PolicyBlocks: An Algorithm for Creating Useful Macro-Actions in Reinforcement Learning

Marc Pickett PICKETT@CS.UMASS.EDU
Andrew G. Barto BARTO@CS.UMASS.EDU
Autonomous Learning Laboratory, Computer Science Dept., University of Massachusetts, Amherst 01003 USA

Abstract

We present PolicyBlocks, an algorithm by which a reinforcement learning agent can extract useful macro-actions from a set of related tasks. The agent creates macro-actions by finding commonalities in solutions to previous tasks. Using these macro-actions, learning to do future related tasks is accelerated. This increase in performance is illustrated in a "rooms" grid-world, in which the macro-actions found by PolicyBlocks outperform even hand designed macro-actions, and in a hydroelectric reservoir control task. We provide empirical comparisons of PolicyBlocks with the Reuse options of Bernstein (1999) and the SKILLS algorithm of Thrun and Schwartz (1995), which elucidate conditions under which each algorithm performs well.

1. Introduction

There are many cases in which a set of several different tasks is given for a single domain. When there is some overlap among the solutions to these tasks, it is possible to use this overlap to transfer knowledge learned from solving previous tasks to new tasks. Examples from real life include when a fencer learns a set of skills (parry, thrust, dodge, etc.) that are useful against any opponent, when a chess master learns a set of opening moves and forks that are useful in several situations, or when a downhill skier learns a set of skills (snowplow, hockey stop) that are useful for several different ski runs. One reason these skills are useful is that they allow an agent to plan or act at a more abstract level than the level of primitive actions. In the reinforcement learning (RL) paradigm, there may be a set of tasks that have identical structure, but with different goals. For example, in Dietterich's Taxi domain (Dietterich 2000), a taxi driver makes deliveries to different locations in an unchanging city. It has been shown (Sutton et al., 1999) that given the right set of temporally-extended actions (called "options") an RL agent can increase its learning rate dramatically.

This paper presents a method called "PolicyBlocks" by which an agent can create useful options automatically. PolicyBlocks creates options by finding commonly occurring subpolicies from the solutions to a set of sample tasks. Because having different skills that have similar effects increases the search space with little increase in functionality, once an option has been created, options with similar effects are no longer considered.

A number of researches have investigated methods for learning skills across several tasks by identifying subgoals. The early approaches by Amarel (1968), and Anzai and Simon (1979) created subgoals by examining solutions to previous problems. More recently, Digney (1998) proposed a hierarchical method in which states that are visited frequently or have a high reward gradient become subgoals. Iba (1989) discussed a heuristic approach for growing macro-operators. McGovern (2002, McGovern and Barto 2001) used several of the ideas for this heuristic for a method of subgoal discovery based on finding commonalities across multiple paths to problem solutions.

Another general technique for finding options is to examine commonalities among sample policies. Rather than focusing on subgoals, PolicyBlocks focuses on sample policy commonalities, and many useful options found by our algorithm have no clearly defined subgoal. Bowling and Veloso (1998) discussed using policies from similar problems for new tasks, and gave a bound for the suboptimality of a previously learned policy that is fixed over a subproblem. Similarly, Bernstein's (1999) Reuse options are probabilistically combined solutions of previous tasks. These methods differ from PolicyBlocks in that they do not explicitly search for common integrated subsequences. The most closely related work is the SKILLS algorithm of Thrun and Schwartz (1995), which creates skills by examining commonalities among the solutions to related tasks. In this work, skills are "learned by minimizing the compactness of action policies using a description length argument on their representations". A comparison of SKILLS and PolicyBlocks is given in Section 5.

This paper is organized as follows: Section 2 contains an introduction to RL, Markov decision processes, and the option framework. Section 3 describes our algorithm for finding useful options. Section 4 demonstrates the effectiveness of the algorithm for a gridworld and for a hydroelectric reservoir control task. Subsequent sections provide discussion of experimental results, limitations of the approach, and conclusions.

