

Efficient Value Iteration Using Partitioned Models

David Wingate

Computer Science Department
Brigham Young University
Provo, Utah 84602
Email: wingated@cs.byu.edu

Kevin Seppi

Computer Science Department
Brigham Young University
Provo, Utah 84602
Email: kseppi@cs.byu.edu

Abstract—In order to solve large-scale value iteration problems, more intelligent allocation of computing time is needed. We introduce the idea of an information frontier, which allows us to identify maximally productive regions of the problem space. We present a *potential information flow* metric which allows us to quantify the frontier precisely. We also introduce a partitioning scheme, which effectively combines with the flow metric to reduce the complexity of problematic operations. The framework is powerful, and can be used to parallelize value-iteration, effectively manage memory in large-scale problems, or further multi-agent cooperative solution methodologies. A complete algorithm is developed and successfully tested on several problems. Experimental evidence is presented which demonstrates the efficacy of the approach.

I. INTRODUCTION

Value iteration is a popular technique for solving reinforcement learning problems. However, most implementations of value iteration do not scale well for problems involving large numbers of states. In order to facilitate the solution of large problems, this paper describes a new characterization of the changes a value function estimate experiences over time. It uses the insights gained to analyze common inefficiencies in value iteration, and presents a new algorithm designed to remedy the problems. The algorithm is named EVA, which is short for “Efficient VAlue iterator”.

The goal of many reinforcement learning algorithms is to compute the value function of a problem. The value function typically has the form

$$V(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a' \in A} V(s', a')$$

where $V(s, a)$ denotes the value of taking action $a \in A$ from state $s \in S$, $R(s, a)$ is the immediate reward of the same transition, $T(s, a, s')$ is the probability of transitioning from s to s' using a , and $\gamma \in [0, 1)$ is the discount factor.

The process of value iteration allows an algorithm to start with any estimate of the value function (denoted by \hat{V}), and, through a series of updates, converge to the true value function (denoted by V^*). A series of updates which tries every action from every state converges by a factor of γ [1].

The principal observation that motivated this paper is the fact that, in a naïve implementation of value iteration, most updates to the value function do not help (of course, other algorithms also strive to remedy this problem. These are discussed in section II). This is because most algorithms iterate over all $s \in S$ and $a \in A$ to update $V(s, a)$. This traditional

approach is inefficient because most updates do not utilize accurate information. Since the value of state s depends on the values of its successor states, if the values of the successor states are incorrect, updating $\hat{V}(s)$ will not move it any closer to $V^*(s)$.¹

This paper therefore develops a theoretical framework designed to optimize the allocation of CPU time, and presents a companion algorithm which uses the framework. The result is powerful enough to facilitate other interesting applications, such as efficient parallelization of value iteration, efficient memory management, and the possibility of multi-agent solution methodologies, although this paper will only focus on enhancing the single-processor / single-agent case.

Section II presents the reasons for the potential flow metric, describes why a partitioning scheme is desirable, and attempts to build intuition regarding the notion of an information frontier. It also describes how this work relates to other prominent techniques. Section III introduces the notation used throughout the paper, and section IV describes our algorithm in detail. Section V refines the algorithm ideas by discussing how to effectively partition the problem, as well as presenting a brief complexity analysis. Section VI explains our experimental setup. Section VII describes our results, and section VIII presents conclusions and ideas for future research.

II. MOTIVATION

The motivation behind the EVA algorithm is clear: avoid updates to the value function that do not help. First, a way is needed to distinguish between updates that will help and those that won't. Second, a way is needed to focus computational effort on the update of maximum utility. Third, the resulting complexity must be managed.

First, we define the notion of the *potential information flow* metric. This flow metric quantifies the amount of information that moves when any given update is executed. In order to minimize time to convergence, information must flow as fast as possible. Thus, the metric aids in deciding where computational effort should be expended. Describing the flow metric in detail is the subject of the next subsection.

Once the metric is in place, computational resources can be focused on profitable regions simply by selecting updates with

¹Although it is true that value iteration is a contraction mapping, it is only a contraction mapping in max norm, which implies that after any given round of value iteration \hat{V} will have *globally* approached V^* . No guarantee is made for the value of any individual state.

maximum values according to the metric. Since value iteration is an off-line technique, arbitrary updates can be executed.

