In *In Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*

Improving Anytime Point-Based Value Iteration Using Principled Point Selections

Michael R. James, Michael E. Samples, and Dmitri A. Dolgov
AI and Robotics Group
Technical Research, Toyota Technical Center USA
{michael.james, michael.samples, dmitri.dolgov}@tema.toyota.com

Abstract

Planning in partially-observable dynamical systems (such as POMDPs and PSRs) is a computationally challenging task. Popular approximation techniques that have proved successful are point-based planning methods including point-based value iteration (PBVI), which works by approximating the solution at a finite set of points. These point-based methods typically are anytime algorithms, whereby an initial solution is obtained using a small set of points, and the solution may be incrementally improved by including additional points. We introduce a family of anytime PBVI algorithms that use the information present in the current solution for identifying and adding new points that have the potential to best improve the next solution. We motivate and present two different methods for choosing points and evaluate their performance empirically, demonstrating that high-quality solutions can be obtained with significantly fewer points than previous PBVI approaches.

1 Introduction

Point-based planning algorithms [Pineau *et al.*, 2003; Spaan and Vlassis, 2005] for partially-observable dynamical systems have become popular due to their relatively good performance and because they can be implemented as anytime algorithms. It has been suggested (e.g., in [Pineau *et al.*, 2005]), that anytime point-based planning algorithms could benefit significantly from the principled selection of points to add.

We provide several methods for using information collected during value iteration (specifically, characteristics of the value function) to choose new additional points. We show that properties of the value function allow the computation of an upper bound on the potential improvement (gain) resulting from the addition of a single point. We argue that the addition of points with maximal gain produces good approximations of the value function, and this argument is empirically verified. Further, we define an optimization problem that finds the point with the maximal gain for a given region.

These new approaches are empirically compared to traditional anytime point-based planning. The results show that our algorithms choose additional points that significantly improve the resulting approximation of the value function. Such improvement is potentially beneficial in two ways. First, computation time can be reduced because there are many fewer points for which computation must be performed. However, this benefit is sometimes (but not always) offset by the additional time required to select new points. Another potential benefit is reduced memory storage for smaller sets of points. Currently, the solution of most systems do not require large numbers of points, but a storage benefit may become more important as the size of problems increases such that more points are required.

2 Background

The planning methods developed here are for partially-observable, discrete-time, finite-space dynamical systems in which actions are chosen from a set $\mathcal{A}$, observations from a set $\mathcal{O}$, and rewards from a set $\mathcal{R}$. There is exactly one action, one observation, and one reward per time-step—planning algorithms attempt to find a policy that maximizes the long-term discounted reward with discount factor $\gamma \in [0, 1)$.

These point-based planning methods are suitable for at least two classes of models that can represent such dynamical systems, the well-known partially observable Markov decision processes (POMDP), and the newer predictive state representation (PSR) [Singh *et al.*, 2004]. For the sake of familiarity, we use POMDPs here, but it should be noted that the methods presented here (as well as other planning methods for such dynamical systems) can just as easily be applied to PSRs (see [James *et al.*, 2004] for details).

2.1 POMDPs

POMDPs are models of dynamical systems based on underlying latent variables called nominal states $s \in \mathcal{S}$. At any time step, the agent is in one nominal state s. Upon taking an action $a \in \mathcal{A}$ the agent receives a reward $r(s, a)$ that is a function of the state and action, transitions to a new state s' according to a stochastic transition model $pr(s'|s, a)$, and receives an observation $o \in \mathcal{O}$ according to a stochastic observation model $pr(o|s, a)$. For compactness, we define the vector r_a as $r_a(s) = r(s, a)$. [1]

[1] The notation $v(e)$ for vector v refers the value of entry e in v.

The agent does not directly observe the nominal state s, and so, must maintain some sort of memory to fully characterize its current state. In POMDPs, this is typically done using the *belief state*: a probability distribution over nominal states. The belief state b summarizes the entire history by computing the probability of being in each nominal state $b(s)$. This computation of the new belief state b_a^o reached after taking action a and getting observation o from belief state b is given in the following update:

$$b_a^o = \frac{\sum_{s'} pr(o|s',a) \sum_s pr(s'|s,a) b(s)}{pr(o|a,b)}.$$

This equation is used as the agent interacts with the dynamical system to maintain the agent's current belief state.