2. Reinforcement Learning

In the RL framework, an agent learns by interacting with an environment over a series of discrete time steps. At each time step t, the agent observes the state of the environment, s_t, and chooses an action, a_t from a finite set, which changes the state of the environment to s_{t+1} and gives the agent a reward r_{t+1}. An environment has the Markov property if the reward and the next state depend on only the current state and action, although this dependency may be stochastic, such that there is a fixed probability $P^a_{ss'}$ of going to state s' when taking action a in state s. A *Markov decision process* (MDP) is an environment with a set of states S, a set of actions A, a reward $R^a_{ss'}$ for every triplet consisting of a state s, an action a, and a next state s', and that has the Markov property. The agent's goal is to maximize its *return* or accumulated discounted reward $\sum_{i=0}^{\infty} \gamma r_t$, where $\gamma \in [0, 1)$ is the agent's *discount rate*. The agent does this by creating a *policy*, which is a function that maps states to actions. In stochastic environments, this policy should maximize the expected accumulated discounted reward. A common strategy is to assign a value to every state-action pair, called a Q-value, that is an estimate of the expected return for choosing that action in that state, and following an optimal policy thereafter. This value can be approximated using methods such as Q-learning (Watkins 1989).

PolicyBlocks uses the options framework developed by Sutton et al. (1999) and Precup (2000). Options are temporally extended actions like macro actions. A (Markov) option is a triple $\langle I, \pi, \beta \rangle$, where I is the option's input set, π is a policy defined over all states in which the option can execute, and β is the stopping condition. An option's policy π is a mapping from a subset of the states to a set of actions, which includes both primitive actions and other options. The termination condition β is a mapping from states, s, to probabilities, where the option terminates with probability $\beta(s)$. Once an agent chooses an option, that option's policy is followed until a termination condition is met. For our purposes, β is set to .001 in I, and 1 outside of I, so that an option terminates if it leaves its input set.

3. Automatic Option Creation

Our agent is given optimal policies $L = \{L_1, L_2, \cdots, L_k\}$ for a sample set M of k Markov decision processes. All elements of M have the same set of states S, the same state transition probabilities P, but each element m has its own reward function $R(m)$. Each policy maps states in S to actions in A. In cases where there are several optimal policies, our agent is assumed to be given one according to a fixed tie-breaking policy. Our agent has no direct access to P or $R(m)$. $R(m)$ is assumed to be drawn from a distribution $\mathcal{R}$. Our agent is to use the sample solutions L to create a set O of options that will aid in solving, by Q-learning, further tasks drawn from $\mathcal{R}$.

PolicyBlocks uses a three step process: it *generates* a set of candidate options by finding where the sample solutions match, it *scores* the candidates and chooses the highest scoring option, then *subtracts* what the selected option "explains" from the sample solutions L. The subtraction phase addresses the problem of option redundancy, when several options accomplish very similar tasks. As will be explained later, the score of an option is a heuristic for minimizing the description length of L in terms of O. (This is part of the purpose of the heuristic used by the SKILLS algorithm.)

A *full policy* is a mapping from states to actions. A *partial policy* is a mapping from states to either an action or to a special empty value $\emptyset$ (where $\emptyset$ is not an action). Note that a full policy is also a partial policy. The *size* of a partial policy, denoted $|\pi|$ for partial policy π, is the number of states over which π is defined to be any action other than $\emptyset$.

Definition: The *merge operator* $Mrg(\pi_1, \pi_2) = \pi_3$ is the function mapping a pair of partial policies π_1, π_2 to a partial policy π_3 such that for each state $s \in S$, $\pi_3(s) = \emptyset$ if $\pi_1(s)$ and $\pi_2(s)$ differ, and $\pi_3(s) = \pi_1(s)$ $(= \pi_2(s))$ otherwise.

This function is an extension of the logical "and" operator. Note that Mrg is commutative and associative. We define $Mrg(\{\pi_1, \pi_2, \cdots \pi_n\})$ to be $Mrg(\pi_1, Mrg(\pi_2, Mrg(\cdots \pi_n)))$. Partial policy π_1

contains partial policy π_2 if and only if $Mrg(\pi_1, \pi_2) = \pi_2$.

Using the merge operator, we can find the similarities among the sample solutions. Naively, we can generate each of the 2^k mergings of L (i.e., $Mrg(L_1, L_2)$, $Mrg(L_3, L_5, L_{13})$, etc.). These generated partial policies become the policies for our set of candidate options. This is adequate for small values of k, but when L gets larger, we can generate only a subset of the mergings. Since $Mrg(\pi, \emptyset') = \emptyset'$ where $\emptyset'$ is the policy mapping all states to $\emptyset$, we can begin by generating all pairwise mergings, and from these, generating all triplet mergings (pruning the tree if the mergings become null). In this way, these same general techniques can be approximated for large k.

