Q-Cut - Dynamic Discovery of Sub-Goals in Reinforcement Learning

Ishai Menache, Shie Mannor, and Nahum Shimkin

Department of Electrical Engineering
Technion, Israel Institute of Technology
Haifa 32000, Israel
{imenache,shie}@tx.technion.ac.il
shimkin@ee.technion.ac.il

Abstract. We present the *Q-Cut* algorithm, a graph theoretic approach for automatic detection of sub-goals in a dynamic environment, which is used for acceleration of the Q-Learning algorithm. The learning agent creates an on-line map of the process history, and uses an efficient Max-Flow/Min-Cut algorithm for identifying bottlenecks. The policies for reaching bottlenecks are separately learned and added to the model in a form of options (macro-actions). We then extend the basic Q-Cut algorithm to the *Segmented Q-Cut* algorithm, which uses previously identified bottlenecks for state space partitioning, necessary for finding additional bottlenecks in complex environments. Experiments show significant performance improvements, particulary in the initial learning phase.

1 Introduction

Reinforcement Learning (RL) is a promising approach for building autonomous agents that improve their performance with experience. A fundamental problem of its standard algorithms, is that although many tasks can asymptotically be learned by adopting the Markov Decision Process (MDP) framework and using Reinforcement Learning techniques, in practice they are not solvable in reasonable time. "Difficult" tasks are usually characterized by either a very large state space, or a lack of immediate reinforcement signals. There are two principal approaches for addressing these problems: The first approach is to apply generalization techniques, which involve low order approximations of the value function (e.g., [14], [16]). The second approach is through task decomposition, using hierarchical or related structures. The main idea of hierarchical Reinforcement Learning methods (e.g., [4], [6], [18]) is to decompose the learning task into simpler subtasks, which is a natural procedure also performed by humans. By doing so, the overall task is "better understood" and learning is accelerated. A major challenge as learning progresses is to be able to automatically define the required decomposition, as in many cases the decomposition is not straightforward and cannot be obtained *a-priori*.

One common way of defining subtasks (statically or dynamically) is in the state-space context (e.g., [7], [11], [15]): The learning agent identifies landmark

states, which are worthwhile reaching, and learns sub-policies for that purpose. This approach relies on the understanding that the path towards achieving a complex goal is through intermediate stages which are represented by states. If those states are discovered, and the policy to reach them is separately learned, the overall learning procedure may become simpler and faster.

The purpose of this work is to dynamically find the target states which may usefully serve as subgoals. One approach is to choose states which have a non-typical reinforcement (a high reinforcement gradient, for example, as in [7]). This approach is not applicable in domains which suffer from delayed reinforcement (for example, a maze with ten rooms and one goal). Another approach is to choose states based on their frequency of appearance (see [7] and also [11]). The rule of thumb here is that states that have been visited often should be considered as the target of subtasks, as the agent will probably repeatedly visit them in the future, and may save time having local policies for reaching those states. The latter approach is refined in [11] by adding the success condition to the frequency measure: States will serve as subgoals if they are visited frequently enough on successful but not on unsuccessful paths. These states are defined as *bottlenecks* in the state space, a term that we will adopt. A problem with frequency based solutions is that the agent needs excessive exploration of the environment in order to distinguish between bottlenecks and "regular" states, so that options are defined (and learned) at relatively advanced stages of the learning process.

The algorithm that will be presented here is based on considering bottlenecks as the "border states" of strongly connected areas. If an agent knows how to reach the bottleneck states, and uses this ability, its search in the state space will be more efficient. The common characteristic of the methods that were presented above is that the criterion of choosing a state as a bottleneck is *local*, i.e., based on certain qualities of the state itself. We shall look for a *global* criterion that chooses bottlenecks by viewing all state transitions. The Q-Cut algorithm, which will be shortly presented, is based on saving the MDP's history in a graph structure (where nodes represent states and arcs represent state transitions) and performing a Max-Flow/Min-Cut algorithm on that graph in order to find bottleneck states, which will eventually serve as the target of sub-goals.