2.2 Point-based Value Iteration Algorithms

Point-based planning algorithms, such as PBVI and Perseus, for partially observable dynamical systems perform planning by constructing an approximation of the value function. By updating the value function for one point, the value-function approximation for nearby points is also improved. The basic idea is to calculate the value function (and its gradient) only at a small number of points, and the value function for the other points will be "carried along". This point-based approach uses only a fraction of the computational resources to construct the approximate value function, as compared to exact methods.

There are two main sets of vectors used: the set P of points, each of which is a belief state vector; and the set S_n which represents the approximate value function V_n at step n. The value function is a piecewise linear convex function over belief states, defined as the upper surface of the vectors $\alpha \in S_n$. The value for belief state b is

$$V_n(b) = \max_{\alpha \in S_n} b^T \alpha.$$

The Bellman equation is used to define the update for the next value function:

$$\begin{aligned}
V_{n+1}(b) &= \max_a \left[b^T r_a + \gamma \sum_o pr(o|a,b) V_n(b_a^o) \right] \\
&= \max_a \left[b^T r_a + \gamma \sum_o pr(o|a,b) \max_{\alpha^i \in S_n} (b_a^o)^T \alpha^i \right] \\
&= \max_a \Big[b^T r_a + \\
&\quad \gamma \sum_o \max_{\alpha^i \in S_n} \sum_{s'} pr(o|s',a) \sum_s pr(s'|s,a) b(s) \alpha^i(s') \Big] \\
&= \max_a \left[b^T r_a + \gamma \sum_o \max_{\{g_{ao}^i\}} b^T g_{ao}^i \right]
\end{aligned}$$

where

$$g_{ao}^i(s) = \sum_{s'} pr(o|s',a) pr(s'|s,a) \alpha^i(s'), \qquad (1)$$

for $\alpha^i \in S_n$. These g vectors are defined as such because they lead to computational savings. The computation of α vectors for $V_{n+1}(b)$ uses the so-called backup operator.

$$backup(b) = \operatorname*{argmax}_{\{g_a^b\}_{a \in \mathcal{A}}} b^T g_a^b, \qquad (2)$$

where $g_a^b = r_a + \gamma \sum_o \operatorname{argmax}_{\{g_{ao}^i\}_i} b^T g_{ao}^i$. In PBVI, the backup is applied to each point in P, as follows.

```
function oneStepPBVIbackup($P$, $S_{old}$)
 1. $S_{new} = \emptyset$
 2. For each $b \in P$ compute $\alpha = backup(b)$ and
    set $S_{new} = S_{new} \cup \alpha$
 3. return $S_{new}$
```

For Perseus, in each iteration the current set of α vectors is transformed to a new set of α vectors by randomly choosing points from the current set P and applying the backup operator to them. This process is:

```
function oneStepPerseusBackup($P$, $S_{old}$)
 1. $S_{new} = \emptyset$, $\overline{P} = P$, where $\overline{P}$ lists the
    non-improved points
 2. Sample $b$ uniformly at random from $\overline{P}$,
    compute $\alpha = backup(b)$
 3. If $b^T\alpha \geq V_n(b)$ then add $\alpha$ to $S_{new}$, else
    add $\alpha' = \text{argmax}_{\alpha'' \in S_{old}} b^T \alpha''$ to $S_{new}$
 4. Compute $\overline{P} = \{b \in \overline{P} : V_{n+1}(b) - \epsilon < V_n(b)\}$
 5. If $\overline{P}$ is empty return $S_{new}$, otherwise
    go to step 2
```

Given this, the basic (non-anytime) algorithms are, for * $\in$ [PBVI, Perseus]:

```
function basic*()
 1. $P$ = pickInitialPoints(), $S_0$ =
    minimalAlpha(), $n = 0$
 2. $S_{n+1}$ = oneStep*Backup($P$, $S_n$)
 3. increment $n$, goto 2
```

where pickInitialPoints() typically uses a random walk with distance-based metrics to choose an initial set of points, and the function minimalAlpha() returns the α vector with all entries equal to $\min_{r \in \mathcal{R}} r/(1-\gamma)$. The algorithm will stop on either a condition on the value function or on the number of iterations. This algorithm is easily expanded to be an anytime algorithm by modifying pickInitialPoints() to start with a small set of initial points and including a step that iteratively expands the set of current points P. Typically, having a current set with more points will result in a better approximation of the value function, but will require more computation to update. The anytime versions of the algorithms are:

```
function anytime*()
 1. $P$ = pickInitialPoints(), $S_0$ =
    minimalAlpha(), $n = 0$
 2. if(readyToAddPoints()) then $P$ = $P$ ∪
    bestPointsToAdd(findCandidatePoints())
 3. $S_{n+1}$ = oneStep*Backup($P$, $S_n$)
 4. increment $n$, goto 2
```