Before describing our partial policy scoring function, we will describe the subtraction of an option. Once we have selected a single partial policy π from the generated partial policies, we create an option whose input set I is the set of all states $s \in S$ such that $\pi(s) \neq \emptyset$, and whose policy is π for this input set. Since this π was generated by merging the sample policies in L, it must be contained in at least one element of L. We remove each element $l \in L$ that contains π, and replace it with $l - \pi$, where $l - \pi$ is defined below. If $l - \pi = \emptyset'$, then there is no need to add $l - \pi$ to L.

Definition: A *subtraction operator* $\pi_1 - \pi_2 = \pi_3$ is a function mapping a pair of partial policies π_1, π_2 to a partial policy π_3 such that for each state $s \in S$, $\pi_3(s) = \pi_1(s)$ if $\pi_2(s) = \emptyset$, and $\pi_3(s) = \emptyset$ otherwise.

For example, if π_1 maps states 1, 2, 3, and 4 to actions a_1, a_2, a_3, and $\emptyset$, respectively, and π_2 maps these states to $\emptyset$, a_2, $\emptyset$, and $\emptyset$, then $\pi_1 - \pi_2$ will map these states to a_1, $\emptyset$, a_3, and $\emptyset$.

To choose a partial policy from our generated set, we give each partial policy a score, and choose that with the highest score. We would like our options to "explain" or subtract out as much as possible. Therefore, our score is equal to how much the partial policy subtracts from the generated set. Namely the score is the size of the partial policy multiplied by the number of elements in L that contain it. It is possible that this "greedy" approach will not subtract out the maximal amount (i.e., choosing two options by a different method may subtract out more than the top two options chosen by this method), but this scoring system serves as a useful quick metric.

Given a set of options O together with the primitive actions, we can compose these to reconstruct the solutions in L. That is, when the number of states is finite, one can describe a policy compactly by listing which options are applicable, then by listing the remaining state-action pairs individually. Partial policies that subtract the most from the solution set allow for the shortest lists needed to describe L (using the partial policies). Thus, our metric serves as a heuristic for minimum description length (analogous to, but different from the heuristic of the SKILLS algorithm).

Table 1. Pseudocode for PolicyBlocks.

```
given a given set of tasks M
choose n as the desired number of options (or let n = ∞)
use DP or RL to create a set solution set L for M
let an option set O be empty
while (L is not empty) and (|O| < n)
   // generate
   let a candidate set C be empty
   foreach element Π of the power set of L
      let π = Mrg(Π) // Merge Π
      add candidate π to C (if π ∉ C)
      score π // Size of π times frequency in L
   find the candidate π* with highest score
   add π* to O
   // subtract π* from L
   foreach element l of the power set of L
      subtract π* from l
      if l is empty remove it from L
return O as the set of found options
```

Given the procedures for generating, choosing, and subtracting options, PolicyBlocks uses the new L (i.e., with the current set of found options subtracted) to generate a new set of options. This procedure will produce the next "best" option. The algorithm continues producing, selecting, and subtracting options until a specified number of options have been found or until L is empty. Since at least one item of one element of L will be subtracted at each iteration, this process is guaranteed to halt.

There are several ways to use options to solve another task drawn from $\mathcal{R}$ (Sutton et al. 1999). The technique used in this paper is a variant of Macro Q-Learning of McGovern (1997) and Intra-Option Q-Learning of Sutton et al. (1999). First, we augment our set of options with the set of primitive actions so that these are always available. We have a Q-value for each state-action pair s, a where a can be a primitive action or an option whose input set I includes s. Our agent starts each task in a randomly assigned state, and takes actions until it reaches a terminal state, at which point the agent restarts. If the task has no terminal states, the agent simply continues indefinitely. In either case, one may also bound the number of actions taken before a restart. The agent uses an "ϵ greedy" exploration strategy. This means that at each state, with probability ϵ, the agent chooses an action/option at random, and otherwise the agent chooses the action/option with the highest Q-value for that state. If the agent chooses a primitive action a from state s and transfers to state s' with reward r, the agent updates the Q-value thus: $Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha (r + \gamma \max_{a'} Q(s',a'))$ where a' is assumed to be in the augmented action set, and α is the step size parameter. If the agent takes an option o, and after execution of the option gets a cumulative discounted reward of r, and winds up in state s, t time steps later, the update expression is $Q(s,o) \leftarrow (1-\alpha) Q(s,o) + \alpha (r + \gamma^t \max_{a'} Q(s',a'))$. While executing the option, the agent will visit states $s_1, s_2, \cdots s_t$ (where $s_t = s'$). For every state s_i visited while executing the option, the value is updated using the expression $Q(s_i,o) \leftarrow (1-\alpha) Q(s_i,o) + \alpha \left(r_i + \gamma^{t-i} \max_{a'} Q(s',a')\right)$, where r_i is the cumulative discounted reward received after time step i. Furthermore, for each of these states, we update the Q-value for the primitive action taken using the single step reward and state transition for that action.