Because of the sizable memory requirements this metric adds, the paper combines it with the idea of a *partition* with the flow metric, which allows a designer to explicitly control the trade off between space-time complexity and computational efficiency. It accomplishes this by limiting the number of states that need to be tested for potential flow by aggregating them into partitions, and then checking the potential flow between partitions. At one end, using EVA with a single partition that encompasses the entire problem space becomes equivalent to traditional value iteration. At the other end, the algorithm may be used with one partition per state, which becomes equivalent to computing a full model inverse.

A. The information frontier

Consider a maze world with a single absorbing goal, where the only $R(s, a) \neq 0$ in the whole problem is obtained by transitioning into this goal state. Early in the value iteration process, updates to states near the start of the maze won't be very profitable, because their successor states don't have correct values. But updating the successor's values won't help either, because *their* successors don't have correct values. The only states in the problem with correct values are the ones nearest the goal. Over time, the situation changes. The successors of states near the start still don't have correct values, so updating the start states won't help. But now updating the states near the goal *also* won't help, since their values have converged correctly. Somewhere in between the start and the goal is a region where updates *would* be profitable, where updates would move the value function closer to its optimal form.

We use the term “information frontier” to refer to this boundary between information and non-information. The information frontier can be thought of as series of waves which propagate away from the primary reward in a problem and throughout the rest of the problem space. Formally, the wave is $d\hat{V}/dt$.

Of course, the wave only “moves” when an update to the value function is executed. Often, we will be interested in characterizing the potential movement any given update *could* produce, if executed. We will denote this as the *potential information flow*. The formal definition is given in section III.

The potential flow metric quantifies the information flow frontier of the problem. Any algorithm which executes value function updates at the maximum of this frontier will be guaranteed to move the most amount of information possible. This fact can be used to reduce the overall time needed for the value function to converge, or it can allow information to flow throughout a problem as quickly as possible (resulting in a policy which is decent, though potentially sub-optimal).

B. Related work

Several other algorithms have concerned themselves with efficient information management. The closest algorithm to EVA is Moore and Atekeson's prioritized sweeping algorithm

[2]. They define a metric which describes how “interesting” an observation is, and use that to prioritize other related observations. However, no attempt is made to explicitly bound the complexity of the priority queue (in fact, they explicitly state that, in practice, the algorithm will never gather enough information to exhaust memory). Additionally, since the algorithm is on-line, the agent is strictly reactive, responding as information enters. EVA is strictly off-line.

Singh and Sutton's eligibility traces are also similar [4], although directed at a Q-learning environment. They store a window of experience and an eligibility factor λ which decays over time. As information is collected, it is quickly propagated through the window, discounted by λ . The principal difference here is that no effort is made to find a globally optimal sequence of updates, since the assumptions of Q-learning are not amenable to forcing a particular transition to occur.

III. NOTATION

This section introduces the notation needed to describe the algorithm in the next section, where we will explain why each term is needed, and what it is used for. Note that all of the partitioning notation has been structured in terms of sets. This has been done to emphasize the fact that although EVA's partition implementations are meta-grids, the way states are placed into partitions does not necessarily have anything to do with the “distance” between states.

Let S be the set of states in the problem and A be the set of actions. $N_s = |S|$ is number of states in the problem and $N_a = |A|$ is the number of actions. Let

$$T : S \times A \times S \rightarrow \mathfrak{R}$$

be the *transition probability matrix* of the problem. Define the value function as

$$V(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a' \in A} V(s', a')$$

For convenience, we also define

$$V(s) = \max_a V(s, a)$$

Define the *potential information flow* of the transition (s, a) as

$$H_s(s, a) = \left(R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s') \right) - V(s)$$

Note that the potential information flow is a gradient between the information that \hat{V} contains and the information that \hat{V} *could* contain. It should be noted that this is *not* the same as the 1-step temporal difference of the value function. H_s describes the *potential* change that any given update to the value function could cause, and not the *actual* difference between the value function at two different times. For convenience, we also define

$$H_s(s) = \max_a H_s(s, a)$$

Let P be a set of partitions which denotes a particular partitioning of the state space, and let $N_p = |P|$ be the number

