bi-objective MDPs.

1 Introduction

Much research in the artificial intelligence planning community has focused on planning under uncertainty. These problems have applications from space exploration robots to decision support for investing for retirement. Traditionally, these problems are represented with a mathematical formulation known as a Markov Decision Process (MDP). Formally, an MDP is defined as follows:

Definition 1.1 *A* Markov Decision Process (MDP) *is a tuple* $\langle S, A, T, R \rangle$ *where:* S *is a finite set of states,* A *is a finite set of actions,* $T : S \times A \to Pr(S)$ *is a transition function giving, for each state and action a probability function over* S*, and* $R : S \times A \to \mathtt{R}$*, is a reward function giving the immediate reward for executing a given action in a given state [5].*

For MDPs the transition model is Markovian, meaning the probability of a transition is only affected by the current state and not the state history. The probabilistic transition model also means the environment is nondeterministic and the agent is dealing with uncertainty when planning. A discount factor γ $(0 < \gamma < 1)$ may also be included with the model. This factor enforces that immediate rewards are considered more important than future rewards. Otherwise, for a problem with an infinite horizon the total reward summed up over time would be infinite. The solutions to the problems represented by MDPs are called policies, which are functions mapping an action to each state. The value of a policy is given by the Bellman equation:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s)) V^{\pi}(s').$$

The policy with the best expected utility is the optimum policy [1]. This optimal policy can be found by maximizing the Bellman equation using linear programming or dynamic programming techniques such as value iteration or policy iteration.

2 Multi-Criteria Markov Decision Processes

MDPs can be extended to MMDPs (Multi-criteria Markov Decision Processes) by extending the reward function to map a state-action pair to a reward vector which assigns a scalar reward for each criterion. The value function will also be vector-valued, and the Bellman equation will continue to define the value of a policy. Note that a policy that maximizes on one criterion will not necessarily do the same for another. Some policies will favor one criterion, some the other criterion, and some will be balanced towards both criteria. There may not be a single policy that maximizes both criteria. Therefore to compare policies of MMDPs, the notion of Pareto optimality will be introduced in Section 3.

3 Pareto Optimality

In multi-criteria optimization, one way of comparing solutions is Pareto optimality. Informally, a solution is Pareto optimal if its value for one criterion cannot be increased without the value on another criterion being decreased. To formally describe this, first, Pareto dominance must be defined. Let a_x and b_x represent criterion x for solution vectors a and b respectively and N be the set of all criteria.

$$a \succ_p b \text{ iff } \forall x \exists y \in \mathbb{N}[a_x \geq b_x \wedge a_y > b_y]$$

The set of Pareto optimal solutions can then be defined as the set of all solutions that are not Pareto dominated by another solution. This set of solutions is referred to as the *Pareto front*. It is easy to see that for even two criteria the cardinality of the Pareto front could be infinite. For example, taking one solution in the Pareto front, one criterion's value can be increased while the other is decreased such that the new solution is not dominated by the former and the former is not dominated by the new soution.

Another term that needs to be defined is *supported solution*. A supported solutions is one that is part of the convex hull of the Pareto front. It is possible for a solution to be non-supported and yet still Pareto optimal.

With possibly infinitely many solutions, a method must be found to efficiently represent all solutions. First, ϵ-dominance is defined as [4]:

$$x \succsim_\epsilon y \longleftrightarrow \forall i \in N, x_i(1+\epsilon) \geq y.$$

One such method is to represent the entire set of solutions with a much smaller subset of solutions. This subset consists of a set of solutions such that all other solutions are ϵ-dominated by at least one point in the subset. This subset is defined as an ϵ-cover or ϵ-approximation. For any

set of solutions to a multi-criteria optimization problem there will be multiple subsets that provide an ϵ-cover. It is therefore possible to search not only for an ϵ-cover but also for the cover with minimal cardinality. It is the goal of this research to find an ϵ-cover with approximately minimal cardinality for the set of all Pareto optimal solutions to MMDPs. The experimental part of this paper focuses on bi-objective MMDPs.