There are three new functions introduced in this anytime algorithm: readyToAddPoints() which determines when the algorithm adds new points to S; findCandidatePoints() which is the first step in determining the points to add; and bestPointsToAdd() which takes the set of candidate points and determines a set of points to add. In Section 4, we present variations of these functions, including our main contributions, methods for determining the best points to add based on examination of the value function.

3 Incremental PBVI and Perseus

In this section we present methods for finding the best points to add, given a set of candidate points. In some cases, the best points will be a subset of the candidates, and in other cases new points that are outside the set of original candidates will be identified as best. In all cases, we will make use of a scalar measure called the *gain* of a point, which is a way to evaluate how useful that point will be. We show how the current (approximate) value function can be used to compute useful measures of gain that can then be used in informed methods of point selection.

For a point b in the belief space, the gain $g(b)$ estimates how much the value function will improve. The problem, then, is to construct a measure of gain that identifies the points that will most improve the approximation of the value function. This is a complex problem, primarily because changing the value function at one point can affect the value of many (or all) other points, as can be seen by examining the backup operator (Equation 2). Changes may also propagate over an arbitrary number of time steps, as the changes to other points propagate to yet other points. Therefore, exactly identifying the points with best gain is much too computationally expensive, and other measures of gain must be used.

Here, we present two different measures of gain: one based on examining the backup operator, written g_B, and the other is derived from finding an upper bound on the amount of gain that a point may have. We use linear programming to compute this gain, written g_{LP}.

Given these definitions of gain, we present two different methods for identifying points to add. The simplest method (detailed in Section 4) takes the candidate points, orders them according to their gains, and then selects the top few. A more sophisticated method leverages some information present when computing g_{LP} to identify points that have even higher gain than the candidate points, and so returns different points than the original candidates. We describe how these selection methods can be implemented in Section 4, following a detailed discussion (Sections 3.1 and 3.2) of the two measures of gain.

3.1 Gain Based on One-Step Backup

The gain g_B measures the difference between the current value of point b and its value according to an α vector computed by a one-step backup of b via Equation 2. This measure does not have strong theoretical support, but intuitively, if the one-step difference for a point is large, then it will also improve the approximations of nearby points significantly. These improvements will then have positive effects on points that lead to these points (although the effect will be discounted), improving many parts of the value function. This gain is defined as:

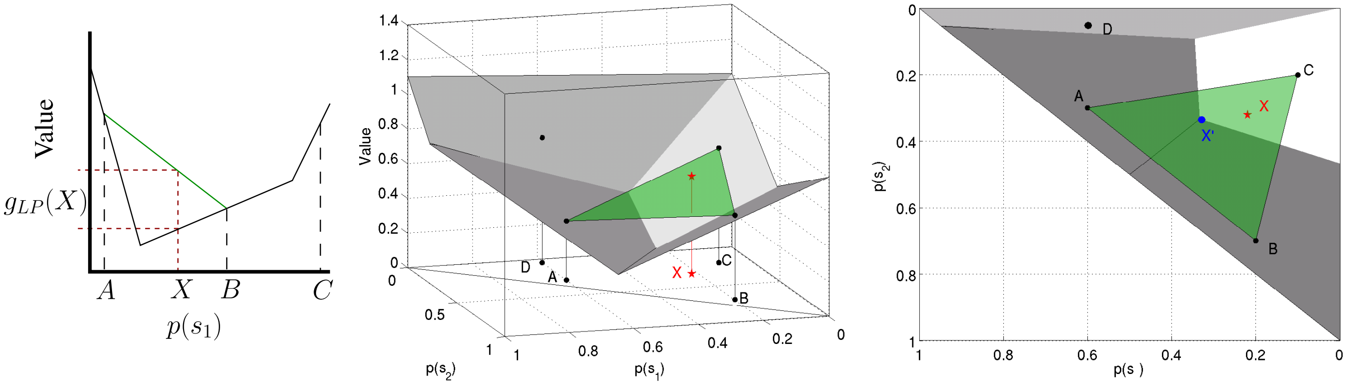
$$\begin{aligned} g_B(b) &= \max_a \Big[r(b,a) + \gamma \sum_o pr(o|b,a) V(b_a^o) \Big] - V(b) \\ &= b^T \alpha - V(b), \end{aligned}$$