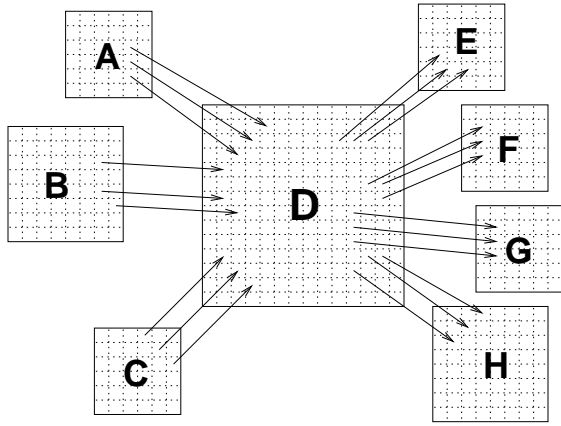


Fig. 1. An illustration of state dependencies for an imaginary problem. Boxes represent partitions, and lines represent some of the transitions between states. Partitions A, B and C all contain states which transition into D, and therefore depend on values inside of D. These transitions form the set $SD_p(D)$. The partitions A, B and C therefore depend in general on D (i.e., $A, B, C \in PD_p(D)$). D has states which depend on states in E, F, G and H.

of partitions. Let each $p \in P$ be a set of states. Let $P_s : S \rightarrow P$ be the mapping of states to the partitions that contain them. All P must tessellate the set S by obeying two properties:

$$\bigcup_{p \in P} p = S \wedge \forall p_1, p_2 p_1 \cap p_2 = \emptyset$$

Define the *state dependents of a state* as

$$SD_s(s) = \{s_0 : \exists_a T(s_0, a, s) \neq 0\}$$

This is the set of all states who have some probability of transitioning to s , and therefore whose value depend on the value of s . Define the *state dependents of a partition* as

$$SD_p(p) = \left\{ \bigcup_{s \in p} SD_s(s) \right\}$$

This is the set of all states whose value depends on some state in the partition p . See Figure 1 for an illustration of the state and partition dependents. Define the *partition dependents* of a partition as

$$PD_p(p) = \{x : \exists_{s \in x, s' \in p, a} T(s, a, s') \neq 0\}$$

This is the set of all partitions that contain at least one state that depends on the value of at least one state in p . Define the potential information flow between two partitions as

$$H_p(p, p') = \max_{s \in PD_p(p)} H_s(s)$$

Note that, in general, $H_p(p, p') \neq H_p(p', p)$. Define the potential flow of a single partition as

$$H_p(p) = \max_{p' \in P} H_p(p, p')$$

Finally, define I_c as the *informational complexity* of a problem (with respect to a given partitioning). I_c represents the average number of times each partition must be visited. W_c represents the average number (per visit) of times each partition must be “washed” to drive all potential information out.

EVA: The Basic Algorithm

Partition the space. Process any desired topological re-mappings. Initialize all potential functions. Select an initial partition p by finding the partition containing the highest $R(s, a)$.

Repeat

- 1) Drive all potential information out of p :
 - let $h = 0$
 - repeat $\forall s \in p, a \in A$
 - update $V(s, a)$
 - $h = \max(h, \Delta V(s, a))$
 - until $h < \epsilon$
- 2) Update potential flow of dependent states:
 - $\forall s \in SD_p(p)$ compute $H_s(s)$
- 3) Update potential flow of dependent partitions:
 - $\forall p' \in PD_p(p)$ update $H_p(p', p)$
- 4) Select the next partition p :
 - $p = \arg \max_{p'} H_p(p')$

until $\max_p H_p(p) < \epsilon$, or until some other stopping criteria is reached.

Fig. 2. The EVA algorithm.

