Discovering Options from Example Trajectories

Peng Zang PENGZANG@CC.GATECH.EDU
Peng Zhou PZHOU6@CC.GATECH.EDU
David Minnen DMINN@CC.GATECH.EDU
Charles Isbell ISBELL@CC.GATECH.EDU
College of Computing, Georgia Institute of Technology, 801 Atlantic Dr, Atlanta GA, 30308

Abstract

We present a novel technique for automated problem decomposition to address the problem of scalability in reinforcement learning. Our technique makes use of a set of near-optimal trajectories to discover *options* and incorporates them into the learning process, dramatically reducing the time it takes to solve the underlying problem. We run a series of experiments in two different domains and show that our method offers up to 30 fold speedup over the baseline.

1. Introduction

Reinforcement learning (RL) is a widely-studied area of machine learning dealing with sequential decision processes such as Markov Decision Processes (MDPs). Past successes include pole-balancing, maze solving, and helicopter control (Sutton & Barto, 1998). One major challenge in RL is scalability. Traditional algorithms such as value iteration (VI) scale with the number of states which grows exponentially with the number of state variables. As a result, such approaches quickly become intractable, and unfortunately, even simple problems such as basic navigation tasks can have many state variables (*e.g.,* location, orientation, fuel, terrain, *etc.*)

One promising line of work to address this issue is problem decomposition. It has several advantages: (1) it is usually much easier to solve several smaller subproblems than one larger one, (2) decomposition enables us to avoid solving a subproblem multiple times, (3) decomposed subproblems and their solutions provide transfer learning opportunities, and (4) subproblems can often take advantage of state and action abstractions not available to the global problem by taking advantage of its reduced scope.

Appearing in *Proceedings of the 26th International Conference on Machine Learning*, Montreal, Canada, 2009. Copyright 2009 by the author(s)/owner(s).

Automatic problem decomposition thus arises as a crucial goal. The problem is non-trivial because the set of possible subproblems is exponential in the size of the state space, and it is typically difficult to predict the speedup effects of factoring out any particular subproblem. Further, one must balance the cost of finding and solving subproblems with their benefits.

In this paper, we adopt the options framework for problem decomposition (Sutton et al., 1999) and address the problem of automatic option discovery and incorporation. Our contributions are: (1) a method for discovering options with motif-discovery over near-optimal trajectories using a novel heuristic, (2) a technique for estimating the usefulness of a proposed option and (3) a way to compute the transition and reward model of learned options so that they may be injected directly into an MDP as an additional action, enabling fast model based solvers.

In the next section we provide additional background and introduce some notation. We then describe our approach and algorithms. This is followed by analysis and empirical evaluations.

2. Preliminaries

We model the reinforcement learning problem as a Semi Markov Decision Process (SMDP). A SMDP $M = (S,A,P,R,\gamma)$ is defined by a set of states S, a set of actions A, the transition function P, reward function R and discount factor γ. The transition function $P(s,a,s') = \sum_{t=1}^{\infty} Pr(s',t|s,a)\gamma^t$ describes the likelihood of ending in state s' upon taking action a in state s over all possible durations t, appropriately discounted. The reward function $R(s,a)$ specifies the expected, discounted reward accumulated over the duration of taking action a in state s. To simplify discussion we will assume, without loss of generality, that rewards are negative.

A policy, π, is a mapping from states to actions that prescribes what action an agent should take in a particular state. The utility or value of a state, $V^{\pi}(s)$, is the expected,

long-term discounted reward an agent receives assuming it follows policy π from state s. $V^*(s)$ refers to the value when following an optimal policy, *i.e.,* one that maximizes the expected long-term discounted reward.

Problem decomposition for (S)MDPs has been treated under several frameworks (Dietterich, 1998) (Parr & Russell, 1997). In this paper we adopt the options framework (Sutton et al., 1999). An option (I, π, β) consists of three components. The initiation set, $I \subseteq S$, indicates the set of states where the option is available. The policy, $\pi : S \rightarrow A$, dictates the actions to be followed while the option is active. Finally, $\beta : S \rightarrow [0,1]$ denotes the probability of terminating in any particular state.

We assume states are represented by a set of features $f_1 \dots f_n$, and thus $S = f_1 \times f_2 \times \dots \times f_n$. A state abstraction $F \subseteq \{1 \dots n\}$ is a set of indices. $S[F]$ refers to the state subspace induced by the cross product of the features whose indices are in F and is called the abstract state space of F.

