

Randomly Sampling Actions In Dynamic Programming

Christopher G. Atkeson

Robotics Institute, Carnegie Mellon University

5000 Forbes Avenue, Pittsburgh, PA 15213, USA, www.cs.cmu.edu/~cga, cga@cmu.edu

Abstract—We describe an approach towards reducing the curse of dimensionality for deterministic dynamic programming with continuous actions by randomly sampling actions while computing a steady state value function and policy. This approach results in globally optimized actions, without searching over a discretized multidimensional grid. We present results on finding time invariant control laws for two, four, and six dimensional deterministic swing up problems with up to 480 million discretized states.

I. INTRODUCTION

Dynamic programming [1], [2], [3] has had limited application to control problems with continuous states and actions (controls) because of its large computational cost for any reasonable discretization of the states and actions. In this paper we describe an approach to remove the need to discretize the actions and reduce the dependence of the computational cost on the dimensionality of the actions for time invariant problems. We do this by comparing a single randomly sampled action to the current best action at each state on each sweep. In current approaches, with a dimensionality of the action vector of d_u , and a discretization of each dimension of the action vector of R_u , the cost of performing a value function update is proportional to $R_u^{d_u}$. In our approach, we only evaluate 2 actions: the current best action and a randomly sampled action. Over many such value function updates of the same state, the random search performs a global optimization.

II. APPROACH

We solve the following approximate deterministic dynamic programming problem, where we have:

1) Deterministic time invariant discrete time system dynamics:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) \quad (1)$$

where \mathbf{x}_k is the state on the k th time step, and \mathbf{u}_k is the control or action vector. We assume a model of $\mathbf{f}(\cdot)$ is available. This paper does not address simultaneously learning a dynamic model and finding a good policy.

2) A time invariant one step reward/cost function. We use a loss function which obeys the convention that costs are positive quantities which must be minimized, rather than maximizing positive rewards:

$$L_k = L(\mathbf{x}_k, \mathbf{u}_k) \quad (2)$$

We optimize the discounted case

$$\min \sum_{k=1}^{\infty} \gamma^k L_k \quad (3)$$

where γ is the discount factor ($\gamma = 0.9999$ in this work).

A. A Simple Solution Scheme (SSS)

We compute a steady state value function $V(\mathbf{x})$ to solve this optimization problem. We use Bellman's equation to iteratively approximate $V(\mathbf{x})$ by solving backwards in time:

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} (L(\mathbf{x}, \mathbf{u}) + \gamma V_{k+1}(\mathbf{f}(\mathbf{x}, \mathbf{u}))) \quad (4)$$

We represent the value function using a large table. The table is indexed by the components of the state vector, and each cell holds a value for $V(\mathbf{x})$ for the center of that cell. Each time a value $V(\mathbf{x})$ for an arbitrary state is accessed, the stored values are used in multilinear interpolation to produce an estimate of that value [4]. If neighboring elements of the value function table do not have a valid value (not yet initialized or all trajectories from the state go outside the table), distance-weighted averaging is used instead of multilinear interpolation. The policy $\mathbf{u}(\mathbf{x})$ can be explicitly represented by storing the minimizing \mathbf{u} from Eq. 4 for each state in a similar table, with multilinear interpolation used to produce actions for arbitrary states. Eq. 4 can be used to update the value function and policy for each cell. We refer to the application of Eq. 4 to a single state as a “Bellman update” of that state. An update of all states is referred to as a “sweep”. The overall process is known as *value iteration*.

In the simple solution scheme a grid of discretized actions is used to perform the minimization in Eq. 4. Each dimension of the action \mathbf{u} is discretized with a resolution R_u , resulting in $R_u^{d_u}$ elements in the action grid, where d_u is the number of action dimensions.

1) Problems With The SSS: The Curse of Dimensionality:

The curse of dimensionality is a general problem in dynamic programming. With the simple solution scheme it takes several forms:

- 1) **Storage cost:** The number of stored samples of the value function grows exponentially with the dimensionality d_x of the state vector. If a resolution R_x in each dimension is used, there are $R_x^{d_x}$ cells in the table. Storing the resulting policy requires $d_u * R_x^{d_x}$ table entries. This paper does not address methods to reduce this cost by intelligently or adaptively sampling states.
- 2) **The computational cost of a Bellman update is proportional to the total number of actions:** The

amount of work to perform the minimization in Bellman’s equation (Eq. 4) at each state (a Bellman update or BU) is proportional to $R_u^{d_u}$. This paper addresses reducing the computational cost to find actions, in the case of continuous actions.

- 3) **The computational cost of a sweep is proportional to the total number of states:** A sweep in which all states are updated has computational cost proportional to $R_x^{d_x} * BU$. This paper does not address methods to reduce this cost by reordering the state updates.
- 4) **The computational cost of interpolating value function and policy entries grows exponentially with the dimensionality of the state.** The cost of grid-based interpolation such as multilinear interpolation grows exponentially with the input dimensionality. This problem has already been solved by other types of interpolation, such as barycentric interpolation [4], kernel regression, and locally weighted regression [5].