IV. ALGORITHM DESCRIPTION

Preprocessing

The algorithm assumes an initial estimate \hat{V} where each $V(s, a)$ is 0. This is very important, since the information flow metric relies on positive values to direct effort. Practically, this poses little problem: a linear scaling of the reward structure such that all $R(s, a) > 0$ will not change the resulting policy.² Since $\hat{V} = 0$, this implies that $H_s(s, a) = R(s, a)$ for all states and actions, which implies that the initial partition selected is the p containing the largest $R(s, a)$.

Obviously, the first step is to partition the state space. This can be as simple or as complicated as necessary. Because the largest cost of the algorithm comes from transitions involving states from different partitions (or “cross-partition transition”), the next section has been devoted exclusively to this issue.

Once partitioned, several mappings that the algorithm depends on must be computed (this includes the terms P_s , SD_s , SD_p , and PD_p , as defined in section III). We assume that the transition probability matrix T is given (or that it can be easily computed). In all of our experiments, these terms were trivially computed while EVA computed T .

²Note that the H_s metric may produce a negative value. These may be ignored, as they are the result of considering a sub-optimal action from a state.

Step 1: Drive all potential information out of partition p .

We perform naïve value iteration on p , repeatedly updating the value function until no transition changes value. If any transition changes value while updating p , every transition in that partition must be processed at least one more time. Once $\Delta \hat{V} < \epsilon$ in p , all information has been driven out. Thus, $H_p(p) < \epsilon$, $\forall_{p'} H_p(p, p') < \epsilon$, and $\forall_{s \in p} H_s(s) < \epsilon$.

The goal in the next two steps is to compute the maximum of H_s anywhere in the state space, so that the partition with the highest potential flow can be processed. Since the only part of \hat{V} which has changed is within p , only partitions that contain states which depend on p must be examined as candidates for the maximum.

Step 2: Update the potential flow of dependent states

This is accomplished by iterating over all $s \in SD_p(p)$ and updating $H_s(s)$. While iterating, it is useful to be cognizant of the next step, and update H_p of each s concurrently to H_s .

Step 3: Update the potential flow of dependent partitions

Even though $H_p(p, p') < \epsilon$, the converse is not necessarily true. Each $H_p(p', p)$ must be updated for every $p' \in DP_p(p)$.

Step 4: Select the next partition.

The next p is selected by evaluating the potential flow of each partition, and selecting the maximum. This can be trivially accomplished using some implementation of a priority queue.

The algorithm terminates when $\max_p H_p(p) < \epsilon$, or when some other stopping criteria is reached.

The preceding algorithm accomplishes the three principal goals explained in section II. H_s differentiates those updates which help from those which don't. Selecting the partition containing the highest H_s allows us focus effort effectively, and the partitioning overhead manages resource requirements.

V. ALGORITHMIC CONSIDERATIONS

A. Partitioning the problem

The most computational overhead that is introduced by the EVA algorithm occurs when an agent has finished processing a partition p , and must select the next partition to work in. This involves updating the potential flow estimates for every state $s \in SD_p(p)$. It is therefore important to minimize the number of these cross-partition transitions. Conceptually, the problem is best visualized as minimizing the surface-area-to-volume ratio of the partition geometries.

Several approaches are possible. Since a discretized state space with a transition probability matrix can be viewed as a directed graph with weighted edges, techniques from graph theory can be leveraged to determine an optimal partitioning scheme (note that the number of states per partition does not have to be equal). For continuous time problems, it may also be possible to adjust the timestep of the control at each vertex. In regions where the default timestep takes a transition too far (resulting in a cross-partition transition), it may behoove the

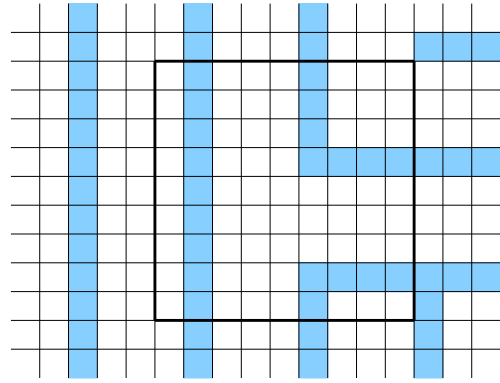


Fig. 3. The informational complexity of a problem is defined as the number of visits that will have to be made in order for all information to wash through all paths. The complexity is a function of the partitioning. Shown is a partition with a complexity of 4.

algorithm to dynamically scale the timestep down as much as possible in an attempt to keep the transition local. This technique presents no difficulties from a stability or accuracy perspective.