Given state abstraction $\tilde{F}$ and F where $\tilde{F} \subseteq F$, we call a function $g : S[F] \rightarrow S[\tilde{F}]$ the *down projection* function. The function $h : S[\tilde{F}] \rightarrow 2^{S[F]}$ is the inverse function mapping a state from the space of $\tilde{F}$ to a corresponding set of states in $S[F]$ and is called the *up projection* function. Sometimes we want an up projection with respect to some state $s \in S[F]$. We will overload the function name h and define this up projection as $h(\tilde{s}; s) = x$ where x_i is $\tilde{s}_i$ if $i \in \tilde{F}$ and s_i otherwise. We call this the up projection of $\tilde{s}$ with context s.

We define a subproblem (M, F, A, ω) as a four tuple consisting of a base SMDP M, state abstraction F, action set A, and a goal $\omega \in S[F]$. F and A must be subsets of the feature space and action space of M. A subproblem induces an abstract SMDP via F and A, which, when solved, yields a solution to the subproblem. We use the term “problem” loosely to refer to both SMDPs and subproblems.

A trajectory T of length k is a sequence of steps $t_1, \dots, t_k$. Each step t_i is a 5-tuple (s, a, s', d, r) where s is the state before the action is taken, a is the action taken, s' is the resulting next state, d is the duration of the action and r is the (discounted) reward received over the duration of the step. An optimal trajectory is one that follows an optimal policy.

We will make use of the Taxi problem, a domain commonly seen in the literature, as a running example. Briefly, the Taxi domain is one in which an agent is a taxi whose objective is to move a passenger from a starting location to a desired destination. The world is a 5×5 grid. There are 4 pickup/dropoff locations each residing in one of the four corners of the grid: `NW`, `NE`, `SW`, and `SE`. The state space is composed of three state features: `taxiLocation`, `dropoffLocation`, and `passengerLocation`. The actions are *North*, *South*, *East*, *West*, *Pickup* and *Dropoff*. The MDP terminates when `passengerLocation` equals `dropoffLocation`. Reward is uniformly -1 except for invalid *Pickup* and *Dropoff* actions which yield -10.

3. Speeding up RL with Options Discovery

We are interested in automatically discovering subproblems to speed up (S)MDP solvers and for the transfer opportunities they offer. Ideally, we want to: (1) find a subproblem that can be solved quickly, (2) solve it, forming an option for performing the subproblem from its solution, and (3) insert the option into the original SMDP to lower its complexity.

The number of possible subproblems is prohibitively large so we would like to prune the set we consider. The reduced set should be significantly smaller but still contain most of the “good” subproblems. To ground our discussion, we have put together some characteristics of “good” subproblems. (1) **Size**: the subproblem should encapsulate a significant chunk of the overall problem. If this were not the case, learning the option would offer little overall savings. (2) **Frequency**: the more frequently a subproblem arises, the more savings the decomposition of the subproblem yields. (3) **Abstraction**: the greater the abstraction the faster we can solve the subproblem.

Our method of pruning the space of subproblems rests upon the following insight: the size and frequency characteristics of “good” subproblems reveal themselves in trajectories; namely, a subproblem of significant size and frequency leaves long, common action sequences that act as “signatures” which can be used to detect the subproblem. By finding these sequences, we can bias our search to those subproblems with significant size and frequency.

In order to judge the usefulness of a candidate subproblem, we need a way of estimating its benefit; that is, how much faster can we solve the overall problem if we factor out this particular piece, solve it, and then solve the rest of the problem. We perform this estimation by using the complexity of VI; however, this estimate requires knowing the maximum solution length (in terms of the number of steps) of the subproblem and the rest of the problem. Here, we again make use of the trajectories as they provide samples of the the subproblem and remaining problem length.

A high-level sketch of our technique is presented in Algorithm 1. We require as inputs a set of trajectories and the SMDP providing the transition and reward models. We further assume that we have or can easily compute from the model, the set of features an action needs (features that affect or are affected by the action). For simplicity we use VI as our baseline (S)MDP solver.