4. Experiments

To illustrate the behavior of PolicyBlocks, we ran it on an 18 by 18 grid-world task. We then ran PolicyBlocks on a larger hydroelectric reservoir task. For each task, we created 20 sample problems by randomly setting the reward structure (details below). We used policy iteration (with deterministic tie breaking) on these 20 problems to obtain a set of 20 sample policies. The discount factor, γ, was set always to .9.

Given a set of sample solutions, we found options according to PolicyBlocks and SKILLS, as well as the Reuse option. All option sets were augmented with the primitive actions. We compared the performance Q-learning with these options sets with Q-learning using only primitive actions. For the grid-world, we also constructed a set of options by hand that we thought would be useful. These options had the goal of exiting a room by the north, south, east, or west faces (if such an exit existed). For the reservoir problem, it was not obvious to us what might be useful "hand chosen" options, so there is no comparison to these.

Using PolicyBlocks and SKILLS, we found the top 1, 2, 3, 4, and 5 options. Bernstein's Reuse algorithm allows for only one option, and has no parameters. In addition to number of options, SKILLS has the parameter η, which weights the relative importance of *description length* to *performance loss*. In general, a larger value of η results in larger options. We set this parameter to .5, 1, and 2. For all of our comparisons, we set both the step size, α, and exploration rate, ϵ, to .01, .05, or .1. Initially each Q-value was set to 1. We evaluated the performance of these option sets by reporting the average accumulated reward over 100 new tasks for the grid-world and 10 new tasks for the reservoir domain.

Bernstein's Reuse option is a stochastic policy defined over all states (i.e., $I = S$). Given a set of solutions L to k Markov decision processes, generate a set of k stochastic policies SP where $SP_i(a_j|s) = 1$ if $L_i(s) = a_j$ and $SP_i(a_j|s) = 0$ otherwise. The probability of taking action a in state s for the Reuse option is $\frac{1}{k}\sum_{i=1}^{k} SP_i(a,s)$.

SKILLS is more complicated. This algorithm seeks to minimize an energy function $E = PerformanceLoss + \eta DescriptionLength$ by gradient descent. Given a set of optimal Q-values to a set of k problems, $PerformanceLoss$ is the total decrease in value (from optimal for the sample problems) caused by constraining the action set to those defined by the option set for each state. That is, if there is an option defined for a state, not all primitive actions are necessarily available for that state. $DescriptionLength$ is the sum of the size of the options plus k for each state s for which no option is defined. To generate options, one must specify the size of the option set and η. For each option, SKILLS then modifies *usages* (i.e., how useful an option is for a particular sample problem), I, and π in order to minimize E.

4.1. The Grid-world Task

Our grid-world is an 18 by 18 grid (Figure 1) that is naturally divided into rooms with doorways. The grid-world has four stochastic actions: up, down, left, right, which each have a .9 probability of success. If an action doesn't succeed the agent goes into one of the remaining legal directions each with equal probability. A direction is illegal if an obstacle blocks its path. Additionally, the agent has a deterministic "remain" action which allows the agent to stay in its current location. Entering a goal state yields a reward of 10, while all other transitions yield a reward of 0. While performing Q-learning, an agent is reset to a random location upon reaching the goal state. Grid-world sample problems were made by choosing a goal state randomly.

4.2. The Hydroelectric Reservoir Problem

A series of hydroelectric dams are positioned at distant points along a river. Each dam sells electricity to a different city, and each has a reservoir to collect water (Figure 3). Since there is no efficient electricity storage, the city uses and pays for the electricity as