In order to understand the use of the Max-Flow/Min-Cut algorithm (see [1]) in the context of Reinforcement Learning, let us first briefly review the graph theoretic problem it solves. Consider a capacitated directed network $G = (N, A)$ (N is the set of nodes and A is the set of arcs) with a non negative capacity c_{ij} associated with each arc $(i, j) \in A$. The Max-Flow problem is to determine the maximum amount of flow that can be sent from a source node $s \in N$ to a sink node $t \in N$, without exceeding the capacity of any arc. An s-t cut is a set of arcs, the deletion of which disconnects the network into two parts, N_s and N_t, where $s \in N_s$ and $t \in N_t$. The problem of finding the s-t cut with the minimal capacity among all s-t cuts is called the s-t Min-Cut problem. It is well known that the s-t Min-Cut problem and the Max-Flow problem are equivalent ([1]). There are quite a few algorithms for solving the Max-Flow problem. The running time is in general a low polynomial in the number of nodes and arcs, making the algorithms

an attractive choice for solving a variety of optimization problems (see [1] for further details on the Max-Flow algorithms and associated applications), and recently also for enabling efficient use of unlabeled data in classification tasks (see [3]). The specific method which we will use in our experiments is *Preflow-Push* (described in [8]), which has a time complexity of $O(n^3)$, where n is the number of nodes.

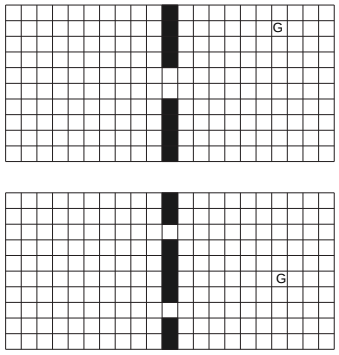

Fig. 1. Simple two room mazes. Goal is marked as "G". After reaching the goal, the agent is positioned somewhere in the left room.

We shall use the Min-Cut procedure for identifying bottlenecks. This process reflects a natural and intuitive characterization of a bottleneck: If we view an MDP as a flow problem, where nodes are states and arcs are state transitions, bottlenecks represent "accumulation" nodes, where many paths coincide. Those nodes separate different parts of the state space, and therefore should be defined as intermediate goal states to support the transition between loosely connected areas. In addition, using a global criterion enables finding the best bottlenecks considerably faster. In order to explain this claim, consider the upper maze of Figure 1. Assume the agent always starts in the left room, and Goal is located in the right room. If the agent visited the wide passage between the rooms, the Min-Cut algorithm will identify it as a bottleneck, even if the number of visits is low in comparison to frequently visited states of the left room. In addition, consider the lower maze of Figure 1. If, for example, the agent reached the goal a significant number of times and used one of the passages in most trials, the cut will still choose both passages as bottlenecks. In both cases, the efficient discovery of bottlenecks is used for forming new options, accelerating the learning procedure.

After introducing the basic algorithm we suggest to use the cut procedure for recursive decomposition of the state space. By dividing the state space to segments the overall learning task is simplified. Each of these segments is smaller than the complete state space and may be considered separately.

The paper is organized as follows: In Section 2 we describe the Reinforcement Learning setup, extended to use options. Section 3 presents the Q-Cut algorithm. In Section 4 we extend the basic algorithm to the Segmented Q-Cut algorithm. Some concluding remarks are drawn in Section 5.

2 Reinforcement Learning with Options

We consider a discrete time MDP with a finite number of states S and a finite number of actions A. At each time step t, the learning agent is in some state $s_t \in S$ and interacts with the (unknown) environment by choosing an action a_t from the set of available actions at state s_t, $A(s_t)$, causing a state transition to $s_{t+1} \in S$. The environment credits the agent for that transition through a scalar reward r_t. The goal of the agent is to find a mapping from states to actions, called a policy, which maximizes the expected discounted reward over time, $\mathbb{E}\{\sum_{t=0}^{\infty} \gamma^t r_t\}$, where $\gamma < 1$ is the discount factor. A commonly used algorithm in RL is Q-Learning ([5]). The basic idea behind Q-Learning is to update the *Q-function* at every time epoch. This function maps every state action pair to the expected reward for taking this action at that state, and following an optimal strategy for all other future states. It turns out that the learned Q-function directly approximates the optimal action-value function (asymptotical convergence is guaranteed under technical conditions, see [2]), without the need to explicitly learn a model of the environment. The formula for the update is:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(n(t, s_t, a_t)) \left(r_t + \gamma \max_{a \in A(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

where $\alpha(n(t, s_t, a_t))$ is the learning rate function which depends on $n(t, s_t, a_t)$, the number of appearances of (s_t, a_t) until time t.