Algorithm 1 *Oplearn*

Require: SMDP M, trajectories T
1: let $subp$, T_{sub}, T_{rem}, $score$ = bestSubproblem (M, T)
2: **if** $score > 1$ **then**
3: let M_{sub} = SMDP induced by $subp$
4: let $subsol$ = recursively solve (M_{sub}, T_{sub})
5: let o = create option ($subp$, $subsol$)
6: let M_{rem} = add option o into M
7: recursively solve (M_{rem}, T_{rem})
8: **else**
9: let F = union state abstraction of actions in T
10: let A = actions seen in T
11: let M' = abstract SMDP(M, F, A)
12: let sol = valueIteration (M')
13: **end if**
14: **return** sol

Algorithm 2 Finding the best subproblem

Require: SMDP M, trajectories T
1: let cas = common action sequence of T
2: let acc = []
3: **for all** $seq \in cas$ **do**
4: let F = union state abstraction of all actions in seq
5: let g = goal search (seq, F)
6: let T_{sub}, T_{rem} = decompose trajectories T
7: let A = union of all actions in T_{sub}
8: let $subp$ = generate subproblem (M, F, A, g)
9: let $score$ = score ($subp$, T_{sub}, T_{rem})
10: add ($subp$, T_{sub}, T_{rem}, $score$) to acc
11: **end for**
12: **return** entry in acc with highest score

Oplearn finds and solves subproblems in a greedy, depth-first manner. We only solve a (sub)problem directly when we estimate further decompositions to be detrimental.

We refer to the SMDP first passed to the algorithm as the **original** problem. In each (recursive) call of the algorithm, the SMDP in the argument is called the **base** problem. If a subproblem is identified and solved, the solution is made into an option which is added into the base SMDP. This produces a modified SMDP with reduced complexity. We refer to this as the **remaining** problem. We refer to trajectories similarly.

In our Taxi example, the original problem would be the full Taxi problem. The first subproblem discovered may be "pickup passenger". If so, our first step is to make a recursive call to solve this subproblem. In the recursive call, "pickup passenger" becomes the base problem. A subproblem discovered for it could be to "navigate" to a particular pickup location. Suppose there are no further decompositions for "navigate" so that it is abstracted and solved directly. The algorithm would then add the "navigate" option into the base SMDP. The remaining problem would be to figure out how to "pickup passenger" with the enhanced action set including the "navigate" option.

Several steps in the algorithm deserve further explanation. We will describe each of these in turn in the following sections.

3.1. Finding the best subproblem

The method for finding the best subproblem is presented in Algorithm 2. It works by generating many candidate subproblems, scoring them, and selecting the highest scoring one. We will first explain how subproblems are formed. The basic progression is to discover common actions sequences, which we then use as seeds to find the goals and action sets that define the subproblems.

We generate common action sequences from the trajectories efficiently using suffix trees. Suffix trees are a technique commonly found in motif-discovery literature. They require only linear time and space to construct (Ukkonen., 1992). The tree is structured such that each path from root to leaf represents a suffix, and thus each path from the root to an internal node represents a subsequence. In addition, the nodes correspond to maximal subsequences in the sense that each edge represents the longest sequence of characters that always follows the prefix represented by the parent node. Thus, by building a generalized suffix tree based on observed action sequences and then traversing the tree structure, we generate all maximal repeated action subsequences in linear time.

Every common action sequence produced by the suffix tree is a seed for finding a candidate subproblem. The process from seed to scored subproblem has several steps. First, the seed is used to determine what state variables are needed by the subproblem, *i.e.,* its state abstraction. This is a simple process of taking the union of the state features needed by each action in the seed.

Next, goal search is performed. A goal that appears frequently is desirable because it means the resulting subproblem will also be frequent. A goal on an abstraction boundary, that is, when the state abstraction required by the actions suddenly changes, is also desirable because it maximizes the benefits of any afforded abstraction and is a natural breaking point. Because abstraction buys us the most speedup, we give it priority. We perform goal search by taking every occurrence of a seed in the trajectories, and extending the sequence until the abstraction "breaks", *i.e.,* when one or more extra state features are suddenly needed. The last state before the abstraction is "broken" becomes a goal candidate. The most frequent goal candidate is chosen as the goal.

Finally, we find the action set by decomposing the trajectories into those that are a part of the subproblem, T_{sub}, and those that are a part of the remaining problem, T_{rem}. The union of all actions in T_{sub} forms the action set. T_{sub} consists of just those sequences in which the subproblem could have been used. Such a sequence ends in a state, which when abstracted, matches the goal state and starts from the earliest state that does not "break" the abstraction. T_{rem} consists of the remaining sequences with one addition: in place of each portion removed, a single step is added representing the option that solves the subproblem.