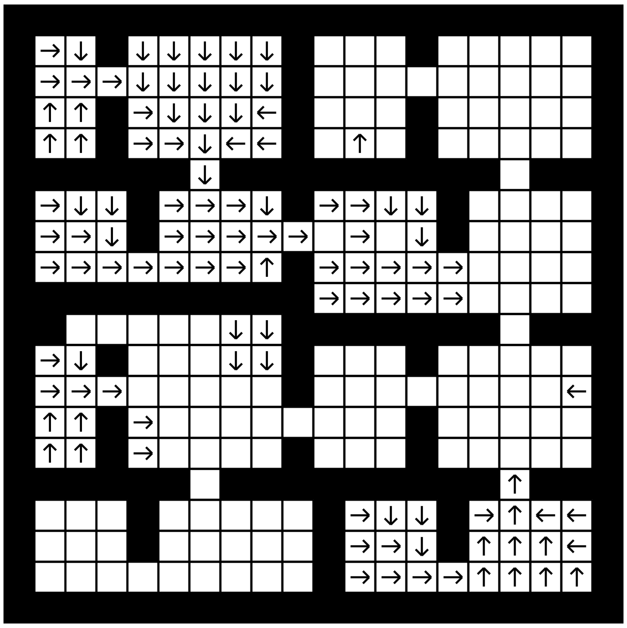

Figure 1. **The grid-world with an example option.** Each black box is an obstacle, and each open box represents a state. The arrows indicate the actions for the subpolicy. Lack of an arrow for a box implies that the subpolicy is not defined for that state. Note that the option is discontiguous, and can be thought of as 8 separate contiguous options. This subpolicy was found in 15 of the 20 sample solutions, and has size 109 giving it a score of $109 \cdot 15 = 1,635$ making it the top option found. Other options found by PolicyBlocks followed this general pattern of taking the agent through 1 or 2 rooms to the door.

soon as it is generated. The price of the electricity is proportional to the city's demand. In addition, due to economic differences, different cities pay more or less for the same amount of electricity and for the same level of demand. Each dam has a reservoir, and an agent must decide which dams to close or open thus producing electricity, letting the water flow down river, and lowering the level of the reservoir. The agent's goal is to maximize its total discounted profit. However, there is no "goal state" as in the grid-world. The 1st reservoir in the sequence gets a constant flow of water. This task is an abstraction of that described by Little (1955).

In our specific example, there are 4 dams, each with a reservoir and a city. Each city i has a variable demand level D_i that can be either *high*, *medium*, or *low*, and a fixed pay rate $P_i \in (0, 1]$ that corresponds to the economic differences. Each dam i has a reservoir water level W_i of either *high*, *medium*, or *empty*. At each decision point, the agent can open or close each of the 4

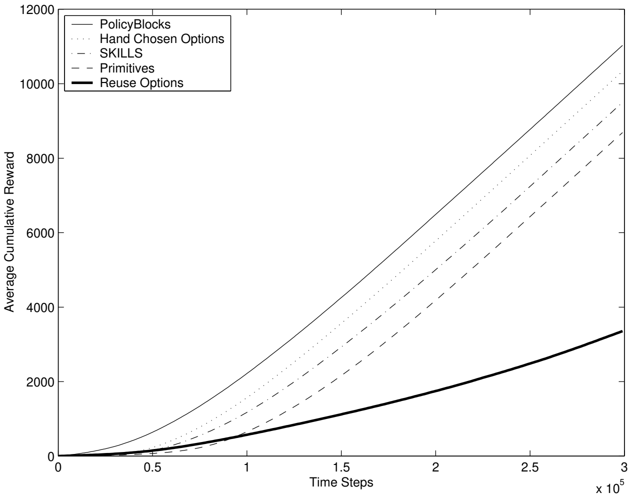

Figure 2. **Option performance comparison for the grid-world.** The x-axis is time steps, and the y-axis is the cumulative reward averaged over 100 MDPs.

dams, resulting in $2^4 = 16$ actions. If the agent opens a dam i, and it isn't empty, a unit of electricity is produced, and the water level is decreased (from high to medium, or from medium to empty). The water flows down river, and the water level for the next reservoir W_{i+1} will be increased for the next decision point. The agent receives a pay from the city equal to its pay rate P_i if demand D_i is medium, twice the city's pay rate if demand is high, and no pay at all if the demand is low. If the agent keeps the dam closed, nothing happens. Between decision points, the reservoirs fill (from empty to medium, or from medium to high) if the dam above them was opened. The top dam always fills. If a dam is already at a high level, and more water is coming from the dam above, the water is uselessly disposed of by flood prevention chutes, and the levels stay the same. The cities also change their demand levels stochastically: each has a .9 probability of remaining at the same level, and otherwise it transitions to an adjacent level (e.g., low to medium). The agent has a discount rate of .9 for each decision point. Since there are 3 levels for each dam, 3 levels for each the city's demand, and 4 cities and dams, there are $3^8 = 6561$ different states.