B. Globally Optimizing The Action

We wish to minimize a continuous function of the action \mathbf{u} at a given state \mathbf{x}_i :

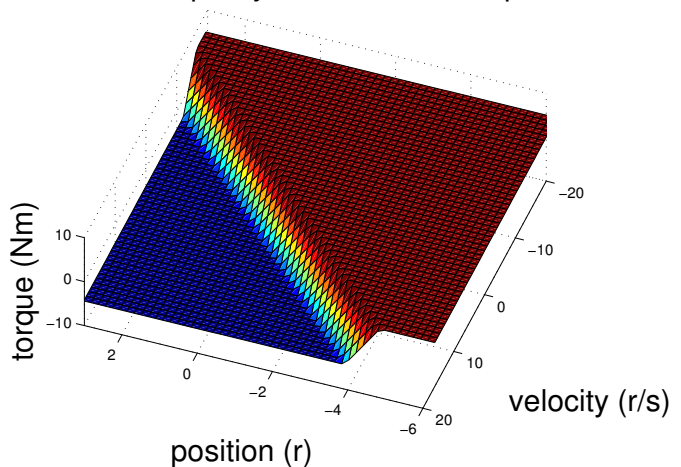
$$g(\mathbf{u}) = L(\mathbf{x}_i, \mathbf{u}) + \gamma V(\mathbf{f}(\mathbf{x}_i, \mathbf{u})) \quad (5)$$

We globally optimize the action by comparing the value of $g()$ at the current best action for state \mathbf{x}_i in the policy $\mathbf{u}(\mathbf{x}_i)$ to the value of $g()$ at a randomly selected action \mathbf{u}_{random} on each Bellman update. The policy could be represented in a number of ways, including lookup tables or parametric representations. We used a lookup table in this work, so the policy corresponding to each state is explicitly represented as a parameter $\mathbf{u}(\mathbf{x}_i)$. The random action could be generated using any distribution function appropriate to the problem being solved. In this work the action is uniformly distributed within bounds set by the user. Each component is independent and bounded by $\pm 10\text{Nm}$. If the random action has a lower value $g(\mathbf{u}_{random})$ than the action from the current policy for the state \mathbf{x}_i , the random action replaces the current action in the policy: $\mathbf{u}(\mathbf{x}_i) = \mathbf{u}_{random}$. In either case the value function is updated with the new value: $V(\mathbf{x}_i) = g(\mathbf{u}(\mathbf{x}_i))$. The new action is compared to other randomly chosen actions on future Bellman updates.

C. Utilizing a Default Policy

In many robotics problems there is a desired end state or goal. We have found that policies created by dynamic programming near the goal are often not as effective as policies created using Linear Quadratic Regulator (LQR) design [6]. We suspect this is due to the limited resolution we can use in higher dimensional problems. In cases where LQR techniques apply, we use the LQR policy as a default policy, and the policy produced by dynamic programming is a correction to the LQR policy (Fig. 1). In this paper the default policy is limited for each action dimension to $\pm 5\text{Nm}$. The initial value function is initialized to the value function produced by the LQR design process.

Default policy for one link example



Computed policy for one link example

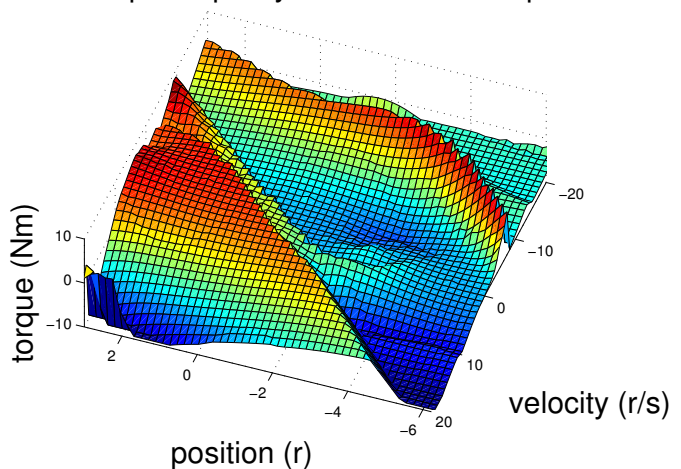


Fig. 1. The default policy and the policy computed by dynamic programming for the one link pendulum example. The sum of these policies is the actual policy applied.

III. RESULTS

Randomly sampling actions has greatly reduced the computational cost and produced as good or better results than fixed resolution action discretizations. We are using swing up problems to empirically explore the effect of state and action dimensionality on performance. A pendulum is swung from the stable equilibrium (hanging down) to the unstable equilibrium (upright). We can create swing up problems of any desired dimensionality by subdividing the pendulum into links connected by powered joints. Each additional joint adds two dimensions to the state (position and velocity) and one action (torque at that joint).