Returning to our taxi example for a moment, suppose the trajectory is "ENNNPWWWWD". The suffix tree might generate "NN" as a common action sequence. This would become our seed. To find the subproblem goal, we extend "NN" to "NNN" and finally to "NNNP" at which point the *Pickup* action breaks our state abstraction of just `taxiLocation`. The goal would then be the state immediately prior to the *Pickup* action: `taxiLocation=NE`. To decompose the trajectories, we look for all states in which `taxiLocation=NE`. For each such state, we extend backwards in time until a step breaks our state abstraction. In our example, this yields "ENNN", which becomes our T_{sub}. If the option corresponding to our subproblem is named "0", then the remaining trajectory would be "0PWWWWD".

Once we find the subproblems, they are added to *acc* along with their score. We score a subproblem by $C/(C'+C_p)$ where C is the computational cost of solving the base SMDP, C_p is the cost of solving a subproblem p, and C' is the cost of solving the remaining problem. Each iteration of VI has complexity $|S|^2|A|$. It requires $min(L,H)$ iterations where L is the maximum optimal path length and H is the horizon induced by the discount factor γ and precision parameter ε. Assuming H is sufficiently large, we estimate $C = N^2AL$, $C_p = N_p^2 A_p L_p$ and $C' = N'^2 A'(L - L_p F_p + 1)$ where N, A and L refer to the number of states, actions, and maximum optimal path length in the base SMDP. N_p, A_p, L_p, N', A' and L' are defined similarly but for the sub and remaining problems. The sizes of various state spaces and action spaces are computed directly from the state and action abstraction of the base, sub and remaining problems. We estimate length from the maximum observed in the various trajectories. Similarly, we estimate frequency, F_p, as the average frequency of the subproblem in the remaining trajectories.

3.2. Option creation

Once a subproblem is found, we must solve it and generate an option. Given base SMDP $M = (S[F],A,P,D,R,\gamma)$ defined over state abstraction F, and subproblem $(M,\tilde{F},\tilde{A},\omega)$ where $\tilde{F} \subseteq F$, $\tilde{A} \subseteq A$ and $\omega \in S[\tilde{F}]$, the subproblem induces a sub SMDP $M_{sub} = (\tilde{S},\tilde{A},\tilde{P},\tilde{D},\tilde{R},\gamma)$.

The state space of M_{sub} is simply the abstract state space $\tilde{S} = S[\tilde{F}]$. The action space is defined by the subproblem, $\tilde{A}$. The transition function $\tilde{P}(\tilde{s},a,\tilde{s}')$ is defined by $P(s,a,s')$ and the reward function, similarly, $\tilde{R}(\tilde{s},a,\tilde{s}') = R(s,a,s')$, where $s \in h(\tilde{s})$ and $s' \in h(\tilde{s}';s)$. In other words, the transition and reward between states in the abstract state space is the same as the transition and reward between corresponding states in the base state space. The sole exception is when the abstract state matches the abstract goal state. In that event, the transition is simply a self-loop of unit time and zero reward.

We solve M_{sub} directly with VI. With the resulting $\tilde{\pi} : \tilde{S} \to A$, we can define an option for the subproblem $O = (I,\pi,\beta)$. The policy $\pi(s) = \tilde{\pi}(g(s))$ is simply derived from querying the solution of M_{sub}. The initial set is the set of states that can reach the (abstracted) goal. The terminal function β is simply 1 for all states which, when projected into $S[\tilde{F}]$, match the goal state, and 0 otherwise.

3.3. Option insertion

Given a solved subproblem and the corresponding option, we must insert the option as an action into the base SMDP. This poses some difficulty. While the transition, duration and reward functions of the base SMDP are well defined over all primitive actions, the same is not true for learned options. In order to insert the option as an action into the base SMDP, we must learn the model of the option. That is, how the option transitions and the reward accumulates when it is executed as an action in the SMDP.

Fortunately, we can efficiently derive the model for the option. When solving M_{sub}, we not only solve for the value of each state $V(s)$, but also the expected time it takes to reach the goal state. This can be done by deriving a Bellman equivalent for time, or rather discount: $T(s) = E_{s',t}[T(s')\gamma^t]$, for the greedy policy. Similar to the Bellman equation for value, this equation says the expected time (in terms of discount) to reach the goal from state s is equal to the expectation, over next states s' and time t, of the discount it takes to get to s' times the expected discount of reaching the goal from s'.