where $\alpha =$ backup (b). Note that this measure includes the immediate expected reward at b, a quantity that was not previously included in the computation of any α vector. This measure can make use of cached g_a vectors (Equation 1), and so it is inexpensive from the computational standpoint. In the following section, we describe a different measure of gain that has a stronger theoretical justification, but also comes at the expensive of higher computational cost.

3.2 Gain Based on Value-Function Bounds

This second measure of gain that we introduce uses the following insight, which we first illustrate on a system with two nominal states, then extend the intuition to a three-state system, and then describe the general calculation for any number of states. The value function is a piecewise linear convex function consisting of (for two nominal states) the upper surface of a set of lines defined by α vectors (see Figure 1a for a running example). The vectors which define these lines correspond to points interior to the regions where the corresponding lines are maximal (points A, B, and C). Intuitively, the value function is most relevant at these points, and becomes less relevant with increasing distance from the point. Here, we assume that the value given by the α vector is correct at the corresponding point. Now consider the inclusion of a new α vector for a new point (point X in Figure 1a). If the value function for point A is correct, then the new α vector cannot change the value of A to be greater than its current value; the same holds for B. Therefore, the new α vector for point X is upper-bounded by the line between the value for A and the value for B. Thus, the gain for X is the difference between this upper bound and the current value of X.

For the case with three nominal states, the analogous planar solution to the upper bound is not as simple to compute (see Figure 1b and 1c). To visualize the problem, consider a mobile plane being pushed upward from point X, which is constrained only to not move above any of the existing points that define the value-function surface (A, B, C, and D in Figure 1b). In the example depicted in Figure 1b, this plane will come to rest on the points defined by the locations of A, B, C in the belief space and their values. The value of that plane at X then gives the tightest upper bound for its value if we were to add a new α vector at X. The gain is then computed in the same manner as in the two state case above, but finding the above-described plane is not straightforward (and becomes even more difficult in higher-dimensional spaces). The difficulty is that in higher dimensional spaces — in contrast to the two state case where the minimal upper bound is always defined by the points immediately to the left and right of the new point — selecting the points that define that minimal upper-bounding surface is a non-trivial task. However, this problem can be formulated as



(a) (b) (c)

Figure 1: **(a):** Value function for a dynamical system with two nominal states (note that $p(s_2) = 1 - p(s_1)$, so an axis for $p(s_2)$ is unnecessary). The value function is defined by the α vectors corresponding to points A, B, and C. The gain $g_{LP}(X)$ for point X is the difference between the upper bound defined by the line $\langle V(A), V(B)\rangle$ and the current value of X. **(b,c):** Value function for a system with three nominal states. The value function is defined by the α vectors corresponding to points A, B, C, and D, and the gain $g_{LP}(X)$ for point X is the difference between the tightest upper bound (defined by plane $\langle V(A), V(B), V(C)\rangle$) and the current value for X. That tightest upper-bound plane is produced by the first LP (FindGainAndRegionForPoint). X' is the best point produced by the second LP (FindBestPointForRegion) for the bounding region $\{A, B, C\}$.

a linear program (LP). We describe this LP in Table 1 and refer to it as FindGainAndRegionForPoint(b_0).

Given a candidate belief point b_0 its current scalar value v_0, the current points $\{b_1...b_m\}$, the current scalar values $v_1...v_m$ of these points, and the value vectors $\alpha_1...\alpha_n$, this minimization LP finds the tightest upper bound $V_u(b_0) = \sum_i w_i v_i$, which can then be used to compute the gain $g_{LP}(b_0) = V_u(b_0) - V(b_0)$. In words, this minimization LP works by finding a set of points (these are the set of points with non-zero weights w_i) such that: i) b_0 can be expressed a convex linear combination of these points, and ii) the value function defined by the plane that rests on their value-function points is the minimal surface above v_0. We call the set of such points with non-zero weights the *bounding region* $\Gamma(b_0) = \{b_i | w_i > 0\}$ for point b_0; it has the property that the point b_0 must be interior to it, and these bounding regions will play an important role in the next section.