For each of the problems in section VI, partitioning was simply done by overlaying the initial discretized state space with another partition grid.

B. Informational complexity

Another interesting way of viewing the partitioning problem is in terms of informational complexity. Consider the diagram of the maze fragment in Figure 3. Note that although the corridors are 3 states wide, the partition is 9 states wide, and even though 9 is a multiple of 3, the partition is not aligned to corridor boundaries. The partition shown will have to be processed four times, since information will flow through it along four different paths, at four different times. This is probably true of many partitions in any given maze – some may be worse, and some may be better, but if the partition size is large, computational cycles may be wasted.

For some problems (such as the maze example), the complexity problem can be solved *a priori* simply by using a regular gridded partitioning scheme, and selecting a suitable partition size. However, it seems that the only *general* solution using regular grids is to test various partition sizes to see what works; by that time, the problem has been solved numerous times. Unfortunately, Figure 7 seems to indicate that I_c is a highly unpredictable function.

C. Complexity analysis

Let L denote the length of the longest sequence of reverse transitions necessary to travel from the largest reward to the most remote state. For every round of naïve value iteration, information back propagates one transition. In order to ensure that information propagates through the entire problem space, at least L iterations must be performed. Each iteration processes every action and every state, resulting in $\mathcal{O}(ALN_s)$. In pathological problems, $L = N_s$, resulting in $\mathcal{O}(AN_s^2)$.

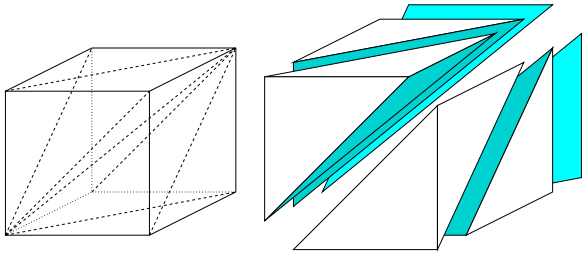


Fig. 4. The Kuhn triangulation of a (3d) cube. A d -dimensional hypercube is tessellated into $d!$ simplices.

Let $b = \frac{N_s}{N_p}$ be the average number of states per partition. Using this, the complexity of EVA becomes $\mathcal{O}(N_p b I_c W_c A \log N_p)$, or $\mathcal{O}(N_p I_c b^2 A \log N_p)$.

Thus, EVA is far more efficient when partitions are small, because it is only squared in the number of states per partition, as opposed to the total number of states. Of course, EVA also introduces additional complexity in order to deal with the extra partition bookkeeping. Many aspects of the algorithm can be efficiently implemented with suitable data structures, which accounts for the $\log N_p$ additional complexity.

D. Notes on Proving Efficiency

Unfortunately, in general $H_s \neq V^* - \hat{V}$. If it were, it could be easily proven that executing updates according to H_s is optimal. However, it can still be shown that selecting the update that maximizes H_s changes the value function estimate by the largest amount possible for any single update (this is possible because we always start with estimates of 0, and we scale the reward structure to be strictly positive; together, they imply that \hat{V} is always less than V^*). Unfortunately, this does not imply that EVA performs an optimal *sequence* of updates. Counter-examples can be easily found to demonstrate this.

VI. EXPERIMENTAL SETUP

The EVA algorithm was validated by running it against several problems of differing complexity. Success was measured by the amount of time it took to complete the value iteration process, and by how accurate the resulting value function was. It performed successfully according to each criteria.

Each of the following problems (with the exception of the mazes) are continuous time, and involve continuous action and state dimensions. We leverage part of Muños and Moore’s variable resolution discretization technique to aid with the discretization of each dimension. See [3] for a complete description with comprehensive citations on component elements.