Because the value of the goal state is zero, $V(s)$ models the reward of the option. $T(s)$ models the duration of the option and, together with the initial and terminal set of the option, allows us to produce the transition function describing the option. Taken together, we can now define a SMDP M_{rem} augmented with the option. $M_{rem} = (S,\hat{A},\hat{P},\hat{D},\hat{R},\gamma)$ has the same state space and discount factor as the base SMDP; however the other elements change. The action set, $\hat{A} = A \bigcup \{o\}$, now contains the option o. The transition function when option o is invoked in state s leads de-

terministically to the goal state of the option, $h(\omega; s)$, with probability 1 and discount $T(s)$. It accumulates discounted reward $V(g(s))$. This, of course, assumes s is in the initial set of o. Invoking o outside of its initial set results in a self loop of unit time with worst possible reward, *rmin*.

4. Analysis

Oplearn assumes that the model is given. Often, the model is either available or can be learned. A more restrictive requirement is the need for near optimal trajectories. Note that this does not mean we need to have a good solution before solving the problem. The algorithm uses a few example trajectories over a handful of states to generalize a policy defined over all states. As our results will show, (1) just a few trajectories are sufficient and (2) the optimality requirement is loose. These factors combine to make many sources of trajectories feasible (*e.g.,* demonstration, single source planning, *etc.*). When trajectories are not available for a particular problem, a smaller or relaxed version of the problem can often be used.

We use a heuristic based on length, frequency and afforded abstraction to discover options. Domains such as flying through a 3D maze, where the actions are thrust and rotate, is a good example of when this heuristic works well. Moving from one point to another (or even hovering) without stalling, spinning out, or otherwise crashing is a nontrivial control problem that must be solved repeatedly. More importantly, this subproblem only needs a subset of the full feature space: features like goal locations, refueling points, *etc.* are not required. Factoring out basic navigation into a subproblem yields considerable advantage because the state space of the subproblem is significantly reduced, requiring far less computation to solve. Each time the subproblem is used, *Oplearn* will extract savings proportional to the length of the subproblem. Conversely, in a domain such as pole balancing, where there are few subproblems being solved repeatedly and no afforded abstractions, *Oplearn* yields little advantage. In summary, our approach is most appropriate in domains with subproblems and where different actions require different state features. This means sensitivity to how the action set is defined. If all actions require all state features, in the worst case, our algorithm reverts to baseline performance; however, in such cases, little state abstraction is available and a large part of any speedup benefit is precluded anyway.

Finally, let us consider the complexity of *Oplearn*. The overhead associated with finding the best subproblem requires the scoring of all maximal repeated action sequences. In domains that are highly nondeterministic or simply have many optimal solutions to a subtask, there will be many such sequences. However, the set is guaranteed to be linear in the size of the trajectories. To score each action sequence we must split the trajectories into those that belong to the candidate subproblem and those that do not. This requires a linear scan through the trajectories. Thus the cost of finding the best subproblem is $O(T^2)$ where T is the length of the trajectories. The overhead of computing the model of learned options is a constant on top of the cost of solving the subproblem itself. Since, typically, $T \ll N$, the computational complexity of *Oplearn* is no greater than that of the base solver.

5. Experiments

We performed a series of experiments designed to explore the speedup of *Oplearn*, and its robustness to varying numbers of trajectories and the quality of those trajectories. These experiments were performed in the Taxi domain where we could easily create variants to suit our needs. To help gauge generalization, we also performed experiments on a more complex domain based on Wargus. Trajectories for the experiments, unless specified otherwise, were derived from optimal policies (*e.g.,* human provided). Each trajectory is one successful episode from a random start state.

Note that in our results, we use operations (OPs) as a measure of speed instead of raw time. OPs are the number of expected value computations. This is similar to number of backups but accounts for the size of the action set. OPs are machine independent, timing tool independent, and more reliable. Experiments measuring raw time maintain the same trends.

5.1. Speedup

To measure the speedup *Oplearn* yields and how that speedup is affected by the size of the domain, we created variants of the Taxi domain. VI is used as the baseline for comparison. To be fair, so that the trajectories do not offer *Oplearn* untoward advantage, when applying VI, we initialized its value table with value estimates from the trajectories. In these speedup experiments, *Oplearn* was given 10 optimal trajectories.

Figure 1 shows the speedup our technique yields over state spaces of increasing size. We generated different sized Taxi worlds by altering the size of the grid. *Oplearn* initially only yields about a 50 percent reduction in the number of operations. By the time the state space has reached 500k states, however, *Oplearn* requires over an order of magnitude fewer operations.