4 Finding the Complete Pareto Optimal Set

As a point of comparison for testing methods of finding an ϵ-cover it is necessary to determine the entire set of solutions. The technique is to use a weighted sum of the reward values for each criterion to find all Pareto non-dominated, supported solutions. For the bi-objective case, this method involves finding two solutions and then optimizing in the direction of the normal of the line between the two solution points. The initial step is to find the optimum value function for each criterion. Then a third solution is found in the direction of the normal to the line between the two solutions. This is done with a scalar reward function which is a linear combination of the reward values for each criterion and a vector of weights. The weights can be found from the slope of the normal and are constrained such that the sum of the weights must equal one. After finding the third point the process can then be repeated looking between the new point and one old point and the new point and the other old point. The process continues recursively until no point is found between a pair.

The algorithm can be extended to multiple criteria, normalizing in the direction of a plane formed by three points for three criteria or, more generally, in the direction of a hyperplane determined by n points for n criteria. Value iteration is an efficient optimization technique for this algorithm but other approaches such as linear programming can be used. The pseudocode for the algorithm is shown in Algorithm 1:

Input: m, an MMDP with criteria x,y and rewards R_1, R_2
Output: The set of all supported Pareto Optimal Solutions
p_1 = optimize(R_1);
p_2 = optimize(R_2);
Q = $\{p_1, p_2\}$
$Q = Q \cup$ directedOptimization(p_1, p_2);
return Q;

Algorithm 1: Algorithm for finding all supported Pareto optimal solutions.

The method **optimize** finds the optimal policy for the MDP with the same states, actions, and transitions as the given MMDP but with the given scalar rewards function. The pseudocode for the **directedOptimization** helper method is shown in Algorithm 2:

Input: p_1, p_2 are coordinate points representing values of each criteria
Output: Set of all supported Pareto optimal solutions between p_1, p_2
$y = \frac{(x(p_1)-x(p_2))}{(x(p_1)+y(p_2)-x(p_1)-y(p_1))}$;
$x = \frac{(y(p_1)-y(p_2))}{(x(p_1)+y(p_2)-x(p_1)-y(p_1))}$;
q = optimize($xR_1 + yR_2$);
if q == x **or** q == y **then**
    return {};
end
return directedOptimization(p_1, q) $\cup$ directedOptimization(q, p_2);

Algorithm 2: Pseudocode for the directedOptimization algorithm

4.1 Approximating the Pareto Front

This section presents a simple method for finding an approximation of the Pareto front. This method involves finding solutions using a standard MDP dual linear program. The algorithm uses a parameter, λ, to weight the different reward functions for each criterion. One criterion is weighted λ and the other is 1 - λ for $0 \leq \lambda \leq 1$. The algorithm iteratively decrements (or increments) λ by a fixed step so that the solutions progressively decrease in value for one criterion while increasing on the other. This method does not find all the solutions (some are always possibly missed regardless of the step size) but, it is a good approximation of the Pareto front. However, it is not guaranteed to be an ϵ-cover.

5 Greedy Algorithm

The greedy algorithm presented here finds an epsilon cover of minimal cardinality for all Pareto non-dominated solutions. It is a general optimization technique that can be adapted to MMDPs. The algorithm can be summarized as follows.Find the maximum value for each criterion. Establish a lower bound for the second criterion within epsilon of the maximum value. Then maximize the second criterion subject to this bound. This solution will then be part of the epsilon cover. Then the loop over the following while each point found does not ϵ-cover the maximum value that was found for the first criteria. Find a point q' that maximizes the second criterion but is not covered by the previous point on the first criterion. Use the value of q' on the second criteria as the new maximum for the second criterion and repeat. This will find the minimum epsilon cover as long as the optimization technique is exact [3].

One problem with this algorithm is that value iteration is not an optimization technique that can be bounded on one criterion while optimizing on another. Therefore linear programming must be used which is less efficient. Further slowing the algorithm is that finding pure policies (policies with a deterministic assignment of actions to states) with linear programing is even slower. The time is still polynomial but large problems take a while to run as will be shown in experimental results.