The state space is divided into a regularly spaced grid. A Kuhn triangle is implemented inside of each cell. The cells completely tessellate the space, and the Kuhn triangles completely tessellate each cell. Approximation of the value function is performed by computing exact values at each of the vertices, and interpolating the value across the interior of each cell. Interpolation is performed using barycentric coordinates (and is therefore linear within each simplex). Since barycentric

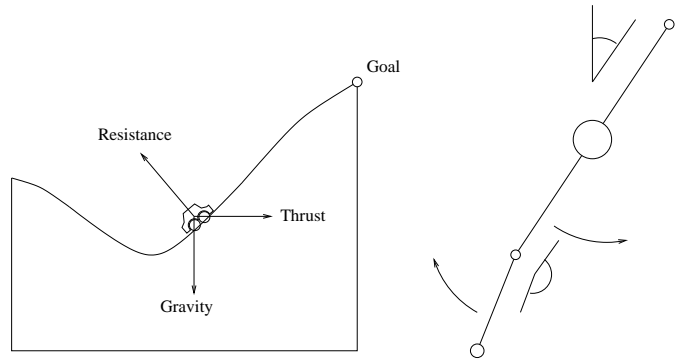


Fig. 5. On the left, the mountain car problem. The car must rock itself back and forth to generate enough momentum to exit the state space. The state space is described by the position and the velocity. On the right, the double-arm pendulum. The state space is described by four variables: θ_1 , θ_2 , $\frac{d\theta_1}{dt}$ and $\frac{d\theta_2}{dt}$. The single-arm pendulum uses the same system dynamics, but only uses θ_1 and $\frac{d\theta_1}{dt}$.

coordinates always sum to one, “doing this interpolation is thus mathematically equivalent to probabilistically jumping to a vertex: we approximate a *deterministic* continuous process by a *stochastic* discrete one” [3] (emphasis in original).

The transition probability matrix was computed by iterating over each vertex (which represents a state s). For each available action a , the system dynamics were integrated using Runge-Kutta and tracked until s' entered a new cell. Since these problems were continuous time, a slightly different form of the value function was used, where γ is raised to the τ , which is the amount of time it took for s' to enter the new cell.

A. The problems

1) *Mountain car*: Mountain car is a two-dimensional control problem, characterized by position and velocity. A small car must rock back and forth until it gains enough momentum to carry itself up to the top of the hill. Any exit on the left-hand side of the problem results in a reward of -1. A gradient reward is given on the right hand side, with the maximum reward of 1 being given if the car exits the state space with zero velocity. A high velocity results in a reward of -1.

2) *Double-arm pendulum*: The double-arm pendulum is a four-dimensional minimum-time control problem (see Figure 5). The agent has a single action available, which represents the torque a central motor applies to the primary link. The secondary link is free-swinging. The agent must learn to balance the second link vertically. Since the problem is minimum-time, bang-bang control is sufficient; two actions are arbitrarily selected, representing positive and negative voltages. The actions are selected (in a similar manner to those of the mountain car) such that the agent does not have sufficient force to move the pendulum from the bottom to the top directly, but rather must learn to rock it back and forth to generate sufficient momentum.

3) *Single-arm pendulum*: The single-arm pendulum is similar to the double-arm pendulum, except that only the main link

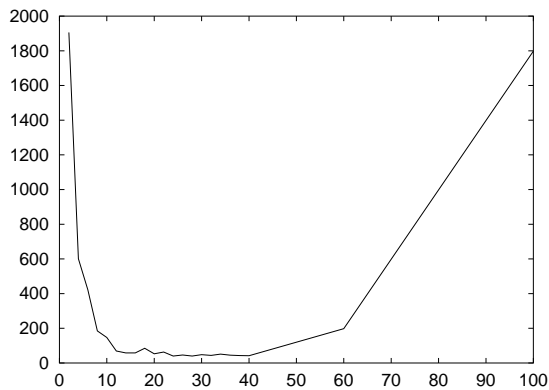


Fig. 6. Performance of EVA against the single-arm pendulum. The horizontal axis represents the square root of the number of partitions. The vertical axis represents the time (in seconds) required for the function to converge. For certain ranges, the problem showed relatively little sensitivity to the number of partitions. The increase in time on the right is explained by a single operation that is linear in N_p , but which could be remedied.

must be balanced. The state space is two-dimensional, being characterized by the angle and angular velocity of the main link. Since the secondary link remains, it makes the problem slightly more non-deterministic than it otherwise would be.