There is one subtle issue regarding the above LP, which is that not all points of potential interest are interior to the set of current points for which α vectors are defined (in fact early in the process when there are few points, most of the useful points will be outside the convex hull of current points). This is problematic because we want to consider these points (they often correspond to relatively poorly approximated areas of the value function), but they are never interior to any subset of the current points.

To resolve this issue, we introduce the set U of *unit basis points*: the points with all components but one equal to 0, and the remaining entry equal to 1 (the standard orthonormal basis in the belief space). We then augment the set of current belief points in our LP to include the unit basis points: $\{b_1...b_m\} = \{P \cup U\}$. The values v_j of the points in $U \setminus P$ are computed as $v_j =$ backup $(b_j)\ \forall b_j \in U \setminus P$, since these points were not updated in oneStep*Backup() as were the points in P. As before, the variables in this augmented LP are the scalar weights $w_1...w_m$ that define which points from $\{P \cup U\}$ form the bounding region $\Gamma(b_0)$

3.3 Finding the Best Point in a Bounding Region

Given the gains g_{LP} for the candidate points, it is possible to choose points to add based on this criteria, but one drawback to this is that we might add multiple points from the same bounding region Γ, and thus waste resources by approximating the value function for multiple points when just one would work almost as well. Further, it may be that none of these points is the point in the region with the highest gain. To address these issues, we introduce another LP called FindBestPointForRegion(Γ) (Table 1) that takes a bounding region and finds the point interior to that region with maximal g_{LP}. Note that this region may contain unit basis points, allowing the new point to be exterior to the current set of points.

This LP takes as input a bounding region $\Gamma(b_0) = \{b_1...b_l\}$, scalar values $v_1...v_l$ of these points, and the value vectors $\alpha_1...\alpha_n$. The optimization variables are: scalar weights $w_1...w_l$, vector b_{new} , which is the point to be found, and the scalar value v_{new}, which is the value of b_{new}. The process of finding the best points to add will call this LP on all unique regions found by running the first LP (FindGainAndRegionForPoint) on all candidate points.

4 Using Gains in PBVI and Perseus

We now define two incremental algorithms, PBVI-I and Perseus-I, using the gains g_B and g_{LP}, and the linear programs above. Referencing the anytime*() algorithm, we explored various possibilities for readyToAddPoints(), bestPointsToAdd(C), and findCandidatePoints(). While many variations exist, this paper presents only a few that seem most promising.

Table 1: The linear programs used to i) compute the gain and region $\Gamma(b_0)$ for point b_0 given a value function, and ii) find the best point within a given region Γ given the value function.

FindGainAndRegionForPoint(b_0):		**FindBestPointForRegion(Γ)**	
Minimize:	$\sum_i w_i v_i$	**Maximize:**	$\left(\sum_i w_i v_i\right) - v_{new}$
Given:	$b_0, v_0, b_1 ... b_m, v_1 ... v_m, \alpha_1 ... \alpha_n$	Given:	$b_1 ... b_l, v_1 ... v_l, \alpha_1 ... \alpha_n$
Variables:	$w_1 ... w_m$	Variables:	$w_1 ... w_l, v_{new}, b_{new}$
Constraints:	$b_0 = \sum_i w_i b_i; \quad \sum_i w_i v_i \geq v_0$	Constraints:	$b_{new} = \sum_i w_i b_i; \quad w_i \geq 0 \ \forall i \in [1..l]$
	$\sum_i w_i = 1; \quad w_i \geq 0 \ \forall i \in [1..m]$		$\sum_i w_i = 1; \quad v_{new} \geq \alpha_j b_{new} \ \forall j \in [1..l]$

Two computationally cheap possibilities for readyToAddPoints() are to add points: i) after a fixed number of iterations, and ii) when the update process has approximately converged, i.e., the maximal one-step difference in value over all points is below some threshold ($\max_{b \in P}(V_{n+1}(b) - V_n(b)) \leq \epsilon$). We have also explored two methods for findCandidatePoints(). The first is to keep a list of points that are one-step successors to the points in P, the set $\{b_a^o | b \in P\}$. The second method is to do a random walk and use a distance threshold to choose points, just as in pickInitialPoints().