One link pendulum example: We initially present the simplest swing up problem, where the pendulum is made up of a single link (Fig. 2). Because the state space is two dimensional (position and velocity for the joint at the base) we can plot the value function (Fig. 3) and policy (Fig. 4)

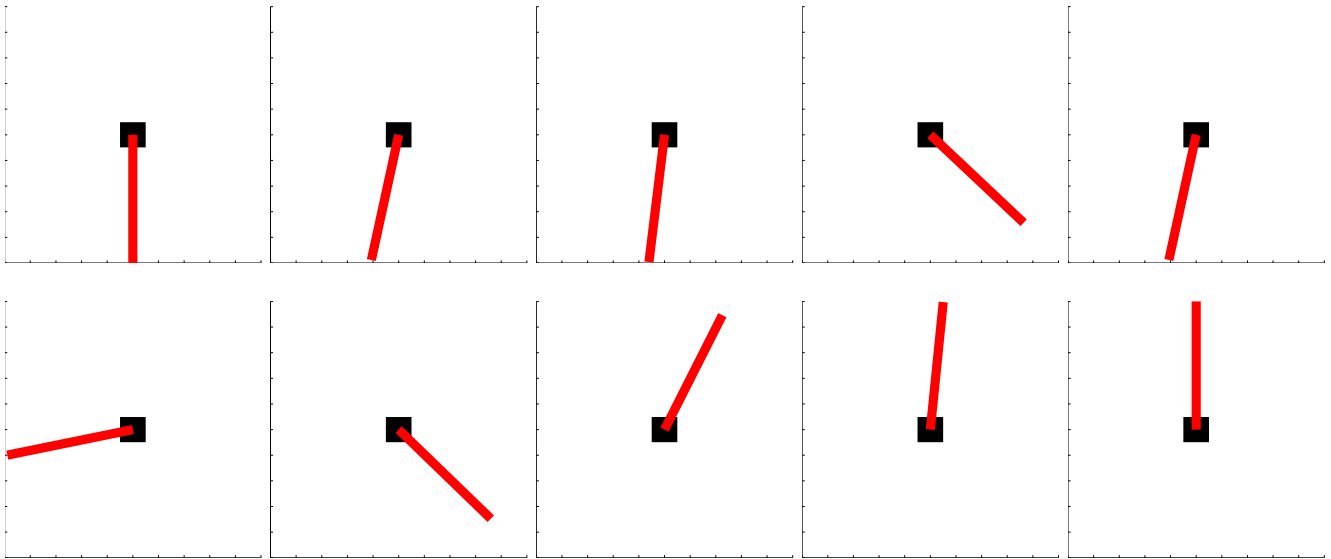


Fig. 2. Configurations from the simulated one link pendulum optimal trajectory at times 0,0.5,1,1.5,2,2.5,3,3.5,4,4.5,5sec., and at the end of the trajectory. This figure is read first left to right, and then top to bottom.

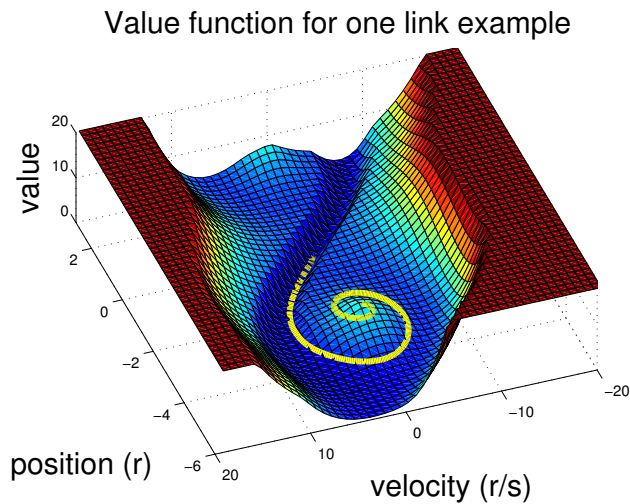


Fig. 3. The value function computed by the proposed approach for a one link pendulum swing up. The optimal trajectory is shown as a yellow line. The value function is cut off above 20 so we can see the details of the part of the value function that determines the optimal trajectory. The goal is at the state (0,0).

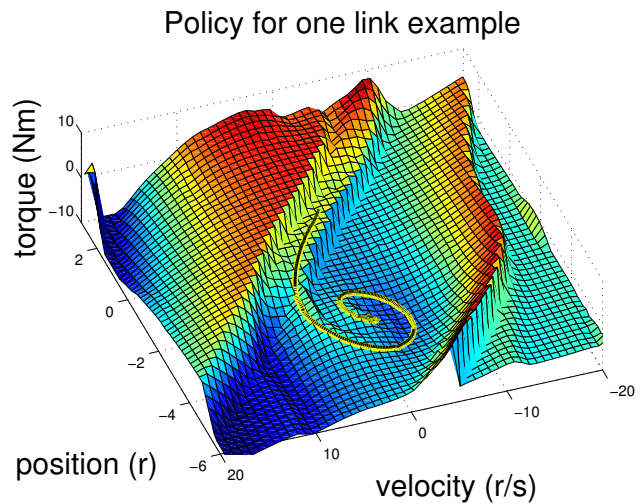


Fig. 4. The policy computed by the proposed approach for a one link pendulum swing up. The optimal trajectory is shown as a black line with a yellow border. The goal is at the state (0,0).

as a function of state. We discretized the states with a resolution of 100 in each dimension, resulting in 10,000 states. The discretized states were bounded by $-2\pi < \theta < \pi$ and $-20 < \dot{\theta} < 20$, where $\theta = 0$ is upright. The angle bound was set to prevent too much rotation of the corresponding joint, which might cause damage to the structure or wiring. The velocity bounds are somewhat arbitrary. The action (joint torque) was bounded by $\pm 10\text{Nm}$. The one step cost function was a weighted sum of the squared position errors and the squared torques: $L(\mathbf{x}, \mathbf{u}) = 0.1 * T * (\theta^2) + T * (\tau^2)$ where 0.1 weights the position error relative to the torque penalty, and

T is the time step of the simulation (0.01s). There were no costs associated with the joint velocity. Randomly sampling one action vector per Bellman update resulted in the trajectory shown in Figures 2, 3, 4, and 6. This trajectory had a cost of 7.2. We applied local trajectory optimization (an approach known as Differential Dynamic Programming or DDP) to the trajectory produced by dynamic programming, and got a further improvement to a cost of 7.1 (Fig. 6) [7], [8]. The value function computed using our approach differs from the value function computed using an action grid of size 100 (and all other solution details kept the same) by at most 0.02.