To apply PolicyBlocks, we consider instances of this problem where the difference is the cities' pay rates P_i which are drawn according to a uniform distribution. Thus, the transition probabilities remain the same across problems, but the reward function varies yielding different optimal policies. For our experiment, we generated 20 sample problems where each P_i was

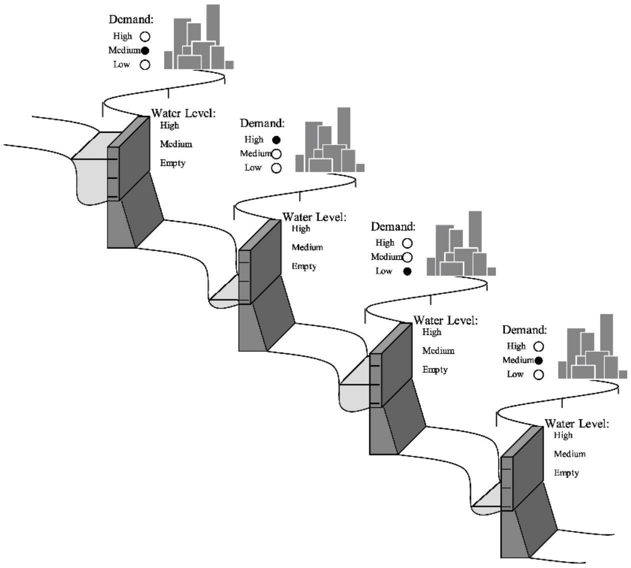

Figure 3. **A sequence of hydroelectric reservoirs and dams.** Each of the 4 dams provides power for a single city. Each city's demand (and pay rate) varies with time. The agent must choose which dams to keep closed or open to produce electricity to sell immediately to its city. Opening a dam also sends some water to the next reservoir downstream. There is assumed to be a steady flow of water into the 1st reservoir.

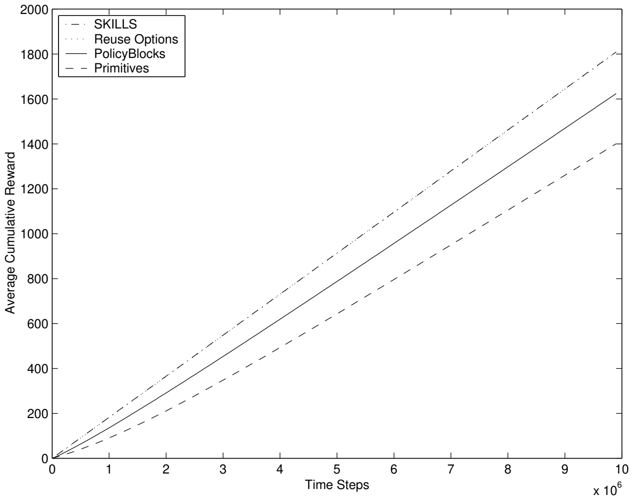

Figure 4. **Option performance comparison for the reservoir problem.** The x-axis is time steps, and the y-axis is the cumulative reward averaged over 10 MDPs. The options generated by SKILLS faired best. A hair's breadth below (nearly atop SKILLS) is the Reuse option. PolicyBlocks performed better than using only primitives, but not as well as the other option selection schemes.

chosen randomly uniformly from $(0, 1]$. We ran Q-learning on this problem on 10 new problems.

4.3. Results

Figure 2 shows results for the grid-world comparison. Over all the parameter settings we tried, the best runs are shown for PolicyBlocks, SKILLS, the Reuse option, hand chosen options, and primitive actions alone. In all cases, option performance, in terms of cumulative reward was maximized among the parameters we tried when $\alpha = .1$ and $\epsilon = .01$. The optimal policy was found when the slope of the line becomes constant. The top 4 options generated by PolicyBlocks outperformed even the "Hand Chosen Options". This was followed by SKILLS, using 2 options and setting $\eta = 1$. Although the Reuse option initially outperformed the use of primitive actions alone, this option failed to learn the optimal policy before 300,000 time steps.

Figure 4 shows results for the reservoir comparison. Again, we show the best runs for each algorithm. The options generated by SKILLS, using 5 options, setting $\alpha = .1$, $\epsilon = .01$, and $\eta = .5$, fared best. Just below that is the Reuse option. PolicyBlocks (setting $\alpha = .1$, $\epsilon = .05$, and using the top 3 options) performed better than using only primitives, but not as well as the other option selection schemes.