Input: ϵ, m: an MMDP with criteria x, y
Output: An ϵ-cover of minimal cardinality
Compute value of optimal policy for criteria x and criteria y: x_{max} and y_{max}
$\bar{y}_1$ = $\frac{y_m in}{1+\epsilon}$;
q_1 = optimize(x, $y \leq \bar{y}_1$);
Q = q_1;
while $x_{min} < x(q_i)(1+\epsilon)$ **do**
    q'_{i+1} = optimize(y, $x < x(q_i)(1 + epsilon)$;
    $\bar{y}_{i+1}$ = min($\bar{y}_i$, $y(q'_{i+1})$) $\frac{1}{1+epsilon}$;
    q_{i+1} = optimize($x, y \leq \bar{y}_{i+1}$);
    Q = $Q \cup q_{i+1}$;
    $i = i + 1$;
    return Q;
end

Algorithm 3: Pseudocode for the Greedy algorithm

6 Divide and Conquer Approach

This is a variation of multi-objective optimization that is designed to find a minimum epsilon cover or at least an approximation within a small factor multiplied by the minimum. This method takes multi-objective optimization and changes the stopping conditions on the recusion in order to find an ϵ-cover. Currently a point is added to the covering set under a few conditions.

1. If it covers both parent points.
2. If it covers one parent it will be added and recursion stops between that parent and the new point.
3. If after search on both sides of the point along the curve, no covering solution is found then the aforementioned point becomes part of the cover.

The problem with this approach is that can possibly miss non-supported solutions just as multi-objective optimization does not find non-supported solutions. There is also no guarantee that the set found is of the minimal cardinality. However, it can be seen from tests results that it is significantly faster than the greedy approach. The initial step of the algorithm is the same as Multi-Criteria Optimization but the maximum for each criteria is not added unless the helper method, divideAndConquer, returns null. The pseudocode for the helper method is shown in Algorithm 6:

6.1 K-Best Policies for Finding Non-supported Solutions

As previously mentioned, the problem with the divide and conquer algorithm is that it fails to find non-supported solutions. This results under the stopping condition that no point is found between the two parent points. It is possible that there is a non-supported solution that is not epsilon

Input: p_1,p_2 are coordinate points representing the values of each criteria
Output: A set of Pareto optimal solutions approximating an ϵ-cover between p_1 and p_2
$y = \frac{(x(p_1)-x(p_2))}{(x(p_1)+y(p_2)-x(p_1)-y(p_1))}$;
$x = \frac{(y(p_1)-y(p_2))}{(x(p_1)+y(p_2)-x(p_1)-y(p_1))}$;
q = optimize(xR_1 + yR_2);
if q == p_1 **or** q == p_2 **then**
    Q = {};
end
if $x(q)(1 + epsilon) > x(p_1)$ **and** $y(q)(1 + epsilon) > y(p_2)$ **then**
    Q = $\{q\}$;
end
if $x(q)(1 + epsilon) > x(p_1)$ **then**
    Q = $\{q\} \cup$ divideAndConquer(q, p_2);
end
if $y(q)(1 + epsilon) > y(p_2)$ **then**
    Q = $\{q\} \cup$ divideAndConquer(p_1, q);
end
else
    Q = divideAndConquer$(p_1, q) \cup$ divideAndConquer(q, p_2);
end
if Q *does not cover* q **then**
    $Q = Q \cup \{q\}$;
end
return Q

Algorithm 4: Pseudocode for the Divide and Conquer Algorithm

dominated by the two points. Therefore the epsilon cover does not cover the entire Pareto front. One method of getting around this problem might be using the technique of finding the k-best policies. In the case that the best solution with the normalized rewards function is no better than p_1 or p_2 the technique is to find the next best policies that are not dominated by p_1 or p_2. These solutions can be used to complete the cover. The method of finding k-best policies that is used in the divide and conquer algorithm is the algorithm K-Best Improved (KBN) [2]. This algorithm relies on the theorem that the k^{th} best policy differs from the i-th best policy for some $i < k$ on exactly one state [2]. The drawback to this algorithm is that k must be known a priori. Ideally, the search for non-supported solutions would involve finding the second best, then the third best, and so on until a solution was found that was epsilon dominated by both p_1 and p_2. But since k must be known a priori the method must generate a list of solutions with a large k and then as many of them as needed are looked at. Once a solution on the list meets the stopping criterion the rest can be ignored. However if no solution on the list meets the stopping criterion then the algorithm must be run with a greater k. KBN can be efficient for large k values but slows as the size of the MDP grows.