Examination of the options learned explain this behavior. *Oplearn* discovers options like “navigate to `NE` corner”. As the size of the Taxi world increases, the frequency with which we see navigate options does not change, nor does the state abstraction of the subproblem. The length of the

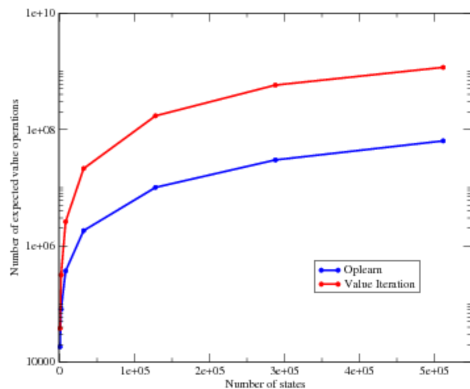

Figure 1. Oplearn and VI in terms of OPs

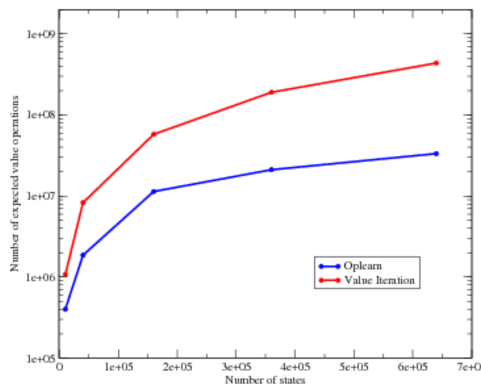

Figure 2. Oplearn and VI in two-person Taxi world

option, however, does change, and this results in increased savings on larger worlds.

We are also interested in how speedup responds to additional state features. To measure this, we performed a similar set of experiments on a two-person Taxi world. In this variant, there are two passengers that need to be conveyed to their destinations. Thus, instead of three state variables, we have five: `taxiLocation`, `passengerA_location`, `passengerA_dropoff`, `passengerB_location`, and `passengerB_dropoff`. As shown in Figure 2, we retain the trends seen previously; however, the slope of the curves suggests possibly greater savings. Many of the discovered options in the two-person world can ignore a larger percentage of the state features than their one-person world counterparts. For example, the navigate option needs only one-fifth of the state features in the two-person world instead of one-third. Similarly, *Oplearn* learns separate options for passenger A and passenger B, allowing more state abstraction in the two-person world: when picking up A, all features *wrt.* B can be ignored and vice versa.

To ensure the results hold for non-deterministic worlds, we ran the same series of experiments on modified Taxi domains similar to the "fickle" version in (Dietterich, 1998). In particular, actions only work 80 percent of the time. The

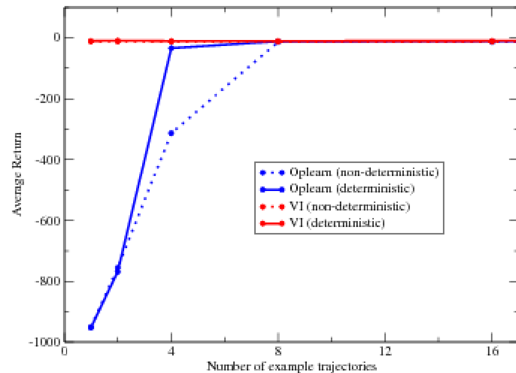

Figure 3. Optimality over the number of trajectories

rest of the time, they fail and leave the state unchanged. Results in the non-deterministic case maintain the behavior and trends displayed above although the savings are even greater. This is due to the ability of options to compartmentalize non-determinism. Due to non-deterministic actions, navigating to the `NE` corner requires varying numbers of steps and accumulated rewards. Many iterations of VI are required before the value of a state will converge, and each of these iterations is over the full state space. By contrast, with *Oplearn*, the navigation task is factored out into a subproblem with an abstract state space consisting of just the taxi location state variable. Although it takes just as many iterations for the value to converge, each iteration is far cheaper because the abstract state space is much smaller. More to the point, the generated option will be deterministic, making the remaining problem much easier. The option "traps" the non-determinism inside itself.

5.2. Robustness

We ran two sets of experiments to explore *Oplearn*'s robustness. The first gauges the reaction to different numbers of trajectories, the second, to varying trajectory qualities.