Figure 5 shows results for the effects of the number of options for PolicyBlocks on a grid-world. The performance is similar with 2, 3, or 4 options. Below these is the top single option, and at bottom is the performance of the top 5 options. All of these outperformed primitive actions except for the top 5 options, which failed to learn the optimal policy after 300,000 time-steps.

Since there are 2^{20} possible mergings, we were pleased to find that the vast majority of processor time was spent on Q-learning. Creating the options never took more than a few seconds once the sample solutions were found. This is because it was rare that more than 8 partial policies could be merged without having a null result.

5. Discussion

For the grid-world, we believe the Reuse option suffered from lacking a single purpose. For example, some of the sample policies for each of the middle rooms went left, and some went right. Therefore, the Reuse option would spend a lot of time wandering arbitrarily left and right in these rooms.

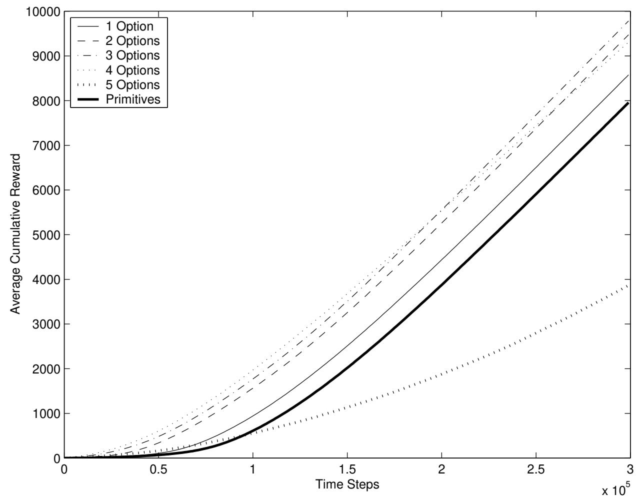

Figure 5. **Effect of Parameters for PolicyBlocks.** The x-axis is time steps, and the y-axis is the cumulative reward averaged over 100 grid-worlds. Using 2, 3, and 4 options has similar performance. Below these is using only the top option, and at bottom is the performance of using the top 5 options.

We believe that PolicyBlocks's options did better than the hand chosen options because they were larger in general. For example, the two rooms in the lower right corner are combined by a single option. The options found by SKILLS were larger still, but perhaps too large, and an agent is not provided with enough choice points when following them. Setting η to be smaller, and using more options creates the problem of introducing redundant options. Redundancy in SKILLS is controlled by its *usage* values, and these can get stuck in local optima during gradient descent.

There are two chief differences in the results for the grid-world and those of the reservoir problem. The first is that the Reuse option did quite well in the reservoir problem while doing worse than even using no options in the grid-world. The second is that the curves have a fairly constant slope for the reservoir problem. At first, we thought that this constant slope might be because 10,000,000 steps was not enough time to learn an optimal policy, but plotting these curves to 100,000,000 steps shows no further increase in the slopes. We believe the two differences may both be due to the relative similarity of the sample solutions for the reservoir problem, compared to that of the grid-world.

For the reservoir problem, the probability that the action for a randomly selected state differs between two of the sample policies is .087, whereas this figure is .191 for the grid-world. This means that solutions for the reservoir problem are much more similar to each other than are those for the grid-world. This is even more pertinent considering that there are only 4 actions for the grid-world, but 16 for the reservoir. Thus, the weakness of the Reuse option for the grid-world becomes a strength in the reservoir problem, since simply using an optimal policy from a sample problem yields a near optimal return for a new problem. Likewise, the amount of redundancy for SKILLS with 5 options is significant and helpful for the reservoir.

The results for the number of options parameter shows that one can have too many or too few options. Having too few options limits functionality. That is, in a certain situation (e.g., a room in the grid-world), there may be several options that are useful for different goals (e.g., going to the left or right door). However, having too many options may introduce redundancy: increasing the search space without increasing functionality. In addition, adding too many options may cause "over-fitting" of the sample solutions, resulting in high bias. This latter effect may be the chief reason for the poor performance of 5 options. This would also explain why using 5 options slightly outperforms both 1 option and using only primitives initially. In both PolicyBlocks and SKILLS, over-fitting can be addressed by limiting the number of options by doing something analogous to cross-validation.

SKILLS and PolicyBlocks both aim to minimize the description length of the sample solutions via options. Although, these algorithms use different measures of description length, the core idea behind these is essentially the same. A more important difference in these algorithms is how they address option redundancy. In SKILLS, this is implicitly addressed by the $PerformanceLoss$ and the *usages*, whereas PolicyBlocks takes a more direct approach via the subtraction operation.