We now recall the extension of Q-Learning to Macro-Q-Learning (or learning with options, see [12] and [15]). Following an option means that the agent executes a sequence of (primitive) actions (governed by a "local" policy) until a termination condition is met. Formally, an option is defined by a triplet $\langle I, \pi, \beta \rangle$, where: I is the options input set, i.e., all the states from which the option can be initiated; π is the option's policy, mapping states belonging to I to actions; β is the termination condition over states ($\beta(s)$ denotes the termination probability of the option when reaching state s). When the agent is following an option, it must follow it until it terminates. Otherwise it can choose either a primitive action or initiate an option, if available (we shall use the notation $A'(s_t)$ for denoting all *choices*, i.e., the collection of primitives and options available at state s_t). Macro-Q-Learning [12] supplies a value for every combination of state and choice. The update rule for an option o_t, initiated at state s_t, becomes:

$$\begin{aligned} Q(s_t, o_t) := Q(s_t, o_t) + \alpha(n(t, s_t, o_t))(\gamma^k \max_{a' \in A'(s_{t+k})} Q(s_{t+k}, a') \\ -Q(s_t, o_t) + r_t + \gamma r_{t+1} + \ldots \gamma^{k-1} r_{t+k-1}) \end{aligned}$$

where k is the actual duration of o_t. The update rule for a primitive action remains the same as in standard Q-Learning.

3 The Q-Cut Algorithm

The basic idea of the Q-Cut algorithm is to choose two states, s, t, which will serve as source and target nodes for the Max-Flow/Min-Cut algorithm, and perform the cut. If the cut is "good" (we shall define a criterion for its quality), the agent establishes new options for reaching the discovered bottlenecks. The whole procedure is outlined in Figure 3. We add the details for the steps of the algorithm below.

Repeat:

- Interact with environment and learn using Macro-Q Learning
- Save state transition history
- If activating cut conditions are met, choose $s, t \in S$
  perform *Cut Procedure(s,t)*

Fig. 2. Outline of the Q-Cut Algorithm.

Cut Procedure(s,t)

- Translate state transition history to a graph representation
- Find a Minimum Cut partition $[N_s, N_t]$ between nodes s and t
- If the cut's quality is "good"
  Learn the option for reaching new derived bottlenecks
  from every state in N_s,using Experience-Replay

Fig. 3. The Cut Procedure.

Choosing s and t: The procedure for choosing s and t is task dependent. Generally, it is based on some distance metric between states (e.g., states that are separated in time or in some state space metric), or on the identification of states with special significance (such as the start state or the goal state). In some cases, choice of s and t is more apparent. Consider, for example, the mazes of Figure 1, under the following experiment: The agent tries to reach the goal in the right room, and when the goal is reached, the agent is transferred back to somewhere in the left room. A natural selection of s and t in this case, is to choose s as one of the states in the "returning area" and t as the goal. The reason for this choice is that the agent is interested in the bottlenecks along its path from start to goal.

Activating cut conditions: The agent may decide to perform a cut procedure at a constant rate, which is significantly lower than the actual experience frequency (in order to allow a meaningful change of the map of process

history between sequential cuts), and might depend on the available computational resources. Another alternative is to perform a cut when good source and target candidates are found according to the procedure for choosing s and t.

Building the graph from history: Each visited state becomes a node in the graph. Each observed transition $i \rightarrow j$ $(i, j \in S)$, is translated to an arc (i, j) in the graph. We still need to determine the capacity of the arc. Few alternatives are possible. First, capacity may be frequency based, which means setting the capacity of (i, j) to $n(i \rightarrow j)$, where $n(i \rightarrow j)$ stands for the number of transitions from i to j. Second, the capacity may be fixed, i.e., assigning a constant capacity (say of 1) to every transition, no matter how many times it occurred. The problem with the frequency-based definition is that we strengthen the capacity of frequently visited areas (e.g., early transitions near the source state, where the policy is actually random) over rarely visited areas (e.g., states that are visited just before performing the cut), thus making it more difficult to find the true bottlenecks. Fixed capacity is lacking in the sense that the same significance is attached to all transitions from some state $i \in S$, a deviation from the actual dynamics the agent faces. Our choice is a compromise between the two alternatives. The capacity is based on the relative frequency, i.e., the capacity of an arc (i, j) is set to the ratio $\frac{n(i \rightarrow j)}{n(i)}$, where $n(i)$ is the number of visits at state i. Experiments show that capacity based on relative frequency achieves the best performance in terms of bottleneck identification.