4) *Large mazes*: Several random mazes were generated of varying sizes. The largest was 2000 by 2000, with 10x10 corridors. In each maze, four actions (N,S,E,W) were possible. A single absorbing state was defined as the goal state. Reaching the goal state resulted in a reward of 1.0. The maze was completely deterministic.

VII. RESULTS

There are two major ways in which our algorithm succeeded. Almost without exception, the time necessary for convergence of the value function was dramatically reduced, while the accuracy of the resulting value function and policy were unchanged (the exceptions occurred when extremely large numbers of partitions were used). Secondly, the algorithm surprised us by not processing certain partitions that could not be reached by any information. This is quite a boon: in certain problems with large regions of space that are inaccessible, why process them at all? EVA automatically detects such regions.

A. Better time to convergence

Several experiments were run against the single arm pendulum to examine the effects that different partitioning schemes would have. Each dimension of the state space was divided into 200 bins. Partitions were then constructed by superimposing a meta-grid on top of the state space grid. In the figures shown, the horizontal axis shows the square root of the number of partitions.

Figure 6 shows the performance of EVA as a function of partitioning. The data demonstrates that EVA's performance dramatically increased at first, and then became less sensitive to the number of partitions. The fact that the graph rises again on the right-hand side is interesting. Our implementation of

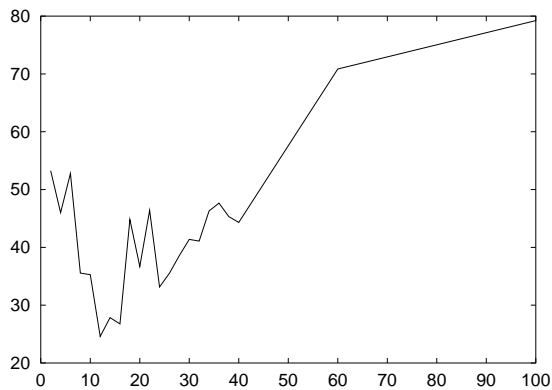


Fig. 7. I_c of the single-arm pendulum as a function of partitioning. The vertical axis represents I_c , and the horizontal axis represents the square root of the number of partitions. No clear pattern is discernible.

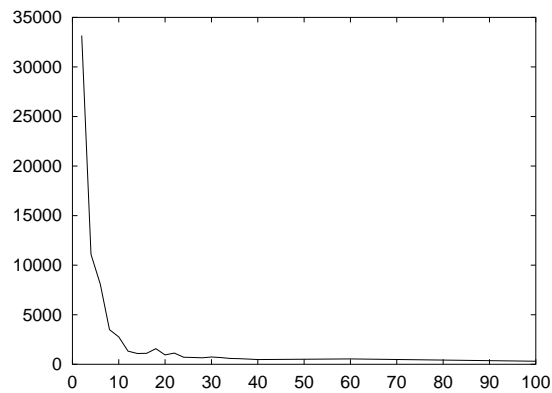


Fig. 8. Shows W_c as a function of partitioning.

EVA requires a full scan of all partitions to determine the max of H_p (this could be remedied by a suitable data structure, such as a Fibonacci heap).

Figure 7 shows I_c for different partitionings. Interestingly, even though the minimum of I_c occurs at around 12 divisions per dimension, the time required for convergence was roughly equal to other experiments. Presumably, this is because although each partition is visited fewer times, they each contain more states.

One of the most spectacular successes of EVA occurred when trying to solve the 2000x2000 maze. Using a 100x100 partition grid, it only required 214 seconds to solve (running on a 2.0GHz PIV). Of course, this represents EVA in it's best light, since the partitioning corresponded exactly to the width of two corridors (i.e., $I_c \approx 2$).

To compare this against the efficiency of naïve value iteration, we ran the experiment. Six hours later, it still had not finished. To determine why, we estimated the amount of time it would have taken the system, given the partitioned performance. I_c for this maze was 1.87, and W_c was 22.75. There were 10,000 partitions, each of which contained 400 states.