The experimental results suggest that PolicyBlocks is most useful for sets of tasks in which sample solutions can vary widely. This is useful for an agent in a environment complex enough to have a variety of tasks, and where the agent will be long-lived in this environment.

PolicyBlocks can also be used for factored MDPs if the *merge*, *subtraction*, and *size* operations can be defined using a factored representation. However, defining these terms can be far from trivial.

PolicyBlocks has some limitations that are remedied with minor changes. PolicyBlocks assumes that it has time to compute optimal policies beforehand. However, a small modification of the algorithm would allow solutions to be added to L as they are found. PolicyBlocks also assumes that in cases where there are several optimal policies, the agent is given a policy according to a fixed tie-breaking scheme. Without this assumption, the current implementation of PolicyBlocks would fail to find similarities among sub-policies that do essentially the same task. However, a slight modification can make the weaker assumption that we have a set of optimal actions for each state. This can easily be found given either a non-deterministic policy, or the set of optimal Q-values for the sample problems as in SKILLS. Given an action set, we can modify the *merge* and *subtraction* operators to return the intersection and subtraction of the action sets, respectively. The rest of the algorithm remains the same. PolicyBlocks also assumes that the different problems do not differ in dynamics. However, this assumption may be relaxed if the dynamics of the different problems are sufficiently close to each other.

6. Conclusion

We introduced a method by which useful options can be discovered by looking for commonalities among solutions to a set of related tasks. We illustrated this approach by showing its performance on a grid-world task and on a hydroelectric reservoir control task, and comparing its results with that of related option creation methods. Our results are especially encouraging since PolicyBlocks outperformed even hand-chosen options on the grid-world task. Although these were toy problems, we believe that the general methods can be extended to larger tasks. The main ideas of PolicyBlocks are the idea of finding commonalities in solutions, and the idea of subtracting these after they have been "explained". We believe that the utility of these ideas is very general, as they apply to many forms of abstraction (e.g., boosting, greedy set-covering algorithms).

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ECS-9980062. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the National Science Foundation. The authors would like to thank Amy McGovern, Ted Perkins, Mike Rosenstein, Dan Bernstein, and Khash Rohanimanesh for their comments.

References

Amarel, S. (1968) *On Representations of Problems of Reasoning about Actions.* Machine Intelligence 3, vol. 3 131-171.

Anzai, Y., Simon, H. (1979) *The Theory of Learning by Doing.* Psychological Review, 86, 124-140.

Bernstein, D. (1999) *Reusing Old Policies to Accelerate Learning on New MDPs.* Technical Report UM-CS-1999-026, Dept. of CS, U. of Mass., Amherst.

Bowling, M., Veloso, M., (1998) *Reusing learned policies between similar problems.* The AI*IA-98 Workshop on New Trends in Robotics Padua, Italy.

Dietterich, T. (2000) *State abstraction in maxq hierarchical reinforcement learning.* NIPS 12.

Digney, B. (1998) *Learning Hierarchical Control Structure for Multiple Tasks and Changing Environments.* From Animals to Animats 5: SAB.

Iba, G. (1989) *A Heuristic Approach to the Discovery of Macro-Operators.* Machine Learning, 3 285-317.

Little, J. D. C. (1955) *The Use of Storage Water in Hydroelectric Systems.* Opns Res. 3, 187-197.

McGovern, A. (2002) *Autonomous Discovery of Temporal Abstractions from Interaction with an Environment*, Ph.D. Thesis, U. of Mass., Amherst.

McGovern, A., Barto, A. G. (2001) *Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density.* ICML.

McGovern, A., Sutton, R. S., Fagg, A. H. (1997) *Roles of Macro-Actions in Accelerating Reinforcement Learning.* Grace Hopper Celebration of Women in Computing, pages 13-18.

Precup, D. (2000) *Temporal Abstraction in Reinforcement Learning.* Doctoral Dissertation, U. of Mass., Amherst.

Sutton, R. S., Barto, A. G. (1998) *Reinforcement Learning. an introduction.* Cambridge, MA: MIT Press.

Sutton, R. S., Precup, D., Singh, S. (1999) *Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning.* Artificial Intelligence 112:181-211.

Thrun, S., Schwartz A. (1995) *Finding Structure in Reinforcement Learning.* Advances in Neural Information Processing Systems 7.

Watkins, C. (1989) *Learning from Delayed Rewards.* PhD Thesis, U. of Cambridge, England.