Figure 3 shows *Oplearn*'s response to the number of trajectories on the simple Taxi domain and its "fickle" variant. *Oplearn* performs near optimal as long as there are a sufficient number of trajectories. If there are too few trajectories, there may not be enough examples of how a subproblem is solved. *Oplearn* infers the abstract action set for a subproblem based on what actions it observes are used to solve the subproblem. As long as a variety of actions are seen, this is fine; too few, however, and the subproblem may not contain all the actions needed to solve it. Consider for example, the subproblem of navigating to the `NE` corner. If there are few example trajectories, it may just so happen that we only ever go to the `NE` corner from the `NW` corner. This would lead *Oplearn* to infer that only *East* is needed to solve the subproblem and produce suboptimal behavior. In practice, *Oplearn* is able to perform near optimally with

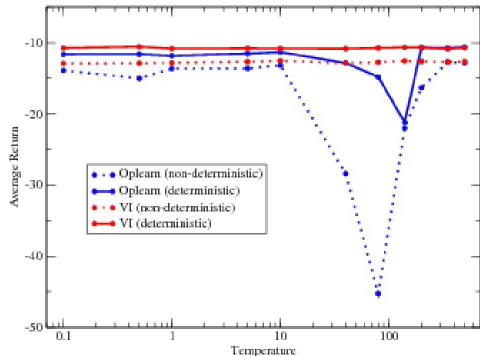

Figure 4. Optimality over increasing noise (temperature)

just four trajectories in the deterministic setting and eight trajectories in the non-deterministic case.

To measure *Oplearn*'s response to the quality of trajectories, we generated trajectories using softmax action selection. In particular, we choose action $a \in A$ with probability $\frac{exp(Q(a)/\tau)}{\sum_b exp(Q(b)/\tau)}$ where τ is a positive temperature parameter, and $Q(a)$ is the expected value of taking action a. High temperatures cause actions to be (nearly) equiprobable while low temperatures cause the action to be greedy. We simulate noisy trajectories by varying τ. Figure 4 shows the results. *Oplearn* was given 10 trajectories for this experiment.

Oplearn performs robustly. Degradation of solution quality as the quality of input example trajectories grows worse is expected. What is interesting is that *Oplearn* maintains near optimality until temperatures of around 10. To give some intuition, at temperatures around 10, there is only a 30 percent chance of choosing the optimal action, just twice as likely as random. The reason *Oplearn* is so robust is because it only uses the input trajectories to identify subproblems. Once found, the subproblems are solved independently. Thus, *Oplearn* routinely performs better than the example trajectories it learns from. The reason optimality falls around 10 is because the trajectories have become so noisy that it begins to affect *Oplearn*'s ability to find good options. One may be surprised that after 200, optimality recovers. This is because by 200 the policy has become roughly random. The trajectories are so noisy that no options are found at all. This results in *Oplearn* reverting to baseline behavior. *Oplearn* suffers worse in the non-deterministic setting due to the additional randomness of the transitions.

5.3. Wargus

To ensure our technique is generalizable beyond simple Taxi domains, we ran additional experiments in a Wargus-like domain. Wargus is a complex, real-time strategy game where players must gather resources, build bases, and train armies to attack each other. A key aspect of the game is the opening when each player is vulnerable and must build up their economies and initial defenses from scratch. The "grunt rush" strategy aims to exploit this weakness by building an army of basic units as fast as possible so as to strike the opponent while they are still weak. For our experiments, we use a version of Wargus where this strategy serves as the goal.

The state space is defined by: `gold`, `wood`, `grunts`, `farms`, `lumbermill`, `barracks`, `blacksmith`, `time`, `location`, `status`. Gold and wood indicate how much gold or wood is available (for simplicity, we modified costs to be in hundreds). Grunts, farms, lumbermill, barracks and blacksmith indicate the number of each that has been built. Location refers to the peon's location. It can be one of three places: townhall, goldmine, or woods. Status refers to the peasant's status, if it is carrying anything, if it is mining, *etc.* Finally, we have added the state variable time indicating the time of day to make the task more interesting (at night, there is some probability of being ambushed by bandits).

The primitive actions are: *Idle*, *GotoGoldmine*, *GotoWoods*, *GotoTownhall*, *Chop*, *Mine*, *Deposit*, *BuildFarm*, *BuildBarracks*, *BuildLumbermill*, *BuildBlacksmith*, and *TrainGrunt*. *Idle* is useful for waiting out the night.

Reward is -10 by default, but there are bonuses and penalties for various events. There a deposit bonus of +1 every time gold or wood is deposited and a building bonus of +2 every time a building is erected. These are part of the scoring at the end of Wargus. There is an army bonus every time a grunt is trained or if a blacksmith is built (the blacksmith gives upgraded weapons and armor) based on the amount by which the army's strength is increased. The army bonus reflects our goal to build as powerful an army as fast as possible. Finally, if the peon is ambushed by bandits we incur a bribe penalty of -40. As our domain is for learning the grunt rush, we end the game when a sufficiently powerful army has been built. We used a γ of 0.9995.