7 Pareto Optimal Policy Relatedness

Given an MMDP, there will be a solution set of Pareto optimal policies such that each policy is not dominated by any other. Also, as presented in Section 1, a policy is a function that maps an action to each state. Another explored topic in this research was by how much do these functions differ from one another. In other words, how many states do two policies disagree on the action to take. This was tested by finding all Pareto optimal supported solutions. Each solution policy then was inserted into an undirected graph. A solution node was connected to another solution node if the number of states that the policies differed on was less than or equal to a given k. Depth first search was then used to find the number of connected components in the graph. The result was that for $k = 1$ (i.e, all Pareto optimal policies differ from another Pareto optimal policy by exactly one state) there were multiple connected components. For $k = 2$ there were fewer but still as the number of states increased so did the number of components. The conclusion for now is that there is no guarantee that all Pareto optimal policies are closely related. Future analysis could look at whether the inclusion of non-supported Pareto optimal policies would completely connect the entire graph.

8 Experimental Results

The divide and conquer algorithm and greedy algorithm were compared on a series of different problems to see how they compared in computational time and cover cardinality. The algorithms were implemented in Java and tests were performed within the Eclipse Interactive Developement Environment. Tests were run on a 2.66 GHz Dual-Core Intel(R) Core(Tm)2 CPU running Linux 3.2. Tests consisted of generating random MMDPs of varying size and measuring the performance of each algorithm on them. For each size problem, each algorithm was run on 10 different problems. Each MMDP was an infinite horizon problem with a discount factor of 0.9. It was not considered if the discount factor could have affected the results. The size of the test problems ranged from 20 to 500 states and for each number of states, tests were performed for 5 and for 10 actions being available in each state. Value iteration was used as the given optimization technique for the divide and conquer algorithm runs. For the algorithms finding an ϵ-cover, an ϵ value of 0.05 was used. To test that an ϵ-cover was in fact produced, all Pareto optimal supported solutions were found using the Multi-criteria Optimization algorithm. The average computation time for this algorithm was also recorded and the results displayed in table 1. Tables 2 and 3 show results for 100, 200, 300, 400, and 500 states with 5 and 10 actions respectively. Also it should be noted that the approximations were verified to be ϵ-covers for each trial.

8.1 Finding all Pareto Optimal Policies

The data for finding all solutions indicates that the number of Pareto optimal solutions and the computation time are both functions of the number of states. However it can be seen that an increase in the number of actions will result in an increase in the first derivative of computation

time. The number of solutions found appears to grow a little faster with more actions but is mainly a function of the number of states.

8.2 Comparing the Divide and Conquer and the Greedy Algorithm

The tabular and graphical comparisons of the divide and conquer and the greedy algorithm show that the former significantly outperforms the latter in terms of computation time. However, as shown in Tables 2 and 3, the greedy algorithm will find the ϵ-cover of minimal cardinality. In general the cardinality of the ϵ-cover found by the divide and conquer algorithm seems to be within a multiple of 2 of the actual minimum but this has not been proven.