The gains discussed in the previous section can then be used for bestPointsToAdd(C). Gain g_B orders the candidate points C and the top k are chosen. For gain g_{LP}, the candidate points define a set of unique bounding regions $\{\Gamma(b) | b \in C\}$. For each region, the LP FindBestPointForRegion() returns the best candidate point b'. These candidate points are ordered by $g_{LP}(b')$ and the top k are chosen. To compare our methods against a baseline anytime algorithm, we used the standard random walk method, choosing the first k points that exceeded a distance threshold.

5 Experiments

To evaluate our principled point selection algorithms, we conducted testing on a set of standard POMDP domains. Three of the problems are small-to-mid sized, and the fourth (Hallway) is larger. Domain definitions may be obtained at [Cassandra, 1999]. We tested three algorithms: i) a baseline, anytime Perseus using the distance-based metric; ii) one-step backup Perseus-I using g_B; and iii) LP-based Perseus-I using g_{LP} and the two LPs for finding the best points.

All three algorithms used the convergence condition with $\epsilon = 1E-3$ for readyToAddPoints(). The algorithms for Network, Shuttle, and 4x3 added up to 3 points at a time, while Hallway added up to 30 points at a time. However, choosing points using LPs often did not add all 30 because fewer unique regions were found. In the one-step backup Perseus-I, we used the successor metric for findCandidatePoints() as it was easily integrated with little overhead, while in the LP-based Perseus-I we used a random walk with a distance metric as it introduced the least overhead in this case. The results were averaged over 30 runs, and the performance metric is the average discounted reward, for a given initial belief state. The results are presented in the graphs in Figure 2, which are of three types. The first type (performance vs. number of points) is a measure of the effectiveness of point selection. The second type (performance vs. number iterations) and the third type (performance vs. CPU time) both measure how well the use of principled point selection affects the overall performance, but the third type shows the impact of additional time incurred by the point-selection process. These graphs show performance on the x-axis, and the amount of resource (points, iterations, or time) used to achieve that performance on the y-axis. When comparing two algorithms with respect to a given resource, the better algorithm will be lower.

Examination of the first-type graphs shows that the LP-based Perseus-I did a good job of identifying the best points to add. The resulting policies produced better performance with fewer points than either of the other two methods. Furthermore, on three of the problems, the one-step backup Perseus-I also did significantly better than the baseline for choosing points. Thus, for this metric, our goal of finding useful points has been achieved. Most promisingly, on the largest problem (Hallway) our algorithms used significantly fewer points to achieve good performance. However, the other graphs show that choosing good points does not always translate into a faster algorithm. On one problem (Network) both versions of Perseus-I were faster than the baseline, but on the other problems, the performance was either closely competitive or slightly worse due to the overhead incurred by identifying good points. While this result was not optimal, the fact that there were significant speedups on one problem combined with the point selection results leads us to draw the conclusion that this method is promising and needs further investigation and development.

References

[Cassandra, 1999] A. Cassandra. Tony's pomdp page. http://www.cs.brown.edu/research/ai/ pomdp/index.html, 1999.

[Izadi *et al.*, 2005] Masoumeh T. Izadi, Ajit V. Rajwade, and Doina Precup. Using core beliefs for point-based value iteration. In *19th International Joint Conference on Artificial Intelligence*, 2005.

[James *et al.*, 2004] Michael R. James, Satinder Singh, and Michael L. Littman. Planning with predictive state representations. In *The 2004 International Conference on Machine Learning and Applications*, 2004.