Determining the cut's quality: The idea behind the design of the quality factor is that we are interested only in "significant" s-t cuts, meaning those with small number of arcs (forming a small number of bottlenecks) on the one hand, and enough states both in N_s and N_t ($s \in N_s$ and $t \in N_t$) on the other hand. Let $|N_s|$ and $|N_t|$ be the number of states in N_s and N_t, respectively. If $|N_s|$ is too small, we need not bother defining an option from a small set. On the other hand, if $|N_t|$ is small the meaning is that the area of states that we wish to enable easy access to will not contribute much to the overall exploration effort. In summary, we look for a small number of bottleneck states, separating significant balanced areas in the state space. Based on the above analysis, the quality factor of a cut is the *ratiocut* bipartitioning metric (See [9] and [17]). We define $Q[N_s, N_t] \triangleq \frac{|N_s||N_t|}{A(N_s, N_t)}$ where $A(N_s, N_t)$ is the number of arcs connecting both sets, and consider cuts whose quality factor is above a predetermined threshold. The threshold may be determined beforehand based on appropriate analysis of the problem domain. It is also possible to change it in the course of learning (e.g., lower it if no "significant" cuts were found).

Learning an option: If the cut's quality is "good", then the minimal cut (i.e., a set of arcs) is translated into a set of bottleneck states by picking state j for each min-cut arc (i, j), with $j \in N_t$. After bottlenecks have been identified, the local policy for reaching each bottleneck is learned by an Experience Replay [10] procedure. Dynamic programming iterations are

performed on all states belonging to N_s, using the recorded interaction with the environment. The bottleneck itself is given an artificial positive reward for the policy learning sake.

3.1 Experimental Results

We illustrate the Q-Cut algorithm on the two simple grids of Figure 1. The experiment conditions are the same as in [11]: The agent starts each trial at a random location in the left room. It succeeds in moving in the chosen direction with probability 0.9. It receives a reward of 1 at the goal, and zero otherwise. The agent uses an ϵ-*greedy* policy, where $\epsilon = 0.1$. The learning rate was also set to 0.1, and the discount factor γ to 0.9. A Cut Procedure was executed every 1000 steps, choosing t as the goal state and s as a random state in the left room. If the cut's quality was good according to the above mentioned criterion, a new option was learned and added to the agent's set of choices. The performance of the Q-Cut algorithm is depicted in Figure 4, which presents a 50-runs average of the number of steps to goal as a function of the episode. Comparing Q-Cut to standard Q-Learning (using the same learning parameters) emphasizes the strength of our algorithm: Options, due to bottleneck discovery are defined extremely fast, leading to noticeable performance improvement within 2 to 3 episodes. In comparison, the frequency based solution of [11] that was applied to the upper maze of Figure 1 yielded significant improvement within about 25 episodes. As a consequence, the goal is found a lot faster than by other algorithms, with near-optimal performance reached within 20 to 30 episodes.