States	Actions	Solutions	Time (s)
100	5	102.3	1.142940687
200	5	196	17.606066236
300	5	289	27.027831748
400	5	372.8	69.621308841
500	5	462.5	152.321939523
100	10	135.2	2.093502994
200	10	269.6	17.606066236
300	10	395.1	82.656662125
400	10	513	185.278586949
500	10	621.4	372.586691124

Table 1: Average run time and solutions for finding all solutions

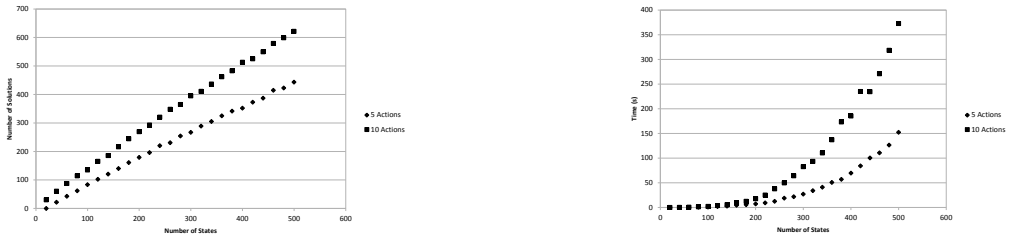

Figure 1: Experimental results for finding all Pareto optimal policies

States	Div and Conq. Time (s)	Greedy Time (s)	Div and Conq. Cover Size	Greedy Cover Size
100	0.0501	3.1151	6.3	3.5
200	0.1495	25.3162	6.1	3.6
300	0.3712	71.3525	6	3.2
400	0.8027	120.2230	6	3.3
500	1.3626	234.0653	6	3.3

Table 2: Average run time for EC Optimization and the Greedy algorithm with 5 actions

States	Div and Conq. Time (s)	Greedy Time (s)	Div and Conq. Cover Size	Greedy Cover Size
100	0.0697	5.1399	6.2	3.2
200	0.3043	39.5615	6	3.4
300	0.8988	119.1357	6	3.3
400	1.5838	218.9639	6	3.1
500	2.5032	416.2294	6	3.5

Table 3: Average run time for EC Optimization and the Greedy algorithm with 10 actions

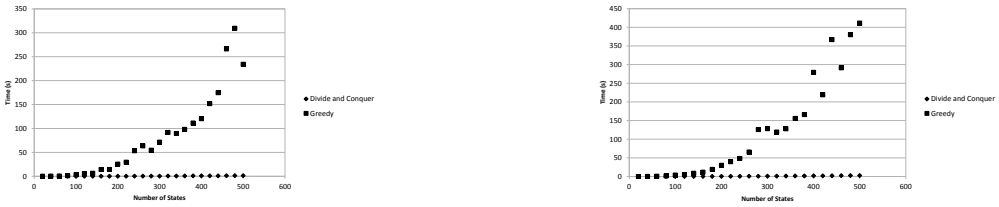

Figure 2: Experimental comparison of methods of finding an ϵ-cover

9 Conclusion

This work has discussed a method for finding all Pareto optimal policies for a MMDP. It has also introduced a novel idea for finding an ϵ-cover based on this method for finding all the policies. This method uses a divide and conquer approach to find the cover. It also makes use of a technique for finding the k-best policies for a given rewards function. This allows the algorithm to search for non-suppored solutions. In addition, an optimization method of finding the minimal ϵ-cover has been adapted for MMDPs. All of these algorithms have been tested experimentally. It was observed that this new divide and conquer algorithm is significantly faster then the greedy algorithm adapted to MMDPs. However, it does not guarantee a cover of minimal size.

Relatedness of Pareto optimal policies was experimentally looked at but results are inconclusive. Supported, Pareto optimal policies do not seem to have to be related but it is possible that if

non-supported policies are included then a relation can be established.

References

[1] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *J. Artif. Intell. Res. (JAIR)*, 11:1–94, 1999.

[2] Peng Dai and Judy Goldsmith. Ranking policies in discrete markov decision processes. *Springer Science and Business Media*, Published Online: 2010.

[3] Ilias Diakonikolas and Mihalis Yannakakis. Small approximate pareto sets for biobjective shortest paths and other problems. *Society for Industrial and Applied Mathematics*, 2009.

[4] Patrice Perny. *Décision Multicritre*, chapter 12. Universit Pierre et Marie Curie, 2011.

[5] Patrice Perny and Paul Weng. On finding compromise solutions in multiobjective markov decision processes. In *European Conference on Artificial Intelligence Multidisciplinary Workshop on Advances in Preference Handling*, pages 55–60, 2010.