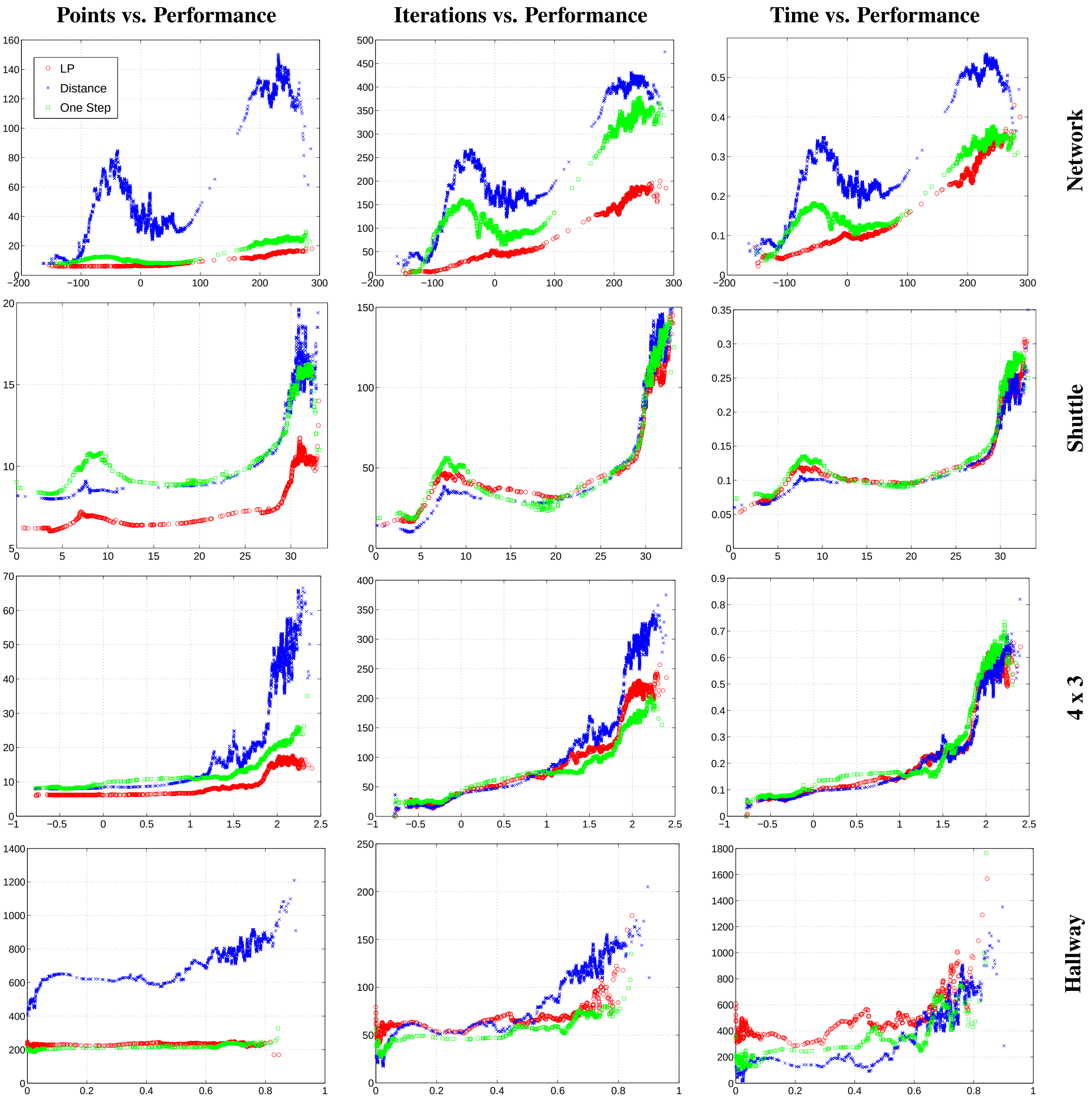


Figure 2: Empirical comparisons of three methods for selecting new points during value iteration. Each graph evaluates performance against one of three metrics—number of points, total number of iterations used in constructing a policy, and total CPU time. The y-axis of each graph represents the average value of the column's metric required to earn a particular reward (x-axis) during empirical evaluation. Experiments were conducted on four dynamical systems (one per row): Network, Shuttle, 4x3, and Hallway. Lower values are indicative of less resource consumption, and so are more desirable.

[Pineau *et al.*, 2003] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *Proc. 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, 2003.

[Pineau *et al.*, 2005] J. Pineau, G. Gordon, and S. Thrun. Point-based approximations for fast pomdp solving. Technical Report SOCS-TR-2005.4, McGill University, 2005.

[Shani *et al.*, 2005] Guy Shani, Ronen Brafman, and Solomon Shimony. Model-based online learning of POMDPs. In *16th European Conference on Machine Learning*, 2005.

[Singh *et al.*, 2004] Satinder Singh, Michael R. James, and Matthew R. Rudary. Predictive state representations, a new theory for modeling dynamical systems. In *20th Conference on Uncertainty in Artificial Intelligence*, 2004.

[Spaan and Vlassis, 2005] M.T.J Spaan and N. Vlassis. Perseus: Randomized point-based value iteration for pomdps. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.