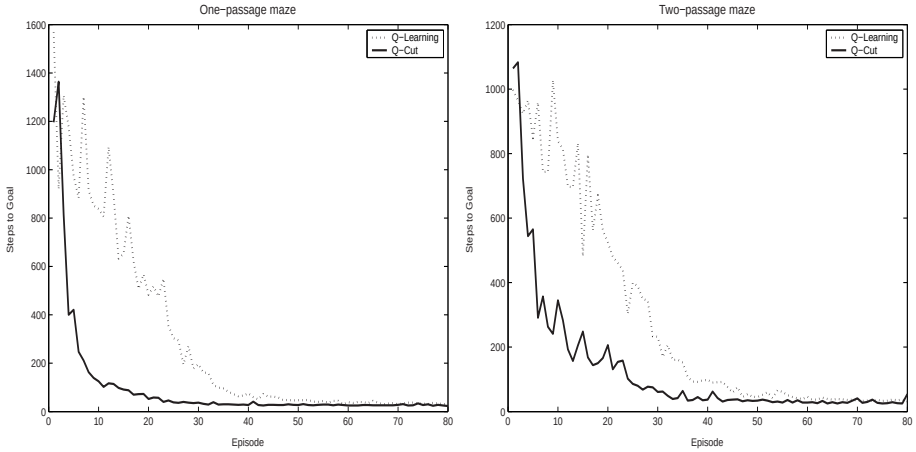

Fig. 4. Performance curves for Q-Cut compared to standard Q-Learning. The left graph presents simulation results for the upper maze of Fig. 1, the right graph presents simulation results for the lower maze of the same figure. The graphs depict the number of steps to goal vs. episode number (averaged over 50 runs).

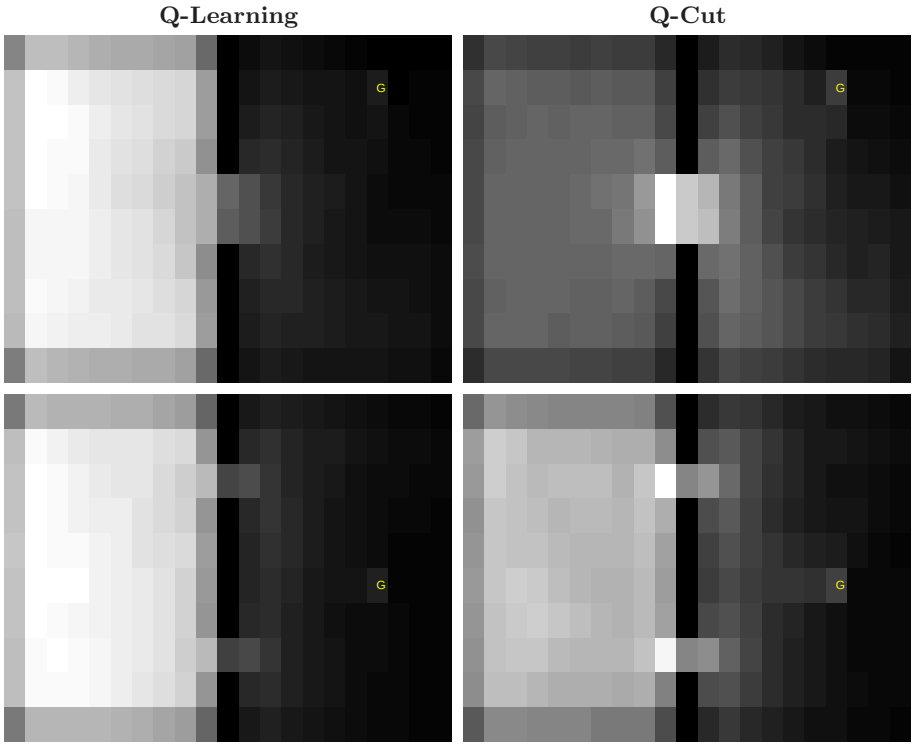

Fig. 5. State frequency maps for both mazes of Fig. 1 (upper maps describe the upper maze of Fig. 1). All measurements are averaged over 50 runs, and were taken each time after 25 episodes. Bright areas describe more frequently visited states. We can see that in both mazes, the Q-Learning agent suffers from exhaustive exploration of the left room. On the other hand, the Q-Cut agent learns the right path towards the bottlenecks, and therefore the bottlenecks themselves are the most visited states of the environment.

In order to clarify the inner working of Q-Cut, we added state frequency maps for both mazes, under Q-Cut and also Q-Learning. Figure 5 presents "snapshots" taken after 25 episodes. Bright areas represent states which were visited often during the course of learning, while darker areas stand for less frequently visited states. We conclude from the Q-Learning maps that the Q-Learning agent spent major efforts in exploring the left room. On the other hand, having discovered appropriate options, the Q-Cut agent wandered less in the left room, and used shorter paths for the passages of the maze (which have the brightest color in the Q-Cut frequency graphs). Being able to efficiently reach the right room, the global policy for reaching the goal is learned in less time, significantly improving performance.