EVA therefore performed $10,000 * I_c * W_c * 400 =$



Fig. 9. The resulting mountain car policy. The horizontal axis represents the position, and the vertical axis represents the velocity. The policy in dark regions is to thrust left, and the policy in light regions is to thrust right. Note the aberrant policies in the lower-left hand corner – the agent never processed these partitions.

170,170,000 updates. Since it took 214 seconds to solve, this is equivalent to processing 795,186 updates per second. Assuming that the shortest possible path through the maze was 4,000 states long (the minimal possible length – a generous assumption), approximately $2,000 * 2,000 * 4,000 = 16,000,000,000$ updates would need to be performed. At the same rate as above, it would take roughly 20,121 seconds (or 5.6 hours) to complete. Of course, the maze is far more complicated than the shortest path.

Interestingly enough, when the algorithm was run against the maze with a 200×200 partitioning, it took 585 seconds to finish. We expected better performance, since $I_c = 1$ and $W_c = 6.73$ (indeed, the system only executed 26,919,319 updates). Presumably, the update rate fell from 795,186 updates per second to 46,015 because of the aforementioned $O(N_p)$ operation.

Similarly good results were obtained on all mazes, as well as the mountain car and double-arm pendulum problems. After tweaking and performance optimizations, EVA takes about 6 seconds to solve a 1000×1000 (i.e., 1 million state) single-arm pendulum; for comparison, naïve value iteration takes about 600 seconds. A very exciting result is that EVA solves (to $\epsilon = 0.0001$) a 75,000,000 state version of the double-arm pendulum in only about four hours (to be fair, however, we should note that it takes 8G of RAM to represent the internal structures necessary).

B. Avoiding impossible states

Consider the diagram of the mountain car policy shown in Figure 9. Note that, in the lower-left hand corner, there are clearly three partitions whose states all have an incorrect policy. The car should want to thrust to the right (an action of +10), but instead thrusts left (an action of -10).

What appears to be a bug actually illustrates a benefit of the algorithm that we had not anticipated. The learner did not

learn an incorrect policy, but instead *it did not learn a policy at all*. These three partitions were simply never processed by the learner. The policy seen is the default action – which, due to implementation details, is to thrust left.

The reason that this occurred is because there is no way for information to flow into those partitions. Logically, this makes sense: those partitions reside at the most negative portion of the position dimension (on the far left of the hill), and at the most negative portion of the velocity dimension (maximally negative velocity, taking the car to the left hand side of the hill). However, there is no way for an agent to actually enter those states from any of the other parts of the state space!

VIII. CONCLUSIONS AND FUTURE RESEARCH

EVA has proven to be an extremely efficient value iterator. It has solved problems which are far larger than any other known value iteration based technique, and in less time. More importantly, EVA has provided a first step towards a comprehensive framework that deals with information more efficiently.

Many future research avenues exist. Cycles containing primary rewards pose a problem for learners such as EVA, since they must process transitions multiple times. Some preliminary work in driving loops directly to convergence indicates promise, as it would allow each transition to be processed only once for every loop it is a part of.

Since cross-partition transitions are relatively expensive, more topological remapping theory might also be in order. In the event that an optimal *a priori* partitioning is not available, an interesting scenario would be to allow an agent to partition the problem on-line. Partitioning criteria, stability guarantees, and bookkeeping datastructures would need to be studied.

The partition framework and information frontier characterization presented can be applied to enhance many other types of reinforcement learning. Given a controller that allows an on-line agent to visit arbitrary states, the same framework could be used to guide the exploration of a Q-learning agent. Or, the information frontier could be analyzed and driven by a multi-agent system. Other types of optimization might further enhance learning. Once the information frontier has been identified, the algorithm could be parallelized to allow several different processors to operate on different portions of the state space simultaneously.

REFERENCES

- [1] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, 1995. Morgan Kaufmann.
- [2] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 1993.
- [3] Remi Muñoz and Andrew W. Moore. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *IJCAI*, pages 1348–1355, 1999.
- [4] Satinder P. Singh and Richard S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158, 1996.