Our experiments indicate that *Oplearn* yields similar results in the Wargus domain as it did in our prior test domains, requiring an order of magnitude fewer ops than VI. As we scaled the size of the problem, by increasing the range of various state variables and the power of the army required, the savings increased. For example, when only an army of power four is needed, VI took approximately 20 million ops while *Oplearn* only needed 2 million, a 10x savings. However, in the larger case where an army of power eight was needed, VI took approximately 80 million ops while *Oplearn* only needed 3 million, a 27x savings. This is because *Oplearn* discovers options like "mine gold", "chop wood", "build a farm", "build a barracks", *etc.* As the size of the problem increases, the amount of work each subproblem chunks away and the frequency of their

use also increases, because more gold, wood and buildings are needed.

Oplearn's solution was near optimal, although it required more trajectories than the Taxi domains. In Taxi, only four to eight trajectories were needed. Here we needed about 15 trajectories before *Oplearn* consistently performed optimally.

6. Related Work

There exists a range of related works in automated problem decomposition. One approach is based on analysis of state features. For example, (Jonsson & Barto, 2006) assumes a dynamic Bayesian network (DBN) transition model and uses it to generate a causal graph depicting how state variables influence each other. It uses this causal graph to produce options for changing the value of various state variables. (Hengst, 2002) also focuses on state features but uses their rate of change as a heuristic for finding options. In contrast with our work, these approaches focus on finding task independent options and do not make use of trajectories.

Closer to our approach is (Mehta et al., 2008). Like (Jonsson & Barto, 2006) they assume a DBN model and use causal analysis but they also take advantage of a single example trajectory to guide the options discovery. While we share many similarities with (Mehta et al., 2008), our focus and approach are different. We focus on subproblem discovery for the express purpose of solving a given problem faster. To this end, we only factor out subproblems estimated to improve running time and infer its model. By contrast, they focus on problem decomposition, aiming for a full MAXQ hierarchy. These differing foci reflect our differing approaches. We discover subproblems based on frequency, length and abstraction affordance. Their work is based on the insight of grouping together consecutive actions that contribute to achieving a specific literal.

Other problem decomposition techniques include approaches based on state clustering and bottleneck detection such as (McGovern & Barto, 2001) and (Mannor et al., 2004), as well as multi-tasks approaches such as (Pickett & Barto, 2002). Unlike the approach taken here, the former typically do not focus on state abstraction and are usually model-free, used in conjunction with Q-learning or similar algorithms. The latter is similar in spirit but aimed to help solve future problems in the same domain, not the current problem at hand.

The general idea of using trajectories to facilitate learning is not new. Apprenticeship learning is based around the concept of taking advantage of example "teacher" trajectories. From this perspective, our method can be viewed as a technique for speeding up apprenticeship learning by discovering and incorporating options.

7. Conclusion

We introduced a method for discovering useful subproblems by taking advantage of example trajectories. We then showed how to achieve significant speedups by solving the subproblems and inserting them as actions into the original problem. Our method produces near optimal policies assuming enough near optimal example trajectories. As seen in our empirical results, it works well with relatively few trajectories and is robust to trajectory quality.

References

Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. *Intl. Conf. on Machine Learning* (pp. 118–126).

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with hexq. *Intl. Conf. on Machine Learning* (pp. 243–250).

Jonsson, A., & Barto, A. (2006). Causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*, *7*, 2259–2301.

Mannor, S., Menache, I., Hoze, A., & Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. *Intl. Conf. on Machine Learning* (pp. 560–567).

McGovern, A., & Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. *Intl. Conf. on Machine Learning* (pp. 361–368).

Mehta, N., Ray, S., Tadepalli, P., & Dietterich, T. (2008). Automatic discovery and transfer of maxq hierarchies. *Intl. Conf. on Machine Learning* (pp. 648–655).

Parr, R., & Russell, S. (1997). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems* (pp. 1043–1049).

Pickett, M., & Barto, A. G. (2002). Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning. *Intl. Conf. on Machine Learning* (pp. 506–513).

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

Sutton, R. S., Precup, D., & Singh, S. P. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.

Ukkonen., E. (1992). Constructing suffix-trees on-line in linear time. *Algorithms*, *1*, 484–492.