4 The Segmented Q-Cut Algorithm

The Q-Cut algorithm works well when one bottleneck sequentially leads to the other (for illustration, imagine a wide hallway of sequential rooms, where adjacent rooms are separated by one or more doors). In general, if cuts are always performed on the entire set of visited states (which grows with time), chances of finding good bottlenecks decrease. Consider the more complex maze of Fig. 6. To solve the above mentioned problem, we may divide the state space into different segments, using bottlenecks that were already found. If, for example, the agent has found Bottlenecks 2 and 3, it may use them to divide the state space into two segments, where the first contains states from the two upper left rooms and the second contains all other states. In that way, cuts may be performed separately on each segment, improving the chances of locating other bottlenecks (Bottleneck 1, for example). The above idea is the basis for the *Segmented Q-*

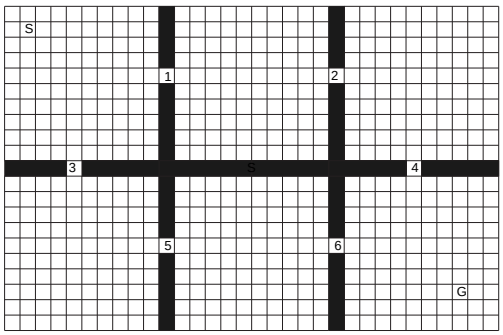

Fig. 6. A 6-room maze. In each episode the agent starts at a random location in the upper left room. Bottleneck states are numbered for illustration reasons.

Cut algorithm. The agent uses the discovered bottlenecks (each of which may consist of a collection of states) as a segmentation tool. We use here a "divide and conquer" approach: Work on smaller segments of states in order to find additional bottlenecks and define corresponding new options. The pseudo-code for the algorithm is presented in Figure 7. Instead of working with one set of states, Segmented Q-Cut performs cuts on the segments that were created, based on previously found bottlenecks. When a good quality cut is found (using the same criterion as in Section 3), the segment is partitioned into two new segments. New cuts will be performed in each of these segments separately. Before performing cuts, each segment is extended to include newly visited states, belonging to the segment. The extension is achieved by a graph connectivity test (a simple $O(nm)$ search in the graph, where n is the number of states and m is the number of arcs representing state transitions), where arcs that belong to a certain valid cut

Initialize:

- Create an empty segment N_0
- Include starting state s_0 in segment N_0
- Include starting state s_0 in $S(N_0)$

Repeat:

- Interact with environment/Learn using Macro-Q Learning
- Save state transition history
- For each segment N, if activating cut conditions are met: *Cut Procedure(N)*

Fig. 7. The Segmented Q-Cut algorithm.

Cut Procedure(N)

- Extend segment N by connectivity testing
- Translate state transition history of segment N to a graph representation
- For each $s \in S(N)$ Perform Min-Cut on the extended segment (s as source, choice of t is task depended)
- If the cut's quality is good (bottlenecks are found)
  - Separate the extended N into two segments N_s and N_t
  - Learn the Option for reaching the bottlenecks from every state in N_s, using Experience Replay
  - Save new bottlenecks in $S(N_t)$

Fig. 8. The Cut Procedure for a segment.

are removed for the connectivity testing procedure. Performing a cut procedure on a segment N means activating the Min-Cut algorithm on several (s, t) pairs, where the sources $s \in S(N)$ are the bottleneck states leading to the segment. The targets are chosen as in the Q-Cut algorithm, based on some distance metric from their matching s.

4.1 Experimental Results

The Segmented Q-Cut algorithm was tested on the six-room maze of Figure 6. The agent always started at a random location in the upper left room. Learning parameters were the same as in the experiments made on the simple maze examples. Results with comparison to Q-Learning are summarized in Figure 9. The Segmented Q-Cut has a clear advantage over Q-Learning.

It is interesting to note when in the course of learning the agent found the real bottlenecks of the environment. On average, a first bottleneck was discovered at the middle of the first episode, the second at beginning of the second episode, and the third at the middle of the same episode. This indicates a fast discovery and definition of subgoals (even before goal location is known), which accelerates the learning procedure from early stages.