Two link pendulum example: The next more complex

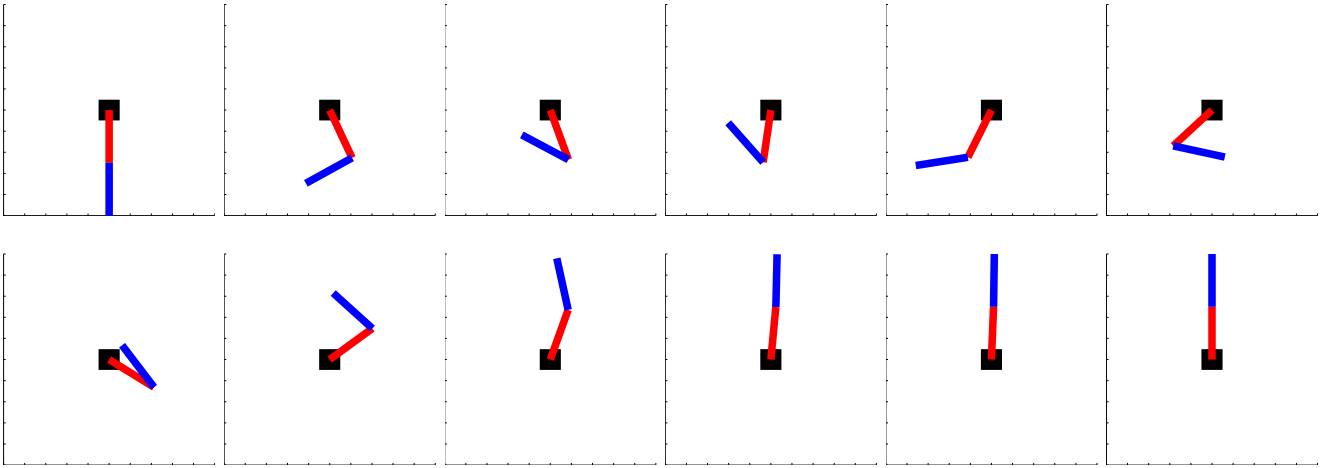


Fig. 5. Configurations from the simulated two link pendulum optimal trajectory at times 0,0.2,0.4,0.6,0.8,1,1.2,1.4,1.6,1.8,2sec., and the end of the trajectory.

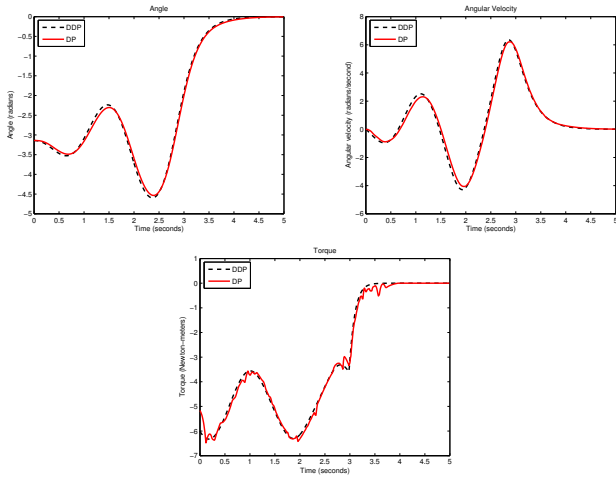


Fig. 6. The positions, velocities, and torques for the simulated one link swing up. The trajectory generated by our approach (DP) is shown with the red solid line (cost = 7.2). A trajectory obtained by performing local trajectory optimization (DDP) on the red trajectory is shown with the black dashed line (cost = 7.1).

example is a two link pendulum which is also swung up from the stable equilibrium (hanging down) to the unstable equilibrium (upright) (Fig. 5). In this case the state has four dimensions (a position and velocity for each joint) and a two dimensional action (a torque at each joint). We discretized at a number of resolutions: 148 in each dimension resulting in 480 million states, 100 in each dimension resulting in 100 million states, and 60 in each dimension resulting in 13 million states. The two finer discretizations produced essentially the same results, and the lowest resolution produced a slightly higher cost trajectory. The discretized states were bounded by $-2\pi < \theta_1 < \pi$, $-\pi < \theta_2 < \pi$, $-20 < \dot{\theta}_1 < 20$, and $-40 < \dot{\theta}_2 < 40$ where $\theta_i = 0$ is upright. The two actions (torques) were bounded by $\pm 10\text{Nm}$. The one step cost function was an

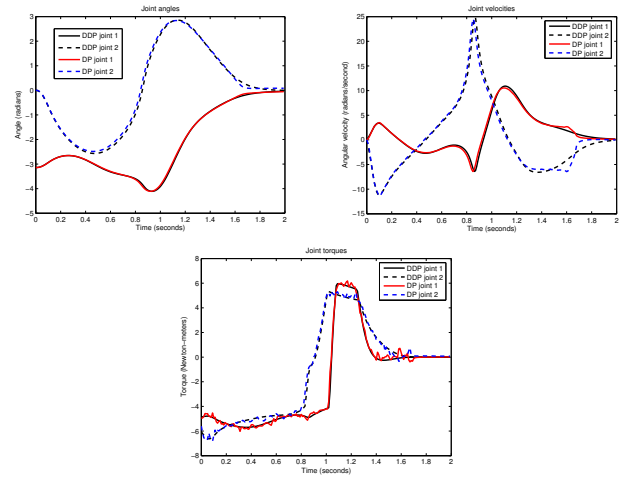


Fig. 7. The trajectory generated by dynamic programming (DP) for the two link swing up, and the locally optimized trajectory (DDP) using the DP trajectory as the initial trajectory.

extension of the one link pendulum cost function, a weighted sum of the squared position errors and the squared torques: $L(\mathbf{x}, \mathbf{u}) = 0.1 * T * (\theta_1^2 + \theta_2^2) + T * (\tau_1^2 + \tau_2^2)$. Again, 0.1 weights the position errors relative to the torque penalty, T is the time step of the simulation (0.01s), and there were no costs associated with joint velocities. Randomly sampling one action vector per Bellman update resulted in the trajectory shown in Figures 5 and 7. This trajectory (DP) had a cost of 3.0. Local trajectory optimization (DDP) of the DP trajectory reduced the cost further to 2.9 (Fig. 7).

How does our approach compare to using a grid to search for actions on each update? Table I compares the computation time per sweep and the cost of the trajectory generated after 100 sweeps for different action grid resolutions on the two link swing up problem. To keep the test duration reasonable, the state resolution was set to $60 * 60 * 60 * 60 =$

action resolution	seconds/sweep	trajectory cost
5x5	204	5.1
10x10	865	3.4
15x15	1958	3.3
20x20	3596	3.3
25x25	5659	3.3
1 random action	24	3.3

TABLE I
EFFECTS OF ACTION GRID SIZE.

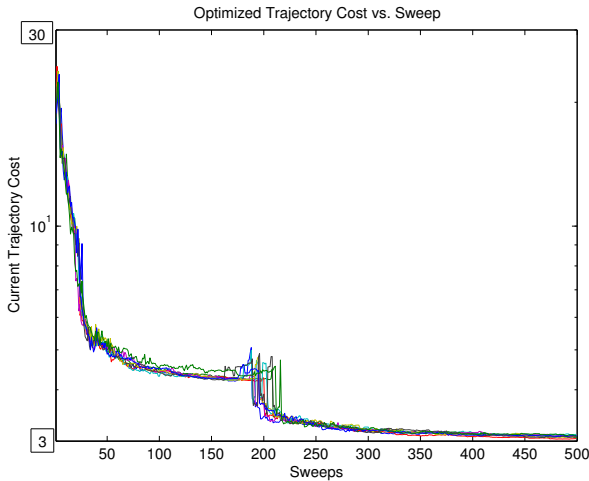


Fig. 8. Comparing convergence curves for multiple runs. Note the log scale on the vertical axis.

12,960,000 total states and as a result the best trajectory cost achievable was higher (3.3). While randomly selecting actions, a sweep took 24 seconds on a hyper-threaded P4 3GHz computer with 3 gigabytes of memory, producing a trajectory with a cost of 3.3 after 100 sweeps. Discretizing the actions with a 5x5 grid resulted in a trajectory with a cost of 5.1 after 100 sweeps. Each sweep took 204 seconds. The table shows how the computational costs grow with increased action grid resolution. Grids 15x15 and larger produced the same quality solution as random search. Using a grid has a greater computational cost than sampling actions randomly, and does not produce significantly better answers, at least for this simple example and the grid sizes explored.

Do random updates cause erratic convergence? To explore whether the random Bellman updates caused convergence to be erratic, we conducted multiple runs of the two link example at the 100^4 resolution. We used the policy computed on each sweep to produce a swing up trajectory. The cost of that trajectory is plotted vs. sweep number in Fig. 8 for 8 runs. The convergence curves are tightly clustered and surprisingly repeatable. The convergence curve for the 148^4 resolution is also plotted in the same figure, and is indistinguishable from the other curves.

Should we consider more than one random action on each Bellman update? Fig. 9 shows similar convergence

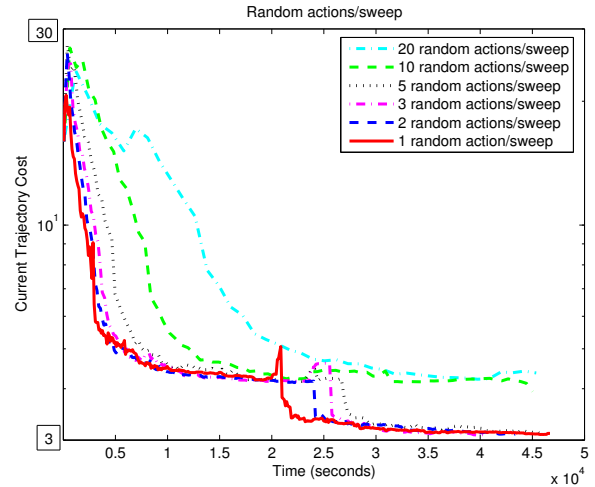


Fig. 9. Effect of changing the number of random actions per Bellman update. Note the log scale on the vertical axis.

curves plotted against wall clock time (to allow comparison) for sampling 1, 2, 3, 5, 10, and 20 random actions per Bellman update. Most of the time the curves for 1, 2, and 3 random actions per update lie on top of each other, indicating that any speedup due to sampling more actions is compensated by the higher computational cost. The single action curve (red solid) is the lowest (fastest convergence) most of the time when the curves do separate. More random actions per update (5, 10, and 20) cause slower convergence. We do not see any advantage to sampling more than one action per Bellman update in this example. These results support minimizing computation per Bellman update. Our philosophy is to minimize the work done to optimize an action on any one sweep, because the value function estimates used in the optimization may change, and the local optimization process will finely tune an action over several sweeps if the value function estimates have stopped changing.

Three link pendulum example: We were not able to reliably get convergence on the 3 link example due to limited state resolution: $22 * 22 * 22 * 22 * 38 * 44 = 391,676,032$ states. The discretized states were bounded by $-5 < \theta_1 < 1$, $-3 < \theta_2 < 3$, $-3 < \theta_3 < 3$, $-10 < \dot{\theta}_1 < 20$, $-35 < \dot{\theta}_2 < 35$, and $-35 < \dot{\theta}_3 < 35$ where $\theta_i = 0$ is upright. The three actions (torques) were bounded by ± 10 Nm. As in the previous examples, the one step cost function was a weighted sum of the squared position errors and the squared torques: $L(\mathbf{x}, \mathbf{u}) = 0.1 * T * (\theta_1^2 + \theta_2^2 + \theta_3^2) + T * (\tau_1^2 + \tau_2^2 + \tau_3^2)$.

However, we did apply local trajectory optimization (DDP) to the trajectory generated by the computed policy after each sweep. The best locally optimized trajectory produced had a cost of 1.9 and is shown in Figures 10 and 11. Future work will explore adaptive sampling of states to more effectively allocate representational resources.

IV. DISCUSSION

Related Work: Although we failed to find this approach

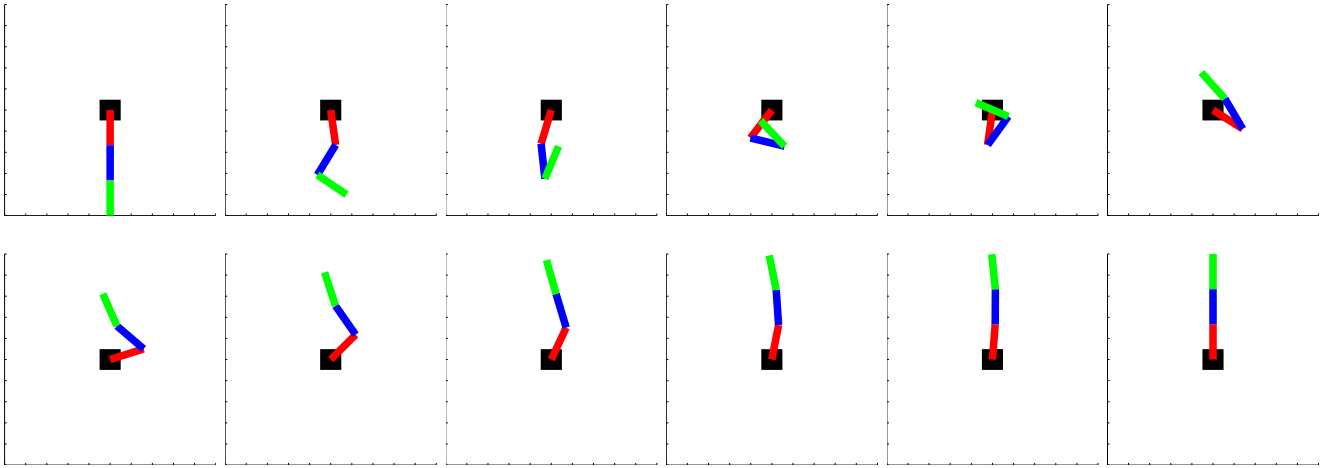


Fig. 10. Configurations from the simulated three link pendulum optimal trajectory at times 0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1sec., and the end of the trajectory.

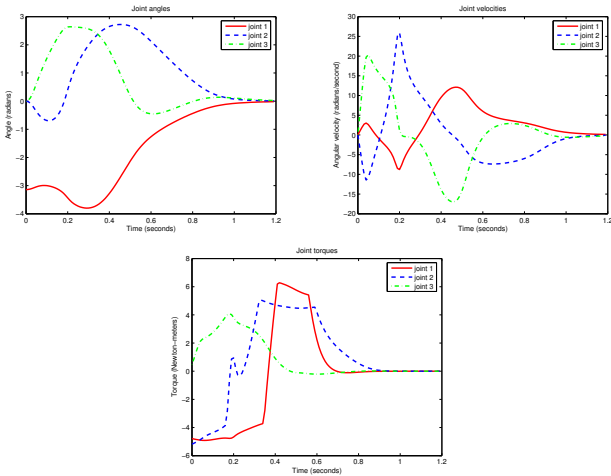


Fig. 11. The simulated trajectory for the three link swing up resulting from dynamic programming followed by local trajectory optimization (DDP).

in the dynamic programming literature, we would be very surprised if others had not made the same or similar observation. Jacobson and Mayne pointed out that actions could be optimized using a wide variety of algorithms in Differential Dynamic Programming, a trajectory optimization procedure derived from dynamic programming [7]. We draw inspiration from this work. Function optimization approaches based on random search such as simulated annealing are especially relevant to this work. References [9] and [10] apply different versions of evolutionary algorithms, a form of random search, to action search in dynamic programming. Reference [11] presents an approach that adaptively samples a discrete action space in dynamic programming.

We are taking advantage of the repetitive sweeps done in dynamic programming for time invariant problems by distributing the action optimization over many value function updates for any particular state. One can view the random

search as being a form of high resolution grid search on the integers. Trying a single pseudo-random number per Bellman update is like trying a single grid element in the grid approach. Thus, Asynchronous Dynamic Programming is another inspiration for this work [12], [13], [14]. Instead of reordering which states are updated, we reorder and spread out in time which actions are checked in the update process. One could also take the view that the *random* search is not necessary, and simply selecting a new element from an action search grid on each Bellman update would also work as well. Reinforcement learning approaches based on simulation or Monte Carlo evaluation have some similarities to the approach proposed here, although often actions are chosen greedily [13], [14]. Exploration, or choosing a random action, is used to avoid local minima in such approaches. One could argue that the approach proposed here is similar to a 100% exploration-based approach, although in our approach we visit all discretized states, rather than follow trajectories as in simulation-based reinforcement learning.

Have we avoided a curse of dimensionality? Rust has shown that random sampling of states can avoid the curse of dimensionality for stochastic dynamic programming problems with a finite set of discrete actions [15]. However, Nemirovsky and Yudin have shown that static continuous optimization problems cannot avoid a curse of dimensionality [16]. The static continuous optimization of $g(\mathbf{u})$ on each Bellman update thus cannot avoid a curse of dimensionality as well. However, these results are theoretical and apply in the limit. It is an empirical question as to whether we can reduce the computational cost of dynamic programming sufficiently to solve practical problems.

It may be that random sampling of actions allows underlying simplicity of the action optimization process to become important. An example of such simplicity is LQR problems, where the dynamics are linear and the cost function is quadratic. In this case there is no curse of dimensionality [17]. Another type of simplicity is when the optimization problem is always

convex.

For many problems with mostly smooth dynamics and a mostly smooth value function, discretizing actions on a uniform grid introduces a lot of unnecessary work, which grows exponentially with the number of action dimensions. This work can be avoided by choosing problem and state specific action discretizations. Sophisticated design of actions for each state is also expensive in human time. Selecting actions randomly and only trying one per Bellman update seems a reasonable compromise. If the underlying problem is simple, selecting a few random actions will adequately explore the space, and it doesn't really matter which actions those are exactly. If the underlying problem is complex or highly discontinuous, a large number of actions must be explored to find good actions. In this case random sampling of actions does not penalize the computational cost relative to the cost of using a large regular grid of actions.

One of the elements that makes random sampling of actions relatively efficient is that a good policy is created relatively quickly, allowing propagation of correct values longer distances in state space. Approaches that examine many actions before the values of subsequent states are "in the ballpark" waste a lot of computation. Only evaluating a few actions per Bellman update allows value iteration to behave more like policy iteration.

There will be a curse of dimensionality in the time required to reach any desired search density (proportional to the volume of the action space $\approx \text{bounds}^{d_u}$) if that is necessary. We note that the use of random actions allows the bounds on the actions to be very approximate. Making them too large merely slows down the random search, but eventually the search density will be arbitrarily large within the bounds.

Convergence: Proving convergence of this algorithm is a straightforward extension of existing convergence proofs for dynamic programming algorithms based on contraction mappings [12]. There are at least two approaches that can be taken. The first is to acknowledge that the random sampling is actually pseudo random, and that a very large grid of possible actions is actually being deterministically sampled. In this case the algorithm is deterministic, with Bellman updates spread over multiple sweeps. Gordon shows how contraction mapping approaches can be used to prove convergence for the simple solution scheme (Section II-A) used as the basis for the algorithm here [18]. His proof can be extended to random sampling by showing that a Bellman update based on one novel action alone does not violate the conditions for the contraction mapping. Random sampling of actions also satisfy convergence conditions requiring that each possible action be evaluated an infinite number of times. Williams and Baird give a convergence proof for asynchronous value iteration when considering a random sequence of single actions [19].

The second point of view is a stochastic one. In this case, convergence in the limit with probability 1 as the number of sweeps increase is desired. The techniques used to show the convergence of Q learning carry directly over to the case described in this paper [20]. Again, assuming finite resolution

of the random actions, the algorithm proposed here satisfies the conditions needed of visiting every state and trying every possible action infinitely often.

Convergence results such as these have little to say about practical performance. The empirical performance seen in experiments is order of magnitudes better than what these convergence results would predict.

Alternate algorithms: There are several variations on the proposed approach that are of interest: 1) applying local optimization (gradient descent) on each Bellman update, 2) prioritizing and reordering Bellman updates, and 3) using other action distributions.

Our attempts to apply local optimization such as gradient descent to the current action slowed down convergence as measured in wall clock time, due to the overhead of local optimization outweighing the improvement in the action selected. In the end we abandoned trying to apply local optimization to the actions.

It is likely that a more sophisticated approach to distributing the action optimization and scheduling Bellman updates will produce even better results. It is clear that the current approach spends a lot of computational effort trying to update actions that are already optimized. Perhaps a failed or small action update should cause a cell to be ignored for several sweeps, while a successful update should give that cell high priority for future updates. We are sure there are other effective approaches to scheduling Bellman updates.

Other action distributions could be considered. For example the actions could be selected from a multidimensional Gaussian centered either at a user selected constant point or at the current best action. The Gaussian could also have a shape, selecting actions with an ellipsoidal distribution. The variance of the Gaussian could be held constant or decreased over sweeps. This approach eliminates the needs for hard bounds on the action distribution.¹

Applying this approach to the discrete action case: Given the power of random search shown here in the continuous action case, it is interesting to consider randomly sampling actions in the discrete action case of dynamic programming [21], [22].

V. CONCLUSION

This work shows the benefits of not discretizing actions and using random search during dynamic programming. Distributing the action optimization for a value function entry over many Bellman updates results in a much more efficient optimization process. We globally optimize actions by testing some number of random actions on each Bellman update. We avoid searching over a discretized grid of multidimensional actions, and produce finely tuned rather than coarse actions. This approach is easy to apply, and the bounds for the actions do not need to be carefully chosen. With this approach we successfully solved two, four, and six dimensional problems at a variety of resolutions with up to 480 million discretized states.

¹This approach was suggested by Martin Stolle.

ACKNOWLEDGMENT

This material is based upon work supported in part by the National Science Foundation under NSF Grant ECS-0325383 and the DARPA Learning Locomotion Program.

REFERENCES

- [1] R. Bellman, *Dynamic Programming*. Dover, 2003.
- [2] D. P. Bertsekas, *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.
- [3] J. Si, A. Barto, W. B. Powell, and D. W. II, *Handbook of Learning and Approximate Dynamic Programming*. IEEE, 2004.
- [4] S. Davies, "Multidimensional triangulation and interpolation for reinforcement learning," 1996. [Online]. Available: cite-seer.comp.nus.edu.sg/56687.html
- [5] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally weighted learning," *Artificial Intelligence Review*, vol. 11, pp. 11–73, 1997.
- [6] F. L. Lewis and V. L. Syrmos, *Optimal Control, 2nd Edition (Hardcover)*. Wiley-Interscience, 1995.
- [7] D. H. Jacobson and D. Q. Mayne, *Differential Dynamic Programming*. Elsevier, 1970.
- [8] P. Dyer and S. McReynolds, *The Computational Theory of Optimal Control*. Academic, NY, 1970.
- [9] H. S. Chang, H. G. Lee, M. C. Fu, and S. I. Marcus, "Evolutionary policy iteration for solving Markov decision processes," *IEEE Transactions on Automatic Control*, vol. 50, pp. 1804–1808, 2005.
- [10] J. Hu, M. C. Fu, V. R. Ramezani, and S. I. Marcus, "An evolutionary random policy search algorithm for solving Markov decision processes," *INFORMS Journal on Computing*, vol. to appear, 2007.
- [11] H. S. Chang, M. C. Fu, J. Hu, and S. I. Marcus, "An adaptive sampling algorithm for solving Markov decision processes," *Operations Research*, vol. 53, pp. 126–139, 2005.
- [12] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation—Numerical Methods*. Prentice Hall, 1989.
- [13] —, *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.
- [14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
- [15] J. Rust, "Using randomization to break the curse of dimensionality," *Econometrica*, vol. 65, no. 3, pp. 487–516, 1997. [Online]. Available: cite-seer.ist.psu.edu/rust96using.html
- [16] A. S. Nemirovsky and D. Yudin, *Problem Complexity and Method Efficiency in Optimization*. Wiley, New York, 1983.
- [17] J. Rust, "Dynamic programming," in *New Palgrave Dictionary of Economics*, 2006.
- [18] G. Gordon, "Approximate solutions to Markov decision processes," Ph.D. dissertation, Carnegie Mellon University, 1999. [Online]. Available: cite-seer.ist.psu.edu/gordon99approximate.html
- [19] R. J. Williams and L. C. Baird, III, "Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems," Northeastern University, Tech. Rep. NU-CCS-93-11, 1993. [Online]. Available: cite-seer.ist.psu.edu/williams93analysis.html
- [20] C. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.
- [21] S. M. LaValle, "From dynamic programming to RRTs: Algorithmic design of feasible trajectories," in *Control Problems in Robotics*. Springer-Verlag, 2002, pp. 19–37.
- [22] D. Burfoot, J. Pineau, and D. Dudek, "RRT-Plan: a randomized algorithm for STRIPS planning," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2006.