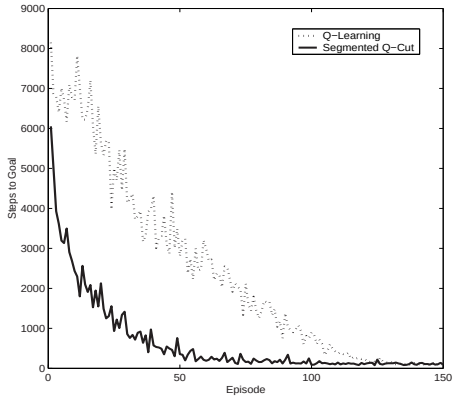

Fig. 9. Performance curves for Segmented Q-Cut compared to standard Q-Learning for the six-room maze simulations. The graphs depict the number of steps to goal vs. episode number. Results are averaged over 50 runs.

5 Conclusion

The Q-Cut algorithm (and its extension to the Segmented Q-Cut algorithm) is a novel approach for solving complex Markov Decision Processes, which are characterized by the lack of immediate reinforcement. Through very fast discovery of bottlenecks, the agent immediately sets its own sub-goals on-line. By doing so, exploration of different areas in the state space, which are weakly connected, becomes easier, and as a by product learning is enhanced. The main strength of the algorithm is the use of global information: Viewing the Markov Decision Process as a map of nodes and arcs is a natural perspective for determining the strategic states, which may be worth reaching. The Min-Cut algorithm is used to efficiently find bottleneck states, which divide the observed state connectivity graph into two disjoint segments. Experiments on grid-world problems indicate the potential of the Q-Cut algorithm. The algorithm significantly outperforms standard Q-Learning in different maze problems. An underlying assumption of this work is that the off-line computational power is at hand, while actual experience might be expensive. Also note that the cut procedure is computationally efficient and is required only once in a while.

The distinctive empirical results motivate the application of the Q-Cut algorithm to a variety of problems where bottlenecks may arise. A car parking problem, a robot learning to stand up (see [13]), and some scheduling problems, are characterized by the existence of bottlenecks that must be reached in order to complete the overall task. Performance of the algorithm in different learning problems, specifically those with a large state-space, is under current study. Additional algorithmic enhancements, such as alternative quality factors and region merging mechanism should also be considered.

Acknowledgements. This research was supported by the fund for the promotion of research at the Technion. The authors would like to thank Yaakov Engel and Omer Ziv for helpful discussions.

References

1. R. K. Ahuja, T. L. Magnati, and J. B. Orlin. *Network Flows Theory, Algorithms and Applications*. Prentice Hall Press, 1993.
2. D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1995.
3. A. Blum and S. Chawla. Learning from labeled and unlabeled data using graph mincuts. In *Proceedings of the 18th International Conference on Machine Learning*, pages 19–26. Morgan Kaufmann, 2001.
4. P. Dayan and G. E. Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, 1993.
5. P. Dayan and C. Watkins. Q-learning. *Machine Learning*, 8:279–292, 1992.
6. T. G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.
7. B. Digney. Learning hierarchical control structure for multiple tasks and changing environments. In *Proceedings of the Fifth Conference on the Simulation of Adaptive Behavior: SAB 98*, 1998.
8. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of ACM*, 35(4):921–940, October 1988.
9. D. J. Huang and A. B. Kahng. When clusters meet partitions: A new density-based methods for circuit decomposition. In *Proceedings of the European Design and Test Conference*, pages 60–64, 1995.
10. L. G. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, 1992.
11. A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning*, pages 361–368. Morgan Kaufmann, 2001.
12. A. McGovern, R. S. Sutton, and A. H. Fagg. Roles of macro-actions in accelerating reinforcement learning. In *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing*, pages 13–18, 1997.
13. J. Morimoto and K. Doya. Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 623–630. Morgan Kaufmann, 2000.
14. S. P. Singh, T. Jaakkola, and M. I. Jordan. Reinforcement learning with soft state aggregation. In *Advances in Neural Information Processing Systems*, volume 7, pages 361–368. The MIT Press, 1995.
15. R. S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.
16. J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.
17. Y. C. Wei and C. K. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE/ACM Transaction on Networking*, 10(7):911–921, 1991.
18. M. Wiering and J. Schmidhuber. HQ-learning. *Adaptive Behavior*, 6(2):219–